UNIVERSITY OF PADUA

Department of Information Engineering

Master Degree in Computer Engineering

Study and Implementation of Blockchain Compression

Matteo Maso student number: 1156281

Supervisor:
Prof. Mauro Conti, *University of Padua*

Co-Supervisors:
Dr. Ghassan Karame, *NEC lab Europe*
Alessandro Sforzin, *NEC lab Europe*

9th December, 2019

Abstract

Blockchain technology is being recognized as the technology innovation that is going to change how society and people interact. After the invention of the Internet, which allows people to break down the barrier of communication among countries, the blockchain has moved the concept of trust from a third party to the protocol. The blockchain technology enable the creation of a decentralized ecosystem, in which transaction can be exchanged and characteristics like persistence, anonimity and auditability are achieved by protocol design.

Despite the excitement, two of the major blockchains, Bitcoin and Ethereum have reached respectively 250 Gb and almost 1 Tb of data, which needs to be replicated among all the users, aiming to participate in the network. Nowadays there is no technology able to reduce the amount of disk space that a blockchain application needs, but also no lean protocol allowing users to participate to the network in a secure and private way. In a future filled with resource-constrained smartphones and IoT devices, the ability to reduce the disk space required by a blockchain is a key factor in enabling the deployment of this powerful technology.

Our study presents two main goals: first, the research of a way to compress an entire blockchain through the use of both legacy compression libraries and a custom compression algorithm, designed accordingly with the Bitcoin structure and data. Our solution enable the long term storage for the blockchain data, useful to maintain the blockchain history for security reason and for data archive.
Furthermore, we do the quantification of the minimum amount of disk space required, in order to guarantee to the user both a good level of privacy and security, throughout any interaction. The result demonstrated the ability to implement a provably secure thin protocol that will allow IoT devices to enter the blockchain ecosystem.

Contents

Chapter 1

Introduction

Nowadays, stirring up worldwide people's interest, "blockchain" has become a buzzword in both Industry and Academia. The origin of the term blockchain is related to Bitcoin; it is a revolutionary software that builds a peer-to-peer network, in which mutually untrusted parties can safely exchange digital currency. Bitcoin comes from an idea, designed and implemented by "Satoshi Nakamoto" in 2008 [1]. Bitcoin, not only arises the researcher's interest but also, has enjoyed a huge financial success thanks to its capital market that reached 237 billion U.S. dollars by the end of 2017 [2].

Blockchain is the underlying technology of Bitcoin - thanks to a new design and protocol - it allows Bitcoin to revolutionise the concept of trust for the entire digital financial sector. The technology lets people exchange transactions, just relying upon mathematical proof, bypassing the expensive trusted third party always used by legacy systems. Blockchain could be regarded as a public ledger, indeed it is a distributed database of all transactions or digital events that the network participants have made and shared. Each record, before the insertion, is verified by a consensus of majority algorithm which runs among participants. Thanks to consensus algorithms and cryptographic techniques, once the information, it can never be erased; Blockchain contains a tamper-proof and verifiable record about every transaction ever made.

Although the most famous blockchain application is Bitcoin, the potential of this technology extends beyond cryptocurrency. The key characteristics of Blockchain that could benefit multiple industries are:

- *decentralization* that is a key characteristic required by nowadays industries, which are increasingly being distributed. Also, a decentralized structure avoid the single point of failure, useful for increasing the level of security;
- *persistence* is almost a precondition for a ledger which aims to provide a historical sense to users' data and transactions;
- *auditability* makes the blockchain's data trustable. It is a tamper-proof strategy, data is correct as long as the system follows the protocol and the majority of users are honest. As soon as data has tampered users is aware of it and the system is not trusted anymore;
- *anonimity* is a key property necessary in a data-driven economy, to benefit from them without the damaging of the human rights.

As described in [3], since Blockchain allows payments' exchange without Banks and third intermediary, it can be used in various financial services such as digital assets and online

payments. Additionally, multiple different areas, like: supply chain [4], Internet of Thing - IoT [5], and Healthcare [6] can take advantages from the adoption of blockchain.

Blockchain is in different studies depicted as a key ingredient for the future's internet systems [7], [8]. It is seen as a powerful way to store and transmit data or value in a fast and secure way. However, Blockchain is facing a challenging technical issue that currently sets a strict upper bound for its adoption and usability - its space requirement. Nowadays, devices that build a blockchain network need to have at least a few hundred gigabytes of space. Furthermore, the challenge scenario is even worst considering the massive amount of data produced by people, that future's internet system needs to cope with. Every day, people worldwide produce 2.5 quintillions of bytes [9]; managing this amount of data with a system based on Blockchain means that each user's device and IoT product needs to be equipped with expensive hardware and that is unreasonable.

Our research aims to broke the space requirement barrier, enabling resource-poor devices to participate and benefit from Blockchain's systems.

1.1 Research Goal & Scope

As mentioned above, we aim to solve the blockchain's space requirement issue. The technology needs to face this challenge before seeing widespread adoption and become a real disruptive technology for the new Internet era.

The size of the entire data storage of two of the most known Blockchain's instances, Bitcoin and Ethereum, by the end of 2019 reached respectively 250 Gb and 2 Tb. These amount of disk space imposes a high initial barrier, especially for the IoT devices, limiting the adoption of the blockchain and yet lower its benefit that comes from the blockchain network's size.

Initially, we conducted a detailed analysis of the protocol and architecture adopted by Bitcoin and Ethereum. Then, we examined the functionality offered by Blockchain and categorized functions used by users. Based on the last part of the study, considering the services that network's participants require, we divided users within different roles.

Based on new roles' division, maintaining the functionality, we have proposed an innovative way to reduce the space requirement reaching a theoretical lower bound. The new approach allows, the most demanding user, who want to maintain an archive version of the data, to cut their space requirement by 40%. Moreover, our solution allows a normal user, to exchange transaction in a blockchain system using a device equipped with a few gigabytes memory.

1.2 Thesis Structure

The thesis is organized as follows:

- Chapter 2 discusses the fundamental concepts behind the blockchain, Bitcoin's protocol, and Data compression algorithms;
- Chapter 3 shows related works;
- Chapter 4 introduces the research path carried out in the thesis;
- Chapter 5 presents the experiments together with a thorough explanation;

- Chapter 6 analyses the experiments' findings;
- Chapter 7 concludes the thesis.

Chapter 2

Background

This chapter begins with an overview of the blockchain technology, follows a deep and detailed explanation about the Bitcoin protocol. In the end the reader will be introduced to the data compression techniques and the modern compression libraries used in the thesis' experiments.

2.1 Blockchain

The following section contains a technical explanation of Blockchain with few examples and motivations to have a better understanding of the topic. Three of the main concepts are explained: Transactions, timestamp server, and consensus algorithms. Also, the section contains a brief explanation about hashing functions and digital signature, two basic concepts for the realization of a blockchain.

2.1.1 What is a Blockchain?

The blockchain is a distributed system, eventually consistent, where multiple untrusted parties can exchange, without a trusted third party, transactions in a secure way.

This section explains how the Blockchain can be realized from a technical perspective, firstly introducing a model to better visualize the concept and later covering all the fundamental steps to implement the system's properties.

According to the founder of Ethereum, a blockchain can be seen as a specialised version of a cryptographically secure, transaction-based state machine [10]. So, basically this distributed system can be modeled as an infinite state automaton. The blockchain system provides developers and system's architect with a new paradigm. Now it is possible to implement specific models of distributed systems where security and consistency are achieved by system design. The economic system, supply chain management, and distributed software are just a few examples where this statement can be applied.

Once we talk about state machine we should specify what a *state* is in the context of the blockchain. To start with an example, within the economic system environment is necessary to agree over the members' accounts balance. A *state* of the system, can be the list of all the members' account values in a specific moment of the history. Proceeding with the example, once user's account value changes, the system's *state* changes too and the automaton moves its *state* S from S_i to S_{i+1}. Trusting the system's state means that everyone agrees above all the steps done which imply agree over the final *states*.

To make the concept more general, where the blockchain is a distributed database, the *state* is a snapshot of the database's values.

With the term *accepting state*, we refer to the last state of the machine. Also, to make the system evolve from one *state* to a new one, a state transition need to be made. In the following section it is explained what a transaction is.

2.1.2 Transactions

In the blockchain model of state machine, to make a blockchain transaction means to apply a function to the *state* of the system. The set of valid functions, combined with their input, represent a set of rules that characterize the system's properties. These rules are part of the protocol and different Blockchains have different protocols, rules, and properties.

A transaction is a single event, allowed by protocol's rules. Transactions, generated within a fixed amount of time, can be grouped forming a bigger transaction. Moreover, viewing the blockchain as a database, transactions are single records plus specific constraints due to their Blockchain's protocol.

Each transaction modifies a specific value of the system's state. However, each type of state field comes with a specific set of functions and an order for the functions' execution. An example is package delivery. Pick the item, package it and send it are functions which cannot be used in a different order.

Now, we can introduce the concept of sequence related to the transactions; each new transaction is allowed by the previous one and it unlocks the creation of the next one. In the blockchain ecosystem, to ensure that this sequence is respected, every new transaction generates a key that is necessary to unlock and use the created transaction. In this way, only the key's owner can execute it.

Blockchain, to enable this function, uses a cryptographic technique known as *Digital Signature*, which is made using a special class of functions called: *Cryptographic hash function*. Both of the concept are explained below.

Cryptographic hashing function

A hash function is any function that can be used to map data of arbitrary size to fixed-size values. Given X and Y respectively, the input and output message space, $\forall x \in X$, $y = h(x)$ s.t. $y \in Y$ and $|y| = l_y$ where l_y is fixed and depends on the chosen function. The output is called, hash value or digest message.

Properties:

- *efficiency* - the function needs to be easy to compute $y = h(x)$;
- *uniformity* - given p_x (the statistical distribution of our input), $y \sim \mathcal{U}(Y)$ where Y is a fixed lenght message space.

A cryptographic hash function is a hash function that also requires:

- *preimage resistance* - given $y \in Y$ it must be hard to find x such that $h(x) = y$;
- *weak collision resistance* - given $x \in X$ it must to be hard to find $x' \neq x$ s.t. $h(x') = h(x)$;

- *strong collision resistance* - it mast be hard to find $x_1, x_2 \in X$, with $x_1 \neq x_2$, s.t. $h(x_1) = h(x_2)$.

These functions are useful to verify the integrity of exchanged data. Once you have downloaded a large file, it is useful to also retrieve the digest to compare it with the digest produced by the file to verify its correctness. A digest is very small so it is less likely to be corrupted by transmission noise and easier to be securely sent.

In the blockchain system, these functions are used to produce a unique identifier and to verify the integrity of data exchanged within the system.

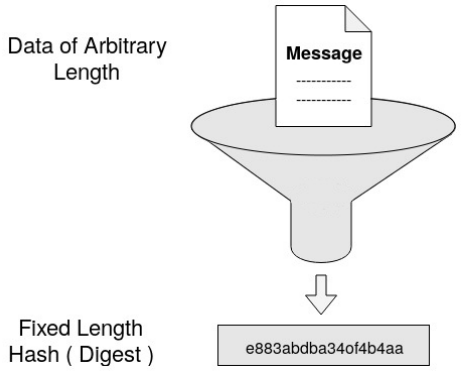

Figure 2.1: Example of an Hash function.

Digital Signature

The term *Digital Signature* comes from the similarity that exists with the handwritten signature however, a *digital signature* is the signature that an entity makes to a digital message.

A digital signature is a security mechanism that allows to obtain data integrity and accountability. Thanks to the use of the asymmetric keys it also offers Non-repudiation.

Properties:

- *Data Integrity* allows everyone to verify the message's integrity just using the creator's public key.
- *Accountability* user's is able to be sure about the creator's identity of all the data.
- *Non-repudiation* connect directly a user's identity with its messages. The creator cannot lying about the creation of the message. This property is essential for money transactions or any trusted system.

The protocol for the creation and verification of a digital signature is made by two steps. Both the signer and verifier user needs to agree over a specific cryptographic hash function and a couple of signature and verification function. The sender user needs also to create a pair of public/private keys and share the public key with the intended receiver. Then, as we can see in the Figure 2.2 during the signature phase it is used the cryptho-hash and signature functions with the private key. Later, the verifier can uses the same cryptho-hash function and the verify function with the public key.

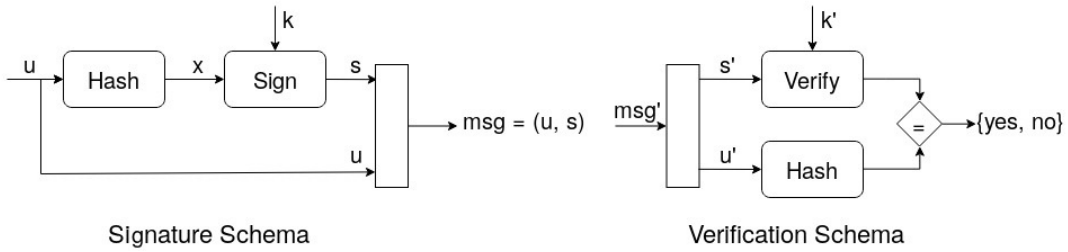

Figure 2.2: Signature and Verification schemes to use digital signature for data integrity.

- Cryptographic hashing function which allows to convert an arbitrary length message into a fixed length digest.
- Signature function allows to create a one way tag t from the u digest x and prive key k.
- Verification function allows to retrieve x from a tag t and the public key k' associated with k.

The Digital signature in the blockchain environment implies that each user owns a pair of a public/private keys. When a user wants to send a transaction she needs to sign it and spreads the transaction combined with the signature into the network. Then, each user that belongs to the network is able to verify the transaction received by using the public key of the sender [11]. The process is shown in the Figure 2.3.

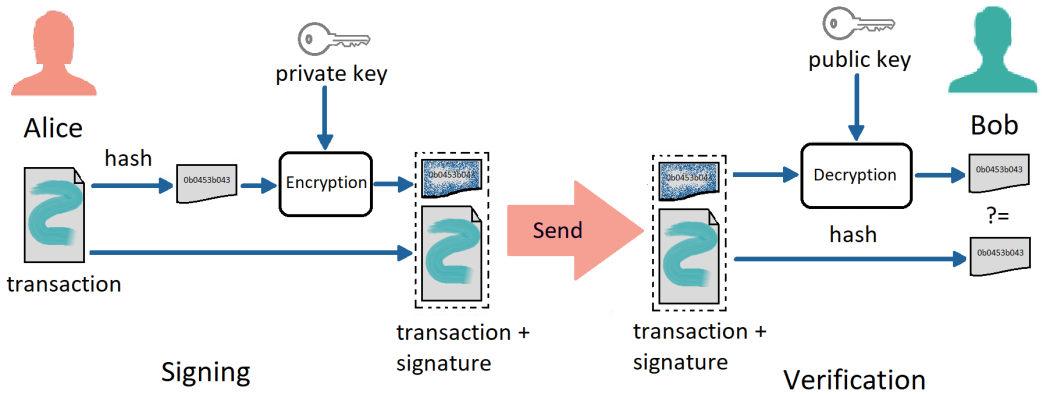

Figure 2.3: Digital Signature used in the blockchain [11]. Two users, Alice and Bob, exchange a message. Alice send a message to Bob who checks the integrity of the message.

Using the digital signature applied to a transaction is possible to exchange transactions in a safe and secure way. However, with this instrument only, Blockchain is not able to guarantee the sequential constraint of transactions' execution.

2.1.3 Timestamp server

As long as we use the technique introduced before, we are able to identify which user has made a specific transaction and if the transaction was created using the correct key. However, the method suffers to the "Double spending attack", which means that there is no way to prevent a user to create a new valid transaction, sending its spent money to another user. It is possible to see that the money is spent twice, see Section 2.4, yet no one can know which is the first transaction created, and which is dishonest.

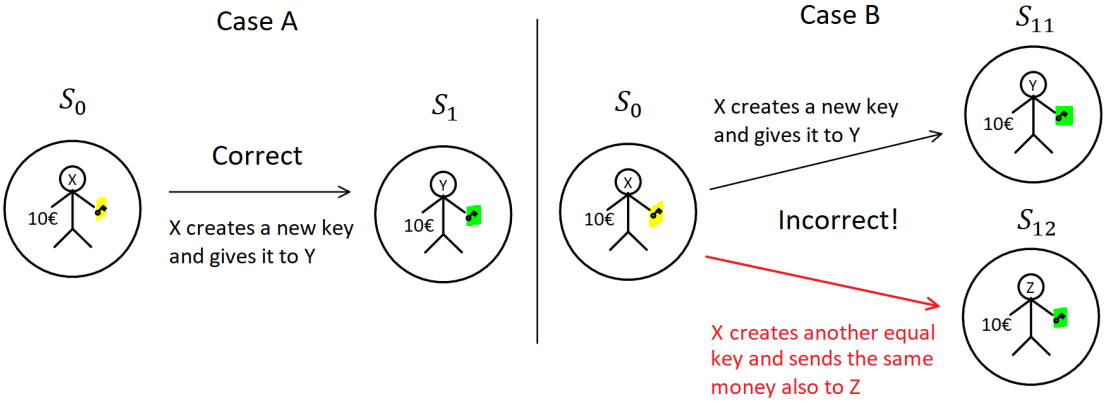

Figure 2.4: The double spending problem, the Case A is correct, instead Case B is wrong since it is ambiguous. In the case B, the user X sends the same money to two different users. This scenario does not exist with a physical currency since it is not possible to give the same banknote to two different people in the same money.

The *Double spending problem*, before the adoption of the Blockchain, was solved relying on a trusted third party, that checks everything and guarantees that a user had spent its money only once. To solve it, is necessary to provide transactions with a time meaning, in this way the first transaction is the valid one. The solution that Bitcoin proposed is to distributed transaction inside chunk called Blocks. The timestamp server works in this way: each new block is hashed within the old hash to form a new hash, as we can see in the figure below 2.5. In this way, each transaction assumes a time meaning, provided by its block's height. A transaction inserted into the block B_i is younger than a transaction inside block B_{i+3}.

The block is created incrementally, thereby once a user creates a transaction and puts it inside a block, it cannot create a new transaction and claimed it to be the newest because the created transaction can only be inserted in the front of the chain. Besides if the user try to tamper the old block, inserting the crafted transaction, the block's hash changes, breaking the entire blockchain's validity.
Before to explain the structure of the blocks is necessary the introduction of a special tree structure, mostly used in the blockchain ecosystem.

Merkle tree

In cryptography and computer science, a Merkle tree or hash tree is a data structure where all leaves are labeled with the hash of desired data and each internal node is labeled with the cryptography hash of its children's label combined. It is a complete binary tree, thereby the deep is $\log_2 n$ where n is the number of leaves.

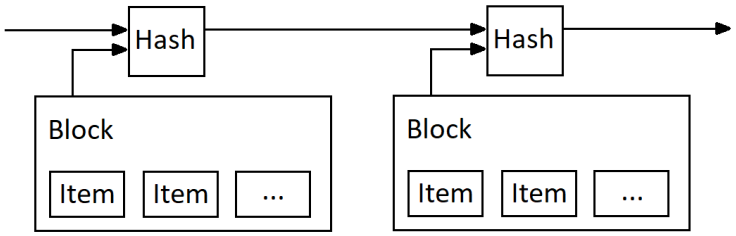

Figure 2.5: Hashing Timestamp [1]. The hash of every block is linked with the hash of theirs predecessors.

The Merkle tree is useful for both checking out the correctness of an entire list of data and verifying efficiently a single part of a data store.

The hash value of the Merkle tree root node is a cryptographic fingerprint about all the leave's label, which means to be a unique shadow of all the data hashed in the leave. Also, the root node does not contain information about the depth of the tree, making the tree difficult to falsify.

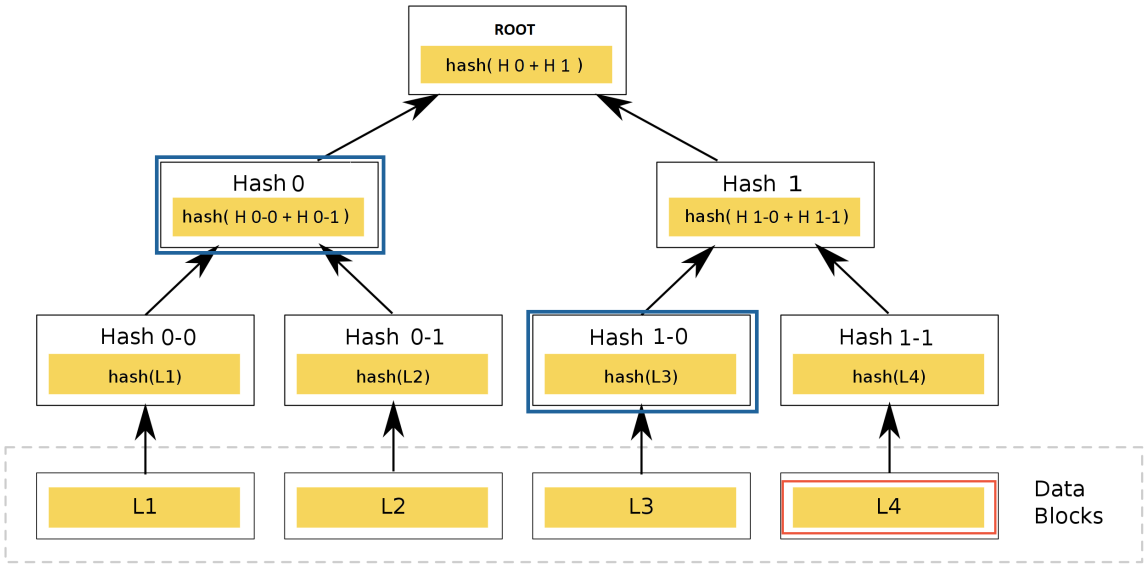

Figure 2.6: The figure describes a standard Merkle tree, build from 4 items which are labeled L_i.

The Figure 2.6 contains an example of a Merkle tree, which is built from four data blocks. The next paragraph describes the Merkle branch.

The *Merkle branch* is one important concept in the Merkle tree. Given a leaf x, a branch called b_x that started from x to the root, the Merkle branch related to x, is composed by all the sibling nodes of b_x's nodes.
As we can see in the Figure 2.7, on the right side, the red leaf depict a random x, the green color underline the b_x and the blue nodes are the Merkle tree related to x.

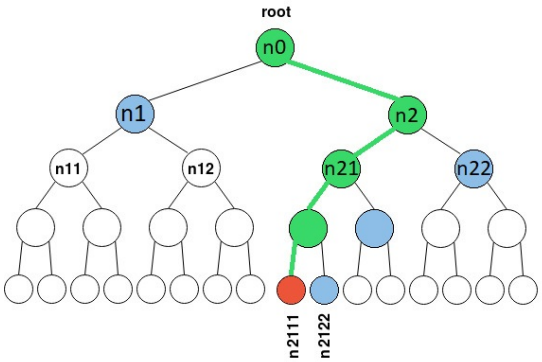

Figure 2.7: Merkle tree branch.

Given the root hash and the data to verify if the root hash corresponds to the data's is necessary to rebuild all the Merkle tree and verify if the new root hash corresponds with the old one. However, to verify if a particular data block belongs to the given root hash, it is necessary to retrieve the Merkle branch associated with the data blocks and to compute as many hashes as the depth of the tree. Normally, looking for the presence of an element inside a set of elements is necessary a linear number of operation using an hash table. However, we can lower the number of operation using a tree structure, so the number of operations is just $\log_2(leaves)$.

Blocks

As we can see in the Figure 2.8, a block is organized in two sections: the `header` and the `body`.

The `body` contains:

- the number of transactions;
- a list which contains all the transactions.

The `header` contains:

- block version, to specify which protocol the block structure respects;
- timestamp, current time in seconds in the universal time since January 1, 1970;
- parent block-hash, used to link a block to its previous one;
- Merkle tree root hash, is the label of the Merkle tree build from the transaction list, which is contained in the block. It is useful to include a fingerprint about all the transaction saving lots of space in the block header;
- nonce, a special number used in the consensus process; a detailed explanation is available in the section: 2.1.4.

In the following Figure 2.9, is represented as a common blockchain's block sequence. All the blocks are linked together, differently from the first one that is called *Genesis block*;

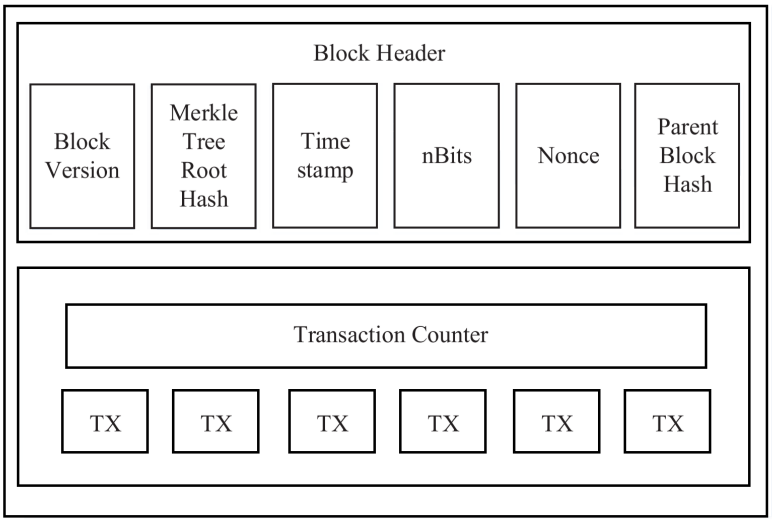

Figure 2.8: The high level block's structure which shows its main components.

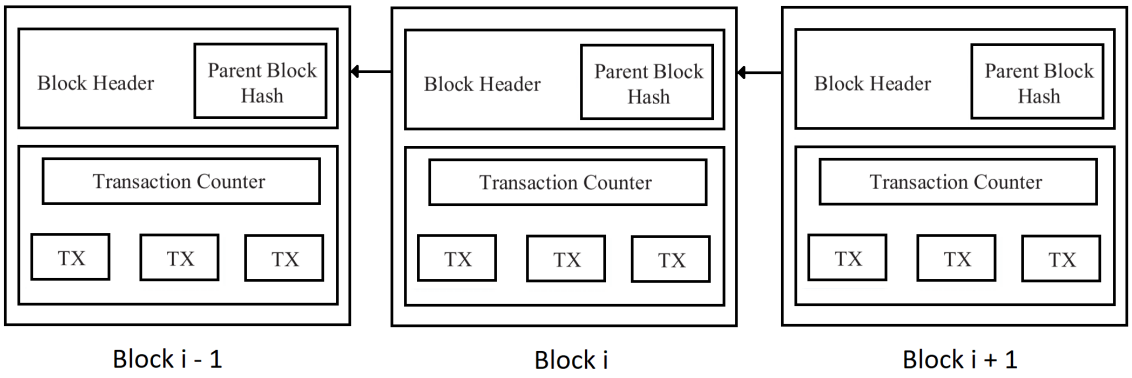

Figure 2.9: An high level blockchain architectural view. Every block's header contains the hash of its ancestor.

it is created only once, during the initialization of the system and from then all the chain has to be rooted there.
The blockchain is a distributed system that aims to reach a majority consensus. Moreover, reaching a majority consensus in a distributed system is a revision of the *Byzantine general problem* which is introduced in the following section. In the section are also described the more important consensus algorithms that are used in the blockchain.

2.1.4 Consensus Algorithm

A consensus algorithm may refer one of several protocols for solving the consensus problem. In the field of computer science the consensus problem aims to achieve an agreement on the current state of a distributed system.

The first consensus protocol suggested in the literature aimed to solve the *Byzantine General Problem*. The *Byzantine General Problem* describes a group of generals of the Byzantine army camped with their troops around an enemy city. Communicating only by messengers, the generals must agree upon a common battle plan. However, one or more of them may be traitors who will try to confuse the others. The problem was generalized as follows [12].
A commanding general must send an order to his n-1 lieutenant generals such that:

- all the loyal lieutenants obey the same order;
- if the commanding general is loyal, then every loyal lieutenant obeys the order he sends.

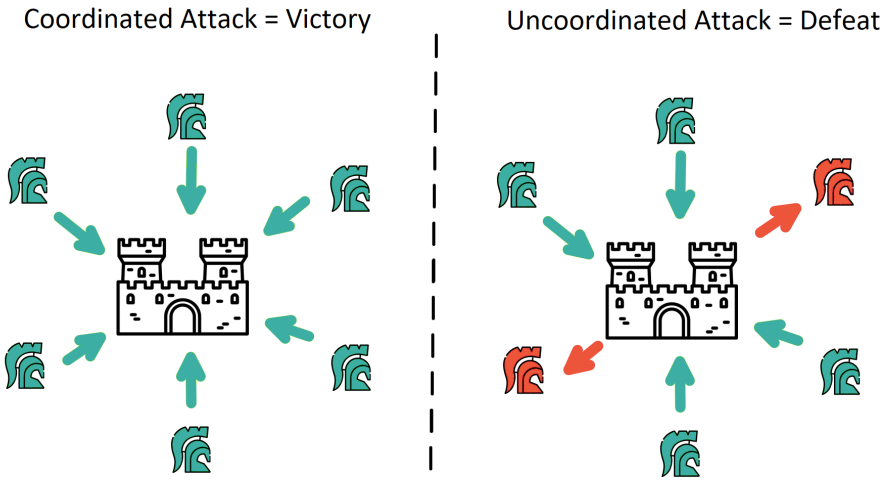

Figure 2.10: On the left side we can see how a coordinate attack succeed, on the other side the defeat caused by an uncoordinated attack.

A consensus algorithm aims to provide a way to agree above a common data value or system's state in a distributed and untrusted system. Finding an agreement between different entities, with conflicting information, in order to maintain the integrity, the operativity and the consistency of a system is the purpose of the consensus algorithm and what the blockchain needs as a distributed, decentralised system used among untrusted entities.

Below, it is introduced and explained the most used consensus algorithm among blockchains: the Proof of Work.

Proof of work

With the term *Proof-of-Work* - PoW we refer to the consent at the base of the blockchain network. The blockchain uses this algorithm to validate transactions and to create new blocks. The PoW encourages the miners to compete in the execution of this algorithm in exchange for a reward. The users who participate in the PoW are called *Miners* and the entire process is called *Mining*.

The PoW forces participant to solve difficult mathematical problems, which on the contrary are design in a way such that the solution is easy to be verified. The mathematical problems usually are based over hash functions, starting from a given input, like a block, is necessary to modify the input accordingly until a digest that satisfies required properties is found. To solve this problem is required a huge amount of CPU power, brute force technique are used because there is not an efficient algorithm known to solve these type of problems.

The growing velocity of a blockchain depends on the difficulty of the problem. The problem should not be too complicated otherwise the chain does not grow, on the contrary, cannot be too easy otherwise the blockchain became vulnerable to external attacks. Also, since every node needs to be able to verify the solution and not all of them has a large CPU power, the solution needs to be easy to verify, even from the poor devices. It is likely that the blockchain growing velocity needs to remain constant, so the difficulty of the problem is chosen accordingly and could change over time; the difficulty increase with the size increasing of the network since more CPU power is involved.

How it works is shown in the Figure 2.11. Miners collect new transactions from the network, fill them into an empty block and start mining it until a desired digest is found. Then, the winner spread the block mined to the others and everyone starts mining a new block again. The hash of the father block is included in the header of the soon, and so on so far, building the chain structure.

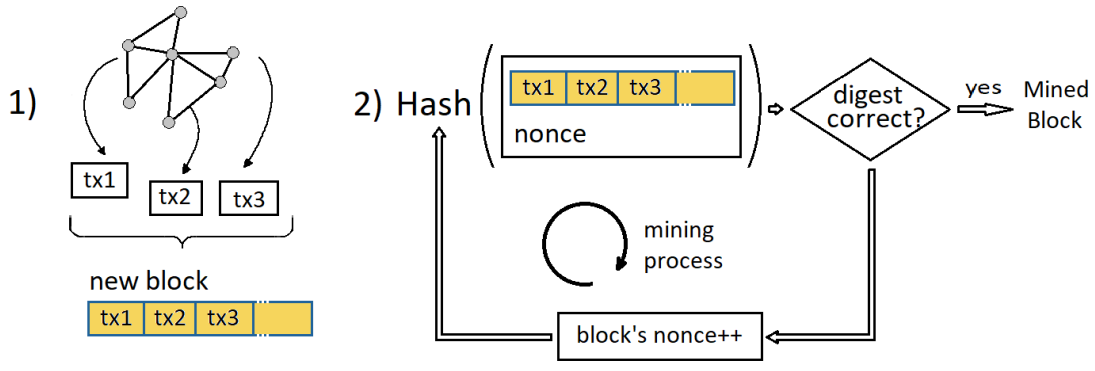

Figure 2.11: The Figure shows: first, the process of block's creation on the right side, then the mining process on the left side.

In the Figure below 2.12 it is represented how the blocks are linked together forming a chain, each block contains the hash of its predecessor. This structure guarantees that each block contains a cryptographic fingerprint of all its predecessors. So, once a

malicious user intends to modify a mined block, it has to mine all the subsequent blocks again. In this way, the cost of modification increases with the aging of the blocks.

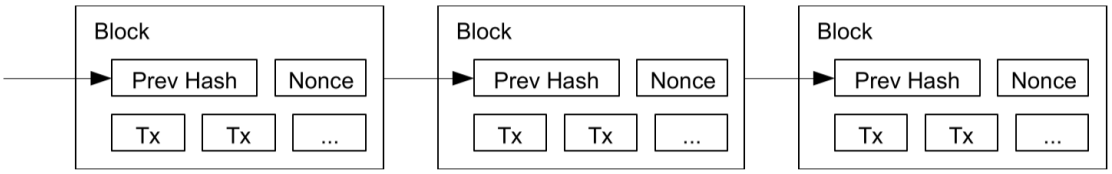

Figure 2.12: Proof of work in blockchain [11].

PoW is a good defense against the Denial of Service attack - DoS. The PoW forces the use of a huge amount of CPU power and time, making an attack very expensive. When a user spends a huge amount of CPU power to conduct a successful attack, the network participants are able to detect it and is likely none use the system anymore since it is compromised. The participants leave the network causing the value of the coin gained by the attacker to collapse.

In the PoW as long as most of the CPU power is held by honest nodes, the whole process is honest [1].

The PoW suffers from the **Fork problem**. The longest chain is considered the correct one and all the Miners keep working to extend it. It can happen that two miners, broadcast in almost the same moment, two different valid blocks. What happens is that a so-called branch in the blockchain occurs and each Miner keeps working on the block which they had first received. It is likely that does not occur the same situation for the next block and so the next block is linked only to one of the branches that become the longest one [1].

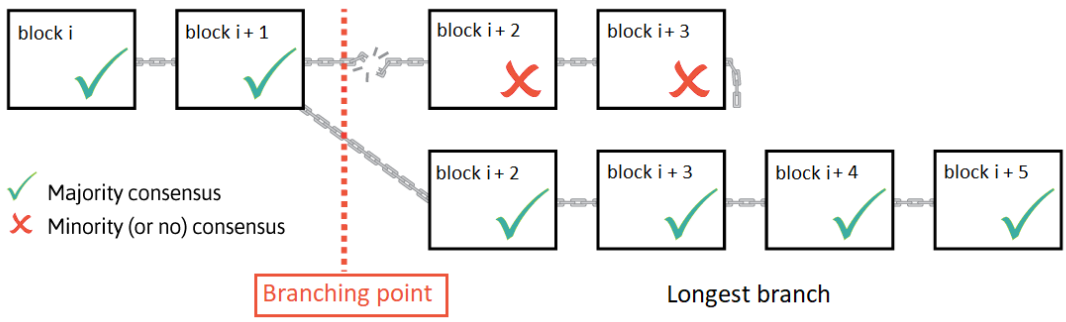

Figure 2.13: The Figure depicts a branch scenario, at block b_{i+2} there is a fork which proceed for the length of two blocks. At block height i+3 only one block is mined and from that point only one branch will proceed and the network discards the shortest branch.

Despite the Proof-of-Work performance the process is a waste of energy, thereby alternatives have been proposed to deal with it and the most relevant solution is the Proof-of-Stake, explained below.

Proof of Stake (PoS)

The Proof-of-Stake - PoS, is an energy-saving alternative of Proof-of-Work, it does not require users to invest their energy power in the process. The idea behind PoS is that the more money a user has, the less she wants to be fraudulent. Therefore, users need to prove the ownership of money and their vote's weights are proportional to their amount of money.

However, this method suffers from unfairness and centralization of power, the richest users control the entire validation process; to cope with the issue various solutions are proposed [11].
Some solutions introduce randomization in the selection of voters. *Peercoin* is one of the first Blockchain based on a PoS algorithm, it designs a solution that supports coin age-based selection. Owners of older and larger sets of Coin have a greater probability to be chosen as validator of the next node, King and Nadal on 2012. Sometimes the *Stack* could be something else, in "BurstCoin", 2014, miners are forced to allocate a huge amount of memory to mine a block [11].

Also, exists a variation of PoS called Delegated Proof-of-Stack - DPoS, which changes the role of the Stake-holders, from mining the next block to delegate the miners of the next block [13]. The purpose is to avoid the centralization of decision power. An example of this algorithm is *Bitshares*, which is a Decentralised Autonomous Cooperation launched in 2015.

2.2 Bitcoin

Bitcoin, designed by Satoshi Nakamoto in 2008, is a decentralized peer-to-peer payment system [1]. It defines an electronic coin as a chain of digital signature; transactions are assets' ownership transfer [1]. The Bitcoin network solved the *Double Spending Problem* using two techniques: it ensures, to all the users, an historical knowledge of every transaction ever made and the chain provides transactions with a temporal meaning by hashing them into an on-going chain of record.

The chain is forged by an hash-based proof-of-work, see section 2.1.4 for more details. The topology of the chain is a rooted tree where the longest branch is approved by the majority of the network's participants. Also, the weight of a single user's vote depends on its CPU power, so as long as the majority of the network's CPU power belong to honest users, the chain is safe [1].

Participants are referred with a virtual address, called Bitcoin addresses; to guarantee privacy, every user owns different addresses, which has a pair of public and private keys attached. The pair of keys is used each time a user aims to spend her money.

2.2.1 Transactions

Bitcoin transactions let users spend *Satoshis*[1], it is an ownership transfer of a certain amount of Bitcoin assets from a sender address to a receiver [15].

The Bitcoin protocol has two categories of transactions:

- The **Coinbase** transactions are used in the Consensus Process and as a way to generate and distribute new Bitcoin assets. Every block has 1 Coinbase transaction at the beginning of its list.
- The **Standard** transactions are all the other transactions, which represent the majority.

Structure

A transaction is actually a structured sequence of bytes. In the Figure 2.14 are depicted all the fields in the correct order.
The next list contains an explanation of all the main fields. Also, a technical structure with implementation details is available in Appendix B.

- `Version number` is used to choose the correct set of rules that needs to be used to validate the transaction.
- `Witness Flag` indicates whether or not the transaction contains Witness data.
- `Input and Output list`, as described above, every transaction has two lists with at least one input and one output; both the lists have also a field containing their length.
- `Witness Data` is an optional field, its presence is announced by the flag `Witness flag` and it contains additional data for the transaction's validation.

[1] *Satoshi*: denominations of Bitcoin value, usually measured in fractions of a Bitcoin; 1 Bitcoin corresponds to 100 M *Satoshis* [14].

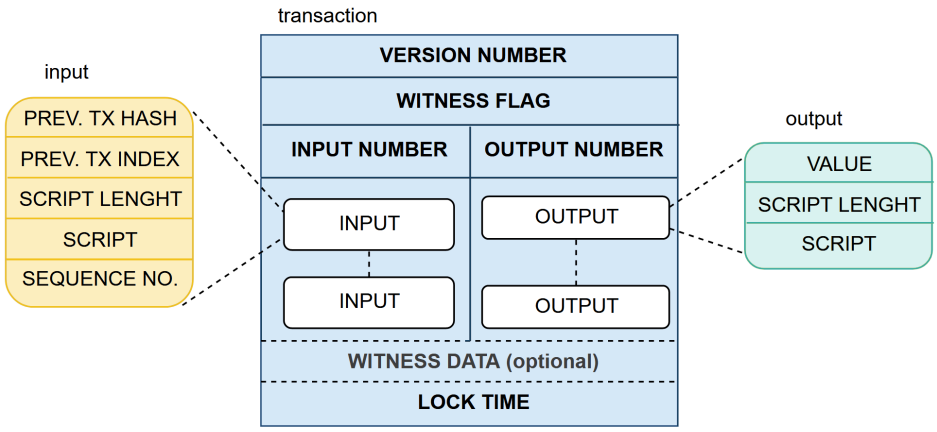

Figure 2.14: Bitcoin transaction's data structure; blue part is the main structure, yellow and green part are subsections, referred as Inputs and Outputs.

– `Lock Time` allows the transaction's creator to insert a lock-time before that the transaction cannot be inserted into the blockchain. This method allows a sender to rethink about the transaction by submitting a new transaction with the same input but a no-lock-time.

Input and Output

So far we have depicted Bitcoin transactions as a transfer of ownership of Bitcoin assets, however, the way they do it is more complicated and it requires a detailed explanation.

Each transaction moves sets of assets, specified within the input, to a set of addresses, specified within the output. The sender also specifies a list of receivers, which are at least one, as outputs. Each Input points to a unique source of assets, which is represented by a transaction's output. Also, every Output is referred to as an *Unspent Transaction Output(UTXO)* until a later Input spends it [14].

The Figure 2.15 shows, focusing on Inputs and Outputs, how a Bitcoin transaction looks like.

Structures of the Inputs and Outputs are important for the authentication mechanism and for the security of the entire blockchain. Both Input and Output structures are represented in the Figure 2.14.

Input

– `Previous transaction's hash`, reference uniquely an existing transaction of the blockchain.

– `Previous tx index`, point a specific transaction output in the transaction's output list.

– `Script length`, contains the length of the next field.

– `Script`, contains information that demonstrate sender ownership of this assets.

– `Sequence number` is a number intended to allow unconfirmed time-locked transactions to be updated before being finalized; not currently used except to disable locktime in a transaction.

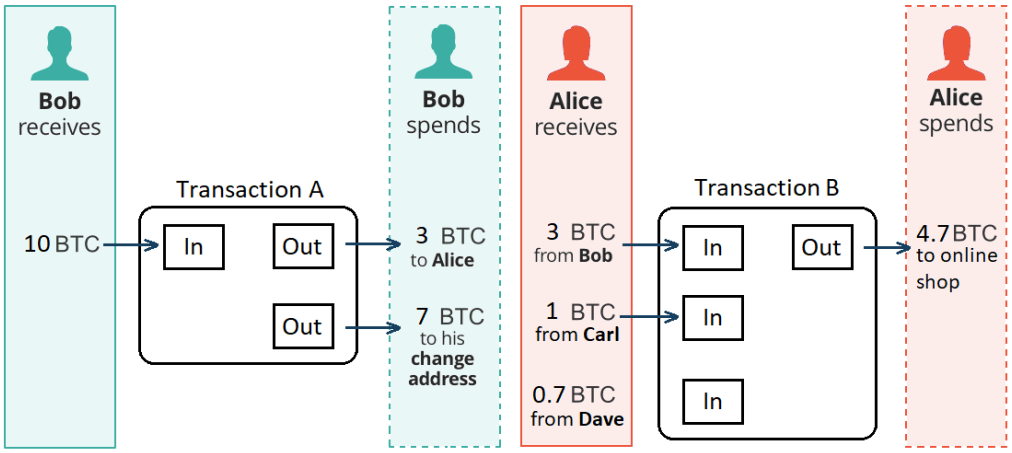

Figure 2.15: The figure shows how the Bitcoins' assets are transferred within transactions; the focus is on the inputs and outputs of each transaction.

Output

- `Value`, amount of asset's value.
- `Script length`, length of follow field.
- `Script`, provides information necessary for the receiver to demonstrate hers ownership. Necessary to spend this assets.

The next section explains the security mechanism of transactions and how they are validated.

Script and Transaction Validation

Bitcoin uses a scripting system[2] to validate transactions. The Script is simple, *stackbased* and has to be interpreted left to right. A Bitcoin Script is a list of instructions, recorded in each transaction, describing how its owner can have access to the transaction's value [16].

As we can see in the picture 2.16, an Input points to a specific Output. To check if an input can be used, you need to combine the `Pubkey Script` of the required output with the `Signature Script` of the Input. Once, the script is executed, the transaction is valid if the script output if True, False otherwise.

Bitcoin protocol supports different types of scripts [18]; the Bitcoin community is still developing new types.
The following list contains all the possible script that we can encounter in the blockchain; the majority of them have a standard structure but a customized part is also present.

- **Pay to Public Key Hash (p2pkh)** is the most common output script. It allows to send a payment to a Bitcoin address[3]. Also, allows users to hide the public key until the Output is spent.

[2]A scripting language or script language is a programming language that supports script. A script is a program designed for a machine to execute a list of actions.
[3]A Bitcoin address is a public key hash encoded in base58check (add cite)

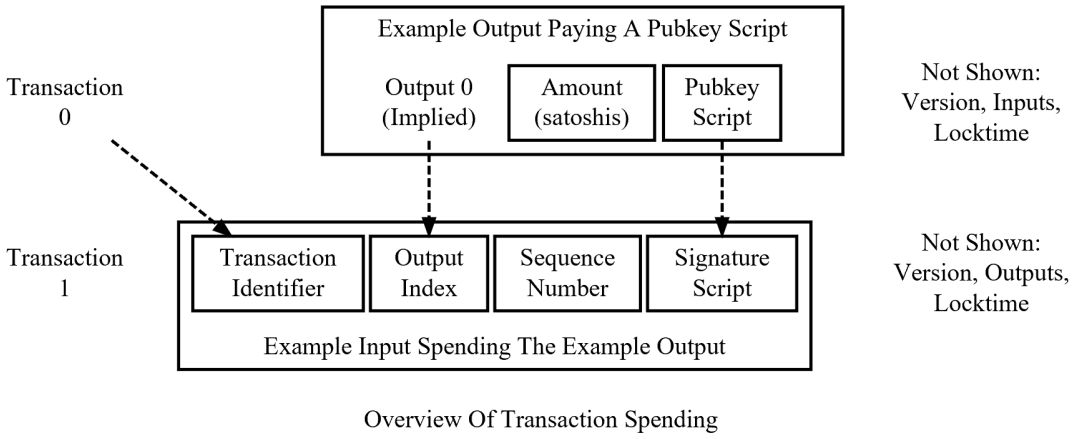

Figure 2.16: Overview of Transaction script works in a Bitcoin transaction [17].

- **Pay to Public Key (p2pk)** is a simplified version of the *p2pkh script*, which is more powerful and secure. This script is not used anymore.
- **Pay to Multisig (p2ms)** allows sending Bitcoin to a group of people, sharing the control of the asset. During the creation phase, a set of public keys and a minimum threshold is required; users can spend the asset fulfilling at least the threshold numbers of constraints. An output with N public keys where M are necessary to spend is called an m-of-n output.
- **Pay to Script Hash (p2sh)** contains the hash of another script, called `redeemScript` that needs to be used with additional data to fulfill the Output script.
- **Data** is not a script but just data; since the script field has just a length constraint a user can use it to fit random data inside the blockchain. The procedure is not recommended since it causes Bitcoin assets unspendable.
- **Custom script**, the name to point the unconventional scripts. Bitcoin protocol provides a set of rules, it is up to the user to combine them to create a secure script. The use of standard scripts is recommended but is not binding.

2.2.2 Block

Bitcoin blocks, as explained in the background section, is a transaction wrapper, used to provide a time-meaning to Bitcoin transactions, ensuring them with security function. The block's maximum size is 1 MB, limited by Bitcoin protocol.

The picture 2.17 shows the Bitcoin blocks structure; formed by a *header* with a set of multiple sub-fields and a list of transactions. Other fields are also present and they are explained in the list below. The *header* contains all the field necessary to provide the blockchain structure and deploy the desired Bitcoin protocol.

- `version`: aim to make the peers aware about which set of block validation rules follow to validate the block; until February 2019 there are 4 different versions and the last one: version 4 was introduced in November 2015.

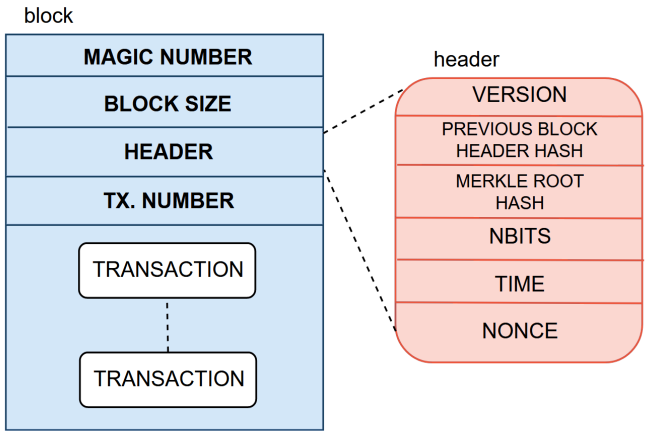

Figure 2.17: The figure shows the structure of a Bitcoin Block; the red part is a zoom over the sub-field *Header*.

- `previous block header hash`: this field is necessary to link this block to its predecessor according to the Bitcoin algorithm.
- `merkle tree root node hash`: contains the hash of the Merkle tree root node, see 2.1.3.
- `time`: it is a Unix epoch time when a miner starts to mine those block.
- `nBits`: it is related to the difficulty that the proof of work function should achieve.
- `nonce`: an arbitrary number, changed over time to modify the header and solve the desired Proof of Work puzzle, see 2.1.4.

The block and chain structures are designed to provide proof that a given transaction was produced and accepted by the majority of the network during a specific moment in the past.

Bitcoin block's structure is made up of *header* and *body* in order to minimize data used in chain security. Each block is connected together however, the hash of the block is just the hash of the header of the block, this way allows to delete transactions without breaking the blockchain [1]. *Header* contains the `Merkle tree root node hash` as a unique fingerprint of the transactions' list contained in the block, using just a small field.

Segregated Witness

Witness is the name of a new data structure proposed as an upgrade of the Bitcoin protocol. This structure is committed to a block separately and does not bellow to the transactions Merkle tree. The data contained within the structure is necessary to check the transaction validity but is not required to determine the transaction effects. In particular, scripts and signature are moved into this structure [19]. Transactions aim to provide ownership transfer of assets, script and signature are only a way to ensure them. This structure allows to keep just important information into the chain and discard validation info after checking them. The transmission of signature data becomes

optional and are transmitted only if a peer is trying to validate a transaction instead of just checking its existence [19].

The adoption of this structure requires a fork of the chain and it needs the consensus of the network. The validity of it started since the 1^{st} August 2017 $\sim$ Block height 478500 when the majority of the network accepted the SegregatedWitness transactions [20].
To achieve the goal was also introduced a new type of transaction index; from now on transactions have 2 IDs.

– The legacy `txid`, which is a double SHA256 of the traditional serialization format:

`[nVersion] [txins] [txouts] [nLockTime]`

– The new `wtxid` which is defined as the double SHA256 of the new serialization plus the witness data:

`[nVersion] [marker] [flag] [txins] [txouts] [witness] [nLockTime]`

2.2.3 Protocol

The life cycle of a transaction in Bitcoin is important and it is represented in the Figure below 2.18.

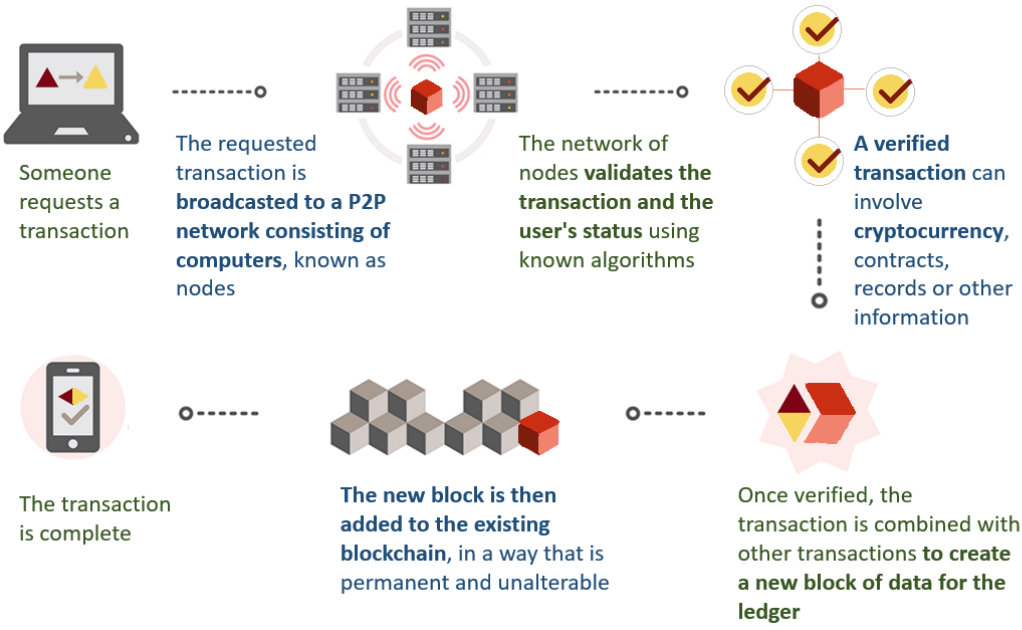

Figure 2.18: This schema shows a Bitcoin transaction lifecycle.

The following bullet point resume which are the main steps of the Bitcoin network and allows new insights. The steps to run the network could be resumed as follow [1]:

1. New transactions are broadcast to all nodes.
2. Each Miner collects new transactions into a new block and it works on finding a difficult proof-of-work for the new block.

3. When a Miner Node solves a proof-of-work challenge, it broadcast the block to all nodes that accept the block after a validation check.

4. The block is accepted when Miners moves to the next block and use the hash of the accepted block as an ancestor.

Mining

Bitcoin uses PoW as consensus algorithm, see 2.1.4. The mathematical problem used consists of building a new block such that the digest produced by hashing the block, is prefixed by at least n zeros. The higher the n, the harder the problem is. The parameter n is estimated in real-time, based on the network's computational power, which must be able to mine a new block every 10 minutes.

When a miner finds the solution she spreads the block with the correct nonce into the network where all the peers can verify the validity of it, by just hashing the block and look at the digest produced. A user accepts a block when she uses it as the ancestor for the next block, also, a block is accepted once $50\% + 1$ of the network's participants accept the block [21].

How the *Incentive* mechanism works?
The blockchain network security relies on a large number of peers involved in the mining process, to encourage them the first transaction of a block starts a new coin owned by the creator of the block. This mechanism is also used to initially distribute coin and once a certain amount of money is created the entire network can relay only above transactions fees [1].

Full node

According to the Bitcoin official guide, a user can participate in the network protocol in different ways. However, the *Full node* is the official type of node, it verifies every block and transaction prior to relaying them to other nodes [22].

The full node is the only type of node which can be a Miner since a miner needs to verify everything before to make the mining functions otherwise, it can happen that it wastes its work if other users verify that she missed something.

The role of Full node is also a key when a new Full node needs to enter in the network since it will provide all the history of the blockchain such that the new node can verify by himself the entire correctness of the blockchain. The role is important also for the maintenance of the decentralization of the blockchain [23].

For a full node client to be fooled, an adversary would need to give a completely new chain with greater difficulty as the local one. However, by definition is not possible as the more difficult chain is by definition the "True" chain.

Another problem is the initial block download - IBD cost, the Full node needs to download the entire blockchain from the beginning and verify every transactions and block. The process can require lots of time, even days and it cannot be used as a wallet until the end of the process [23].

This node as bone of the Bitcoin's network is also the main target for malicious users who want to bring down the Bitcoin system.

Two subcategories are also suggested:

- **Archival** nodes, which are full nodes that store the entire blockchain and can serve historical blocks to other nodes. Their existence is important to allow the creation of new full nodes, which require to re-verify the entire blockchain since the beginning.
- **Pruned** nodes, which still are full nodes but they do not store the entire blockchain.

Since the Bitcoin white paper was disclosed, a "light" way to verify the payment was suggested by Satoshi Nakamoto. The next section describes the *Simplified payment verification* method [1].

Simplified payment verification

The Simplified payment verification - SPV, was introduced to provide a way to verify the validity of a transaction without the necessity of running a full node. The user needs to keep in memory all the blocks' header of the longest chain, which it can retrieve by querying all the network node until it assumes that is the correct one. Once, it receives the transaction to verify, it needs to query the network again to retrieve the Merkle tree's branch, to verify if the transaction belongs to a block in the longest chain.

The method has some assumption to be secure, it relies upon the fact that the majority of the network is trusted and that it can reach the longest chain querying the network, also it assumes that once it asks for the branching tree the user will provide him with that branch. In the end, the verification is only partial and the assumption is that the network accepted the transaction it is true.

This method arises some problem and web is plenty of possible attacks described for this type of client.

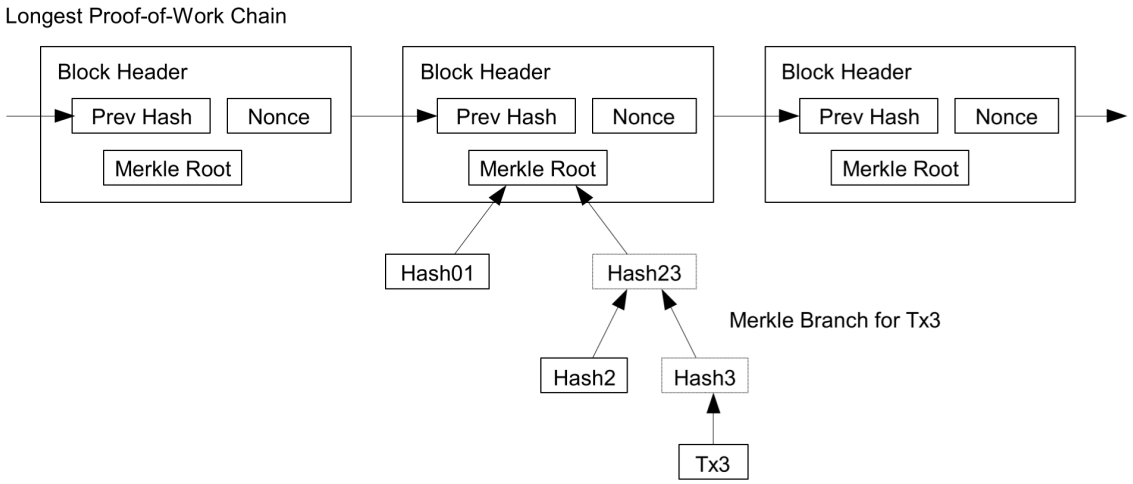

Figure 2.19: The figure shows the chain of the blocks header that a users needs and how a transaction can be verified to be attached to a specific block. The Merkle tree branch is necessary [1].

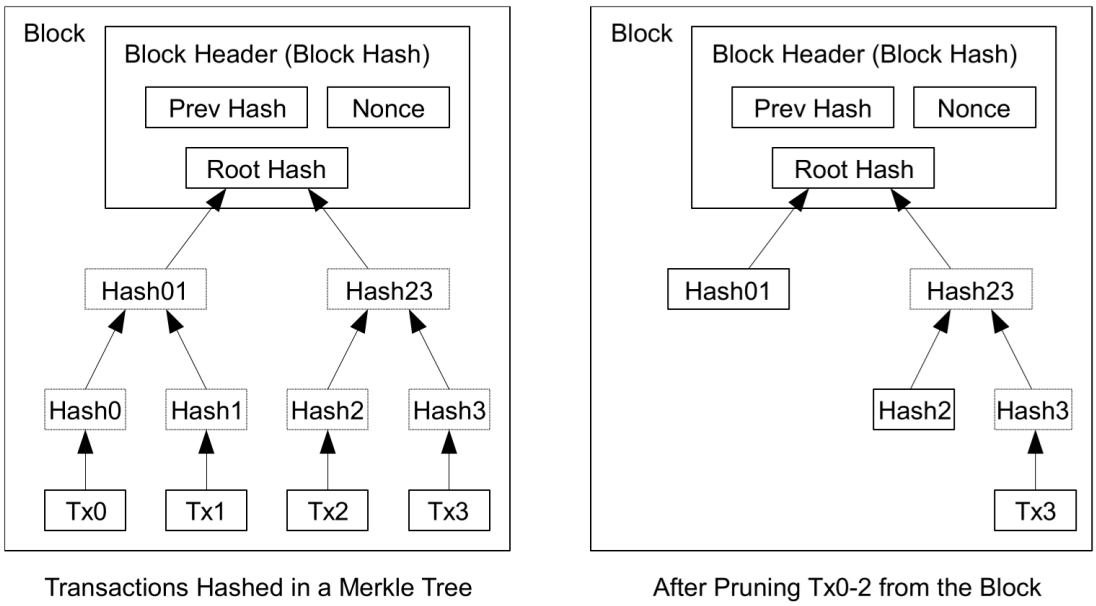

Figure 2.20: The Merkle tree inserted into a Bitcoin block. It shows how just the internal node is necessary to verify if the tx3 is present or not.

2.2.4 Data Storage

This section describes how Bitcoin data storage works. A Bitcoin core client stores, personal configuration data, wallet[4] data, whether a wallet is required, and Blockchain data. An additional data structure is sometimes required too, to increase software performance.

The blockchain data are stored into a single folder called `Blocks` chunked in multiple files; an index structure for this data is stored in another folder named *blocks/index*. A structure called *Chainstate*, is stored inside a folder named `chainstate`.

These structures are explained in more detail with the next three sections.

Blocks' Directory

This folder contains all of Bitcoin's blockchain data. Several files, named sequentially blk00***.dat, contains Blocks in a blob format; the 2.21 represent the structure used. Blocks are not ordered inside the files, also contiguous block can be stored in different files. A structure called *Block's index*, which is a noSQL database, is basically an indexing system of these files.

Also, every blk00***.dat file has a "shadow file" that contains information necessary to revert the blockchain effect caused by the blocks contained in the file. Shadow files are called rev00***.dat, they are structured as a blk file but since they contain less information, are shorter.

The structure of both, blk and rev files, as it is described in the Figure 2.21 is a vector of block's record. Bitcoin blocks have not a standard length so each record is prefixed by 4 bytes of *Network magic number* and 4 bytes integer to indicate the size of the block in bytes.

[4]A Bitcoin wallet, contains all the information regarding user's address, private and public keys. Also, could store its amount of Bitcoin.

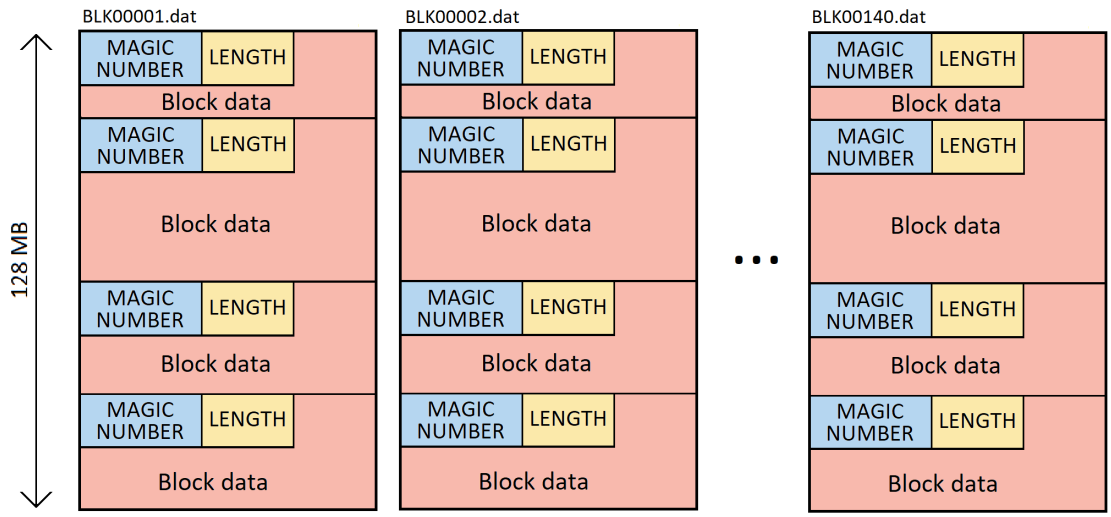

Figure 2.21: The figure shows the so called blk*.dat files in which Bitcoin blocks' data is stored.

Without an indexing structure, every time a user needs to retrieve a block it should be scroll through all files. When a Miner needs to validate a transaction, if blocks are not indexed, retrieve all the transaction's output within different blocks causes an immense waste of time. The following sections introduce the indexing structure.

block/index

The *index* directory is a *block directory* subfolder, it contains a key-value database, which is a LevelDB[5].

This database aims as an index for *blk* files. It is maintained by the Bitcoin client software and it can be rebuilt, providing the following command `-reindex` at the Bitcoin software.

The keys follow specific rules, which are specified in the source code of the bitcoin core: `bitcoin/src/txbd.cpp`. Follow a list with all the information written in the structure:

'f' + 4-byte file number contains metadata about a specific blk*.dat file:

- number of blocks contained
- file size expressed in bytes
- Undo file size
- height of the first block stored
- height of the last block stored
- *Timestamp* of the first block contained
- *Timestamp* of the last block contained

[5]LevelDB is a simple key-value store built by Google. It's used in Google Chrome and many other products. LevelDB supports arbitrary byte arrays as both keys and values, singular get, put and delete operations, batched put and delete, bi-directional iterators.

'b' + 32-bytes block hash contains metadata about specific blocks:

- the hash of the block's ancestor
- the block's height
- number of blk*.dat file that contains this block
- Offset inside blk*.dat
- Offset inside raw*.dat
- version, Hash Merkle Tree Root, Timestamp, size, Nonce, Status and number of transactions.

'B' contains the 32-bytes block hash of the latest known block.

'H' contains the hash of the last block which the database is consistent too.

'R' a flag that indicates whether the software is doing a reindexing process.

'I' is a 4-bytes file number that points to the last used blk*.dat file. It is used to maintain a pointer to the last, not empty file chunk.

'F' a flag to indicate whether the Database is free or need to be stable.

Chainstate

The *Chainstate* is another additional structure, build for efficiency reason by Bitcoin software. It is a LevelDB, like the *block's index*, which contains in a compact format all the *Unspent transactions Outputs* called **UTXOs**. This database is updated on real-time and it could be rebuilt from scratch, by scan all the chain, however, it requires a lot of time [1]. Every UTXO corresponds to a single transaction output not already spent and it contains a pointer where the related transaction can be found in memory plus few metadata.

The reason for the structure is to facilitate miner users, which needs to check the script contained in the UTXO to verify the upcoming transaction.

2.3 Data compression

During the information age, an increasingly larger amount of data is produced, since the network resource and the space storage were limited, multiple techniques had been studied to reduce the space required by data storage.

Compression techniques and algorithms refer to a technique developed to reduce the space used by storing digital content. These techniques consist of pairs of algorithms: the *compression algorithm* which taking an input X, generates its representation X_c, and the *decompression algorithm* which taking X_c as input, generates Y. There are two categories of compression algorithms: *lossy techniques* which allow Y to be slightly different from the original X, and *lossless techniques* that require Y to be equal to X [24].

– **Lossy** compression involves some loss of information, therefore when a data is compressed with a lossy compression technique cannot be reconstructed exactly. For a lot of application, such as voice, image and video compression, some details are lost; despite that, it is very useful and used along with voice or video streaming as a pre-processing technique.

– **Lossless** compression allows to reconstruct exactly the compressed files. It is very useful to store and send text contents, documents, and code.

The *compression ratio* is the most widely used method for measuring the performance of compression algorithms. In particular, it expresses the reduction in the amount of data required as a percentage of the size of the original data [24]; if the original data is 10 GB and the compressed size is 2 GB we are going to say that the Compression Ratio is 20%. Also, different formulas can be used to express the same concept:

$$CompressionRatio = \frac{UncompressedSize}{CompressedSize}$$

$$SpaceSavings = 1 - \frac{CompressedSize}{UncompressedSize}$$

To understand both the model and the benefit of different compression algorithms, a brief introduction about *Information Theory* is essential.
Defining A as an event that is an outcome of some random experiment and $P(A)$ as the probability that the event A occurs, then the *information* associated with A is given by:

$$i(A) = log_b \frac{1}{P(a)} = -log_b P(A)$$

The general idea is that the rarer an event is, the more information it has. However, to calculate the information in bits, base 2 logarithm is necessary. In this way, the information produced by an event with a $P(A) = 50\%$ is $i(A) = log_2(\frac{1}{2})^{-1} = 1$ *bits*, in the same way an event with a $P(A) = 12.5\%$ is $i(A) = log_2(\frac{1}{8})^{-1} = 3$ *bits*.

Coding symbolic data

To compress some data, is necessary to make certain assumptions; for example, we need that data is composed by symbols that belong to an alphabet, moreover, a probability distribution is related with this alphabet. Also, any compression algorithm involves two phases: modelling and coding. During the modelling part, the algorithm needs, given the symbol table, to build the probability distribution and model about each symbol.
The coding part, based over the model plus some additional calculations, encode the data symbols with new symbols, to hopefully reduce the space used by the compressed file; also, a coder called encoder can be independent of the model [25]. A similar structure is also used for the decoder, which used the same model used by the encoder to replace the encoded symbol to the original one, to rebuild the original content.

The model can be updated during the compression process; this brings two different compression types, namely, *static* or *adaptive*, based on whether the models may be updated during the compression process. Also, the system can be *Static* if the model used for compression and decompression are identical or *Asymmetric* when it is not.

Based over the length of the codewords, used before and after compression, algorithms can be classified the next way:

– **Fixed-to-fixed**: each symbol has the same length before and after compression. Ex. A=00, B=01, C=10, D=11.

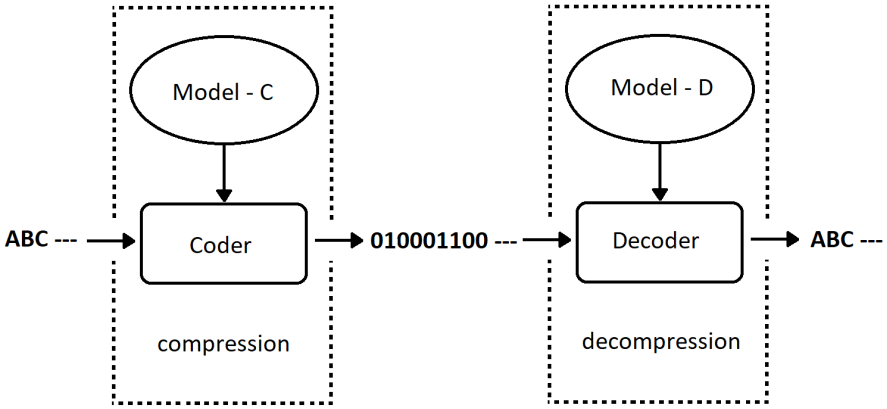

Figure 2.22: The Figure depicts the model of a compression algorithm; *Model-C* is used to represent the model that compression technique uses and *Model-D* for the decompression phase. ABC ... is the source message, and 010001100 ... is the compressed message.

- **Fized-to-Variable**: before compression each symbol has the same length, but different length after compression; the number of bits after compression is based over compression strategy.
  Ex. A=0, B=10, C=101, D=0101.
- **Variable-to-fixed**: symbols can be grouped together before compression, and fixed length has the symbol after compression.
  Ex. ABCD=00, ABCDE=01, BC=11.
- **Variable-to-variable**: symbols are grouped with different length, after compression symbols have also variable length.
  ABCD=0, ABCDE=01, BC=1, BBB=0001.

In the following sections, we explain types of lossless techniques.

2.3.1 Run-length-Encoding

Run-length-Encoding (RLE) is one of the first compression technique used for fax and tv streaming compression. The idea of RLE is to find long sequences of repeated characters and replace them with their shortest repeated subsequence followed by the number of its occurrence.

The technique is also very useful in structured data compression, which is often characterised by long sequences of zeros used for padding. It can be used also for streaming bitmap images, but it is applicable over generals sequences of bits, as we can see in Figure 2.23.

2.3.2 Entropy encoding

One of the most used entropy encodings creates and assigns a code to each unique symbol of the source code which needs to be compressed. The length of the code words is inversely proportional to the amount of information carried by the replaced symbol. The more a symbol is frequent, the shorter the symbol, used to replace it, should be.

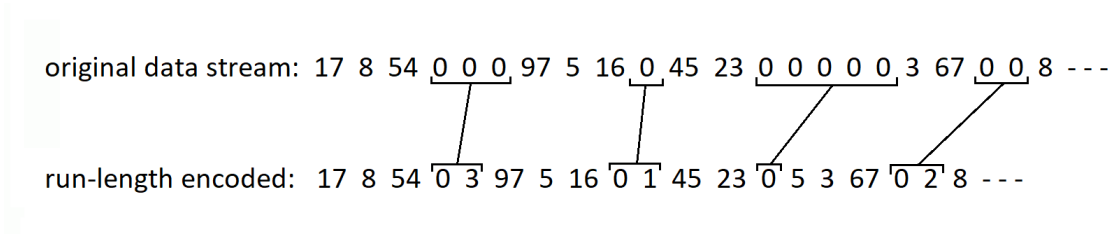

Figure 2.23: Example of bits compression using Run-length-encoding.

According to the *Source coding theorem of Shannon*, the best fit of the code length for a symbol is:

$$-log_b P(x)$$

where b is the number of symbols used to form the output code (2 in a binary encoding) and $P(x)$ is the probability of the input symbol.

Two of the most widely used technique for entropy encoding are "Huffman Encoding" and "Arithmetic Encoding". However, if the entropy model of a stream of data is known before, other technique can be used.

Huffman coding

Developed in 1952, by *David A. Huffman*, the Huffman Encoding is an encoding algorithm used for data compression. The idea is based on the information theory, it aims to find the optimal encoding based on the relative frequency of the original symbols. The technique chooses the best way to represent each symbol, producing an encoding without prefixes - no codeword is a prefix to another code word.

The power of the technique is how it assigns the code word for each symbol. It is a *Fixed-to-variable* technique so it takes each alphabet's symbol of the initial data source with the same length, let suppose to be an ASCII alphabet, orders according to their occurrence in the source data. Then, it produces a tree, as we can see in the figure and the conversion table comes free from the branching label of the tree 2.24.

2.3.3 Dictionary based

The dictionary encoding represents a class of algorithm which has a list with lots of symbols, and it replaces the source file with references to the table whenever a match occurs. The most relevant part of the algorithms is the building of the dictionary. It can be a Static dictionary, to compress English texts, based over a unique English dictionary for example. Or, it can be updated dynamically to compress streaming content.

LZ77 and **LZ78** are the most important algorithms that implement this technique. Both names come from their inventors, *Abraham Lempel* and *Jacob Ziv* and were respectively created in 1977 and 1978. These two algorithms enabled the creation of many future algorithms like DEFLATE, LZMA, and LZO.
From a technical point of view, LZ77 and LZ78 are both dictionary-based however, LZ77 uses a sliding window to compress and update the dictionary that enabled the possibility to compress and decompress the data from a chosen starting point. LZ78 instead rely on a two step process, it builds the dictionary first and compresses the data later. This force the compression and decompression from always from the beginning of the data.

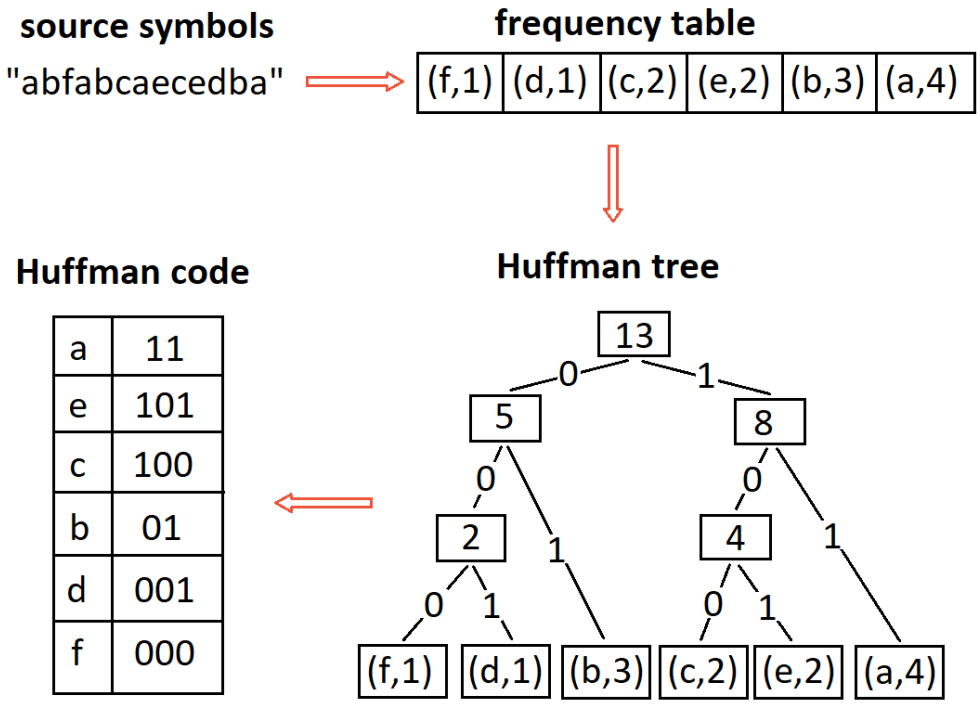

Figure 2.24: The Figure depicts the tree that comes from the example sequence of character using the Huffman technique.

For a streaming environment, both the algorithm can be used and the starting point requirement of the LZ78 is not a limitation since the stream of data can be chucked in multiple chunks and compress them independently or based over a pre-build dictionary.

The algorithm **Lempel-Ziv-Markov chain** known as **LZMA** is a variation of the LZ77 algorithm with a variable dictionary size. It is also used for the `7z` format and in the related software 7-zip.

DEFLATE is another lossless compression algorithm which is a combination of LZ77, LZ88 and Huffman coding algorithms. On the first stage of compression both LZXXs are used to eliminate the string duplication. During the second stage, it is applied the Huffman coding compression which builds an unprefixed tree of non-overlapping intervals, where the length of each codeword is inversely proportional of its frequency.

The following section explains some of the modern Python libraries used to compress generic data. Generic data is data with an unknown frequency model. Voice or video data have a known data model which can be used for compress file with specific software. Compressing blockchain which is structured data with a high level of entropy is necessary a compression library which can build its data model and then compress it.

2.3.4 Modern compression library

The following library is the best one for the generic data compression, which has a Python version. We are going to present their usage, parameter configuration and limitations. The use of python is not a limitation since most of the existing modern algorithm has its Python implementation version.

LZMA

This python library implements the LZMA algorithm compression. The compression function is configurable with many parameters:

- *compression level*: which can be from 1 to 9, with an extra option extreme.
- *dictionary size*: dictionary size in bytes and it can be a number bigger than 4 KB and lower than 1.5 GB.
- *match finder* specify different ways to build the dictionary during the data model creation of the algorithm. The following parameters are available: MF_HC3, MF_HC4, MF_BT2, MF_BT3, or MF_BT4.

The function is able to compress binary data with a size of no more than 2.5 GB, which is a con in the blockchain compression since data are hundred of Gigabytes. During the data model creation, the bigger amount of data a function can analyse, the better would be the compression.

LZO

The Lempel-Ziv-Oberhumer (LZO) algorithm is another lossless compression library optimized for decompression speed.

It allows only a compression level parameter, that can be in the range 1-9.

LZ4

LZ4 is an LZ77-type compressor with a fixed, byte-oriented encoding. It is optimized for compression and decompression speed. This library produced an output compressed file which is a sequence of little compressed chunks. The size of the chunks is configurable.

It allows the setting of two parameters:

- *compression level* an integer in the range 0-16, where the level 16 aims to provide the best compression level.
- *block size*: where the block is a single chunk of data which is compressed together. This parameter can be from 16 KB up to 4 MB.

Chapter 3

Related Work

The Bitcoin white paper has a section named "Reclaiming Disk Space", the space requirement issue has been considered by Satoshi Nakamoto during the Bitcoin design [1]. However, the space required to run a Bitcoin node is still an entrance barrier for many users which aims to use common devices like smartphones. The whole Bitcoin chain has reached 240 GB in the end of 2019 [26]. The chapter is divided into two sections:

- *Bitcoin core software* describes all the solutions already implemented which cope with the space requirement.
- *Test and Tries* is a list of all the researches done and tools provided in the literature which are related to space reduction in Blockchain applications.

3.1 Bitcoin core software

According to the Bitcoin documentation, the most important Bitcoin node is the *Full network node*, the code was initially provided by Satoshi Nakamoto and an even bigger developers community started working on it since, adding new functionality and optimizing the code.

Firstly it is analysed the *Pruning mode* which aims to limit the general space usage by a Bitcoin node through the elimination of blocks from local storage.
Then it is analysed the *Light client*, which implement the Simplified Payment Verification, from a privacy and security perspective.
In the end it is also discussed the *Segregated Witness* which is a new transaction's datastructure to separate the *Script's* information from the security data.

3.1.1 The pruning mode

The *Pruning mode* is a mode in which a Bitcoin software can be executed, introduced in the Bitcoin software in August 2015 with the released version 0.11 [27].
This option allows the usage of a fully validating node without the necessity to maintain a copy of the raw blocks (blk_XXX.dat) and Undo data (raw_XXX.dat) in local. The block index and UTXO database remain unchanged, build from raw data as a normal full node. The block index still maintains blocks' metadata which are used to retrieve necessary data, like specific blocks or transactions from other nodes.

The option does not implement any type of automatism for the choice of the space that needs to be allocated. The user specifies how much space to allot for blocks' data, with a minimum of 550 MB as the only constraint.
This mode is enabled by adding the option `-prune x` before the execution of a Bitcoin node. The parameter x can be selected as follow:

- $x = 0$ means no pruning mode;
- $x \geq 550$ is the maximum number of Megabyte that the software can uses to store the blockchain.

One limitation of this option is that the `-reindex` command, which checks again the entire chain to retrieve metadata and address's state, is disabled.
Also, any time a memory corruption occurs the software deletes the local copy of blk?? and raw?? data to re-download and verify the entire blockchain, since the last chain version cannot be trusted anymore.

Since Bitcoin version 0.12 is also possible to run a wallet while is using a full node in *pruning mode*.
However, old information cannot be retrieved from their local chain copy and they cannot provide the older blocks to others node.

Version 0.14 enables the manual pruning, which needs to be activated by setting the parameter `-prune=1`. Once it is active, the sentence "`command: pruneblockchain [height/timestamp]`" written in the user interface of the Bitcoin software starts the pruning until the chain height or timestamp specified.

This method allows a lot of advantages in term of space reduction, however it is not for free and a lot of related disadvantage. The massive adoption of this running option decrease the health and security of the Bitcoin network, since less and less network's nodes has the ability to verify themselves the correctness of blocks and transaction.
Also, new full nodes and users who are looking to retrieve old information had to rely on others which store the complete version of the blockchain.
However, even if the Bitcoin community still promotes the maintenance of the entire blockchain, without the technology that enable this choice for all the users, it will still remain an utopia.
If there are not beneficial to maintain a local copy of the chain, with the increasing of the resource demanding users will be discouraged and the number of complete chains will decrease, putting the entire Bitcoin network in danger.

Another issue regards the way in which a user can choose the pruning mode. The security of a blockchain increases with its aging, however none of the method implemented allows a user to prune automatically blocks older than a chosen age, based over a fixed security level chosen or suggested.
Also, there are not studies about which should be the correct number of blocks to keep in memory.

3.1.2 Light clients

A *Light client* is a type of Bitcoin node that uses the Simplified Payment Verification already introduced in the Bitcoin White Paper, see section 2.2.3 for more details. This node provides a light way to participate in the Bitcoin ecosystem, users can send and check transactions with a software that uses just a few gigabytes of disk space.

Despite the benefit, the Light client is not a Full node, it does not validates any blocks or transactions by himself and it relies its trust on other Full Nodes, to whom it requests data to fulfill its lack of a local copy of data.
Every Light clients enroll in the Bitcoin's network with a fast initial blocks synchronization which means to download only blocks' header and to check their validity just comparing the hash with their local copy of header's chain.

This client is discouraged from the community since the information cannot be trusted at all, they rely on a third party full node and they have no way to understand if the information received is correct or not.
Also, this method suffers by the lack of privacy and it was proven that if a user owns less then 20 address, asking its related information to the full node, it discloses almost all its information [28].

Thanks to the Simplified Payment Verification 2.2.3 this client are able to verify if a transaction was included or not into a specific block. They have a local copy of all the blocks' header plus the transaction they aims to verify. Firstly they need to send a request to the network in order to retrieve the entire Merkle branch associated with the transaction that they wants to verify. Then, using the Merkle branch and the hash of the transaction they can compute the Root hash of the Merkle tree, by comparing its value with the hash contained in a specific block's header they can verify if the transaction is or not contained in that block.
However, all the local blocks' header are not fully validated and even if they are correct, the retrieved Merkle branch could still be fake, or even worst other nodes can provide a fake one just to convince the user that the transaction is not correct.

3.1.3 Segregated Witness

The *Segregated Witness* transaction is a type of transaction adopted from the majority of the network on the 1st August 2017. This is one of the biggest modification of the Bitcoin structure done to save space in the Bitcoin blockchain [29].
The key idea is to separate the validation data contained in the transaction from the effect caused by the transaction himself.
It moves the security from the Script contained in the transaction to the age of a transaction. Before the introduction of the Segregated Witness a transaction was trusted thanks to the Script chain which connect its Input and the fact that the transaction is mined into the blockchain.
However, the script into a transaction is useful for the miner which has to verify the correctness of a transaction and for the other user which has to verify if the information contained in a mined block are correct.
Once, a transaction is accepted and mined within a certain amount of block, its security information are not used anymore. The novelty is to move the security data in a separated section in the transaction which is not considered in the block Merkle tree computation. This allows the possibility to delete them without compromising the correctness of a block Merkle tree, which means the invalidation of a block and all the consecutive blocks.

On the contrary the meaning of the transaction, data exchange using a blockchain technology is worthy to be maintained in the memory. The design of this new type of transaction called **Segwit** allows to include only the effect brought by a transaction once it is hashed. The validation script and signature can be omitted by the data without

compromise the hash chain.

Users should delete the validating data once the transaction aged enough to be claimed secured, then just the effect of it is necessary and the security is provided by its attachment into an old mined block.

However, a Full node which aims to fully validate the entire blockchain still needs to verify the chain from blocks zero, which implies to have the possibility to retrieve the security information from other nodes.
Other option would be to introduce some security breakpoint along the chain from which a node can start from there, assuming the previous chain correct and secure. However, the Bitcoin blockchain does not implemented it in the protocol.
Also, the implementation of this transaction type is quite new, since it was available in August 2017 and as far as we know, any option is available to delete the *Segwit data* from a local copy of the chain.

3.2 Attempted solution or strategy

Follows four different solutions which aims to reduce blockchain space requirement or to limit the data growth of the blockchain data.

3.2.1 UTXOs analysis

In the paper "Analysis of the Bitcoin UTXO set" [30] is introduced and describe a tool used to analyse the UTXOs dataset. This tool provides a static analysis about the UTXOs dataset and few results obtainable are shown in the paper. The paper contains analysis done with a snapshot of the Bitcoin's blockchain state at block 491,868 which corresponds to the 26th of October 2017.
The analysis provides some insights, which are:

- the *dust problem* that concern all the UTXOs which require a fee too big to make the UTXOs expendable. A fee bigger than 1/3 of the transaction's value means that the transaction's output is not-spendable alone and they can be never spent.
- the oldest UTXOs, in which whose keys can be both lost or simply not used recently. However the situation are indistinguishable and a Full-node which aims to verify, in the fastest possible way, all the incoming blocks and transactions needs to keep in memory all the UTXOs, even if some UTXOs would not be spent.

An analysis over the average time in which a UTXOs is likely to be spent would be useful for the Full-node which aims to save space, loading only the most likely expendable UTXOs and retrieving the others with a few more time. However, since the tool consider only the data in the UTXOs dataset this type of analysis cannot be done.

3.2.2 Streaming compression

During late November 2015, Peter Tschipper wrote a message in the Bitcoin community [31] with the title: *Test Results for : Data-stream Compression of Blocks and Tx's.* The author tries to compress the data stream of Blockchain to reduce the bandwidth requirement for a Bitcoin node.

He published a few experiments which aims to compress stream of blockchain data using two different compression libraries - *zlib* and *LZO*. The method used achieves a space savings up to 29%. Also, he shows that increasing the data size, the compression is better. Among other findings he shows that aggregating blocks, the compression ratio is better.

However, the maximum size of data that he tries to compress is up to 1-2 MB, also he did not consider that the content of data may influence the result. Indeed the initial part of the blockchain contains more headers data than transactions. Headers data are more structured and contain a lots of common value which are filled by default. The analysis done does not consider this problem, on the contrary our study aims to compress the entire blockchain data.

As a reply for this message, Emin Gün Sirer, on [32], why Bitcoin data seems to does not compress well. He explained why the choice of a dictionary-based compression algorithm like, *zLib* and *LZO* is not a good choice, as soon as they perform better with English text or data with patterns. A custom compression algorithm that considers the Bitcoin structure would perform better. To confirm the thesis, two comments are written by Vitalik Buterin, the designer of Ethereum, he shows how the Bitcoin structure is made and how it can be used to compress it. He also, underline how much zeros bit are wasted and how they can be replaced.

Moreover, they also introduce the idea of replacing the `Block index` field with a pointer. Another idea is to maintain a temporary structure to avoid the memory waste to broadcast a transaction and later to receive the mined block with that transaction.

No-one talks about long term storage and as far as I know, the idea was never tested. But for our research, it throws useful seeds.

3.2.3 The Mini-Blockchain Scheme

In the paper called "*The Mini-Blockchain Scheme*" [33], which gives life to a new cryptocurrency called "Cryptonite", the authors claim to provide a new way to reduce the size required by a blockchain.

Despite the introduction where the paper talks about Bitcoin, the solution suggested is not even close to be an upgrade of Bitcoin. The idea is composed by the introduction of a *Account Balance tree*, which contains all the actual amount for each non-empthy user and all the transactions is a modification of user's balance. Old transactions can be deleted after a certain amount of time, which can be decided.

However, the proposed solution is based on a completely different technique from Bitcoin's and is therefore not compatible with it. We can't talk about a reduction of the space on Bitcoin blockchain but rather about a new model of Blockchain, the security therefore tshould not be inherited from that of Bitcoin but rather requires an ad-hoc study.

3.2.4 The MimbleWimble blockchain

MimbleWimble is the name of a new blockchain model suggested in an article published in 2016 by a person with the pseudonym Tom Elvis Jedusor [34]. The blockchain model aims to solve the lack of privacy that the Bitcoin protocol suffer, in specific cases such as in the case of Light Clients.
After the publication of this white paper, thanks to a project called Grim, the implemen-

tation of the MimbleWimble idea started. In its model there is neither the concept of value nor the concept of address, so the traceability of a user or a transaction is excluded from the protocol.

The main innovation introduced is the possibility to aggregate all inputs with outputs in order to verify the validity of the system.
A Bitcoin transaction is made using several inputs and one or more output, its correctness is based on the validity of each element. In MimbleWimble the keys of all Outputs are added together using a homomorphic function, the function output is subtracted from the result of the homomorphic sum of all inputs. The state of the system is valid when the result of the mathematical operation described is null.
Every inputs and outputs does not contain a value field, but the value is represented by a key that is updated each time so that it cannot be tracked. Moreover, once an output is spent, the data can be deleted from the system state, because the homomorphic sum operation done on the inputs and outputs is done in such a way that the result does not change with the presence of a pair of related transactions. So as soon as the sum of the inputs and outputs is zero all the steps in the middle can be cancelled. The status of the blockchain must only consider the outputs not yet spent.

The benefits of this protocol are that when a user enters for the first time to be part of the ecosystem he only needs to download the final status of the system without the entire history.
Moreover, it is not possible to create unspendable money as it can be done in Bitcoin, because the MimbleWimble protocol requires that every output created must be valid otherwise the status is invalid.

The disadvantage is that most of the mathematics used in this environment has not been proven from a mathematical point of view. A system where many information are discarder by design is more difficult ti be tested and verified.
Also, additional information needs to be added as witness data in the transaction to make this protocol compatible with the Bitcoin protocol. This also makes the protocol much slower than Bitcoin's.

3.2.5 Smart contract recycling

The paper called "*Recycling Smart Contracts: Compression of the Ethereum Blockchain*" investigate a way to compress Smart contract by the reuse of frequent code pattern [35]. This study provide a saving space up to 75% in a Ethereum smart contract database.

The technique used and insight can be used also for the Bitcoin environment since the Smart contract in Ethereum have the same structure of Bitcoin Script sequences. Script language can be viewed as standard contract and pattern are present 2.2.1.
However, this technique was never tested in the Bitcoin blockchain data.

3.2.6 Inputs Reduction in Bitcoin Blocks

Michal Zima, the author of the paper named: "Inputs reduction for more space in Bitcoin Blocks" focus its attention on the space wasting by non-cryptographic use of hashes and uncompressed numbers within transaction input [36]. The main insight of the paper shows that transactions can made approximately 16% smaller, depending on their complexity.

The idea is based on Output reference in inputs, which is formed by two fields of fixed length:

Script input:	transaction hash	output index
# bytes	32 - bytes	4 - bytes

The *transaction hash field* in the input structure aims to identify uniquely the transaction which the output aims to be spent. The *Output index* identify the specific transaction output inside the transaction. The paper say that only 160 or 128 bits, instead of 256 are necessary for the first field. For the second field they suggest to use a variable length field since the majority of transaction contains just few transaction output, 99.97% of the transaction contains less than 252 outputs so only 1 byte is necessary to represent this information.

The result obtained by the application of the discussed solution is a savings of 14.5% by the reduction of bits in the *transaction hash field* and up to 16% compressing also the *transaction output index* field.
However, despite the result the solution adopted has few collision as side consequence, which is solved by the re-design of few Bitcoin protocol functions.
For these reason the solution can be implemented only as a form of soft fork of the Bitcoin blockchain.

Chapter 4

Methodology

This research aims to lower the entrance barrier, caused by the space requirement, for devices that want to participate in the Blockchain ecosystem, while also reducing the waste of resources and maintaining the security standard.
The problem of space requirements was firstly mentioned in the Bitcoin white paper, since then the space requirement is still a problem for many devices like smartphones and IoT devices.

The initial phase of the research aims to analyse the solutions already available and identify similarity among different Blockchain models to find a common strategy. However, we have soon decided to focus over the Bitcoin blockchain-based since an optimization strategy would be related to the data structure and protocol which are both specific. Ethereum blockchain uses a completely different way to store data and information within the chain so a common strategy would be useless.

Once, the protocol and the data structure were analysed from a theoretical perspective we moved to the real data analysis. Firstly, we have studied the transactions' output, considering the work done by the community, see Section 3.2.1, we have analysed both the *./chainstate* folder and the behaviour of the transactions' output. The Section 4.1 will explain the strategies adopted.

Later, looking at the entire data from a high level we tried a black-box approach to compress it using existing compression libraries. The Section 4.2 shows the experiment conducted and the strategy.
To achieve a better result, analysing the insight retrieved by the community and explained in section 3.2.2 and 3.2.6 we have studied a custom compression approach, which is explained detailed in the Section 4.3.

Moreover, since not all the users care about the entire blockchain history but just to participate in the netowork to receive or send transactions, we have analysed the existing solutions. We have identified the *Light Client*, a Bitcoin client which aims to provide a fast way to send and receive Bitcoin transaction using a poor resources devices, relying on other full Node. However, different attacks were demonstrated to compromise Light client security and privacy [28]. These nodes are not secure and also they lack privacy relying on others to retrieve the information they need.
The last part of the research done aims to quantify in terms of space requirement the methods which provide a solution for these users. The Section 4.4 will explain the existing solution, the method and experiment done to quantify the data requirement.

4.1 Transactions outputs

To figure it out a good strategy we started by analysing the behaviour of Transactions' output, which is stored in the *Chainstate folder* until their expenditure.
Bitcoin stores the unspent transaction outputs - UTXO - in the *Chainstate* folder, a separate structure. These data are used frequently by the Verification process, made by nodes, which aims to verify the validity of incoming transactions and blocks.

In the first analysis, we aim to characterize the amount of data involved in this structure, how the UTXOs are stored and if there is the possibility to optimize the used strategy.
A study named *Analysis of the Bitcoin UTXO set* analyses all the UTXOs which remain inside the *chainstate folder* for a long time and clarify the problem of UTXOs dust, see Section 3.2.1.
The Experiment 5.1.1 answers to these questions and to update the data shown in the mentioned paper. Moreover, the idea is to replicate the same experiment in the Litecoin blockchain data since it is another Bitcoin-based chain and it could be helpful into identifying pattern related to chain implementation or chain users behaviour.

However, analysing only the *Chainstate folder* we have a limited view and comprehension about UTXOs behaviour since the structure represent just a snapshot, we aim to characterize it in time.
With a second experiment we aim to verify the behaviour of transaction output along with the entire life of blockchain, we need to characterize their lifespan.
We have designed the Experiment 5.1.2 which is something never done before and it aims to retrieve this information necessary to characterize the lifespan of transactions output.

4.2 First compression attempt

After the analysis of UTXOs and transactions' output, we move to the characterization of the entire blockchain data using a black-box approach.
The official Bitcoin core node stores data using a LevelDB which produces multiple *blkxxx.dat* files.
In this phase, we aim to understand if the blockchain data can be compressed with a lossless technique.

In order to compress all the blockchain data, we have first studied the types of public libraries and algorithms which provides a lossless compression, see Section 2.3. For those who are looking for a better explanation about the compression library and technique, see Section 5.2.
Lossless compression libraries are divide according to the type of data they have to compress, lots of compression technique was developed for voice or music transmission, image or video. Therefore the choice of a library depends on the type of data needs to be compressed however, our case requires a general-purpose library since none of them is designed to compress Blockchain data.
Blockchain data are different from Video and Music content, due to security function, blockchain data are characterized by a high entropy caused by the randomness of the transactions' id and Bitcoin addresses. Dictionary-based libraries are the most promising however, we do not choose those designed to compress specific data type like English poem or Latin literature but those which can build the dictionary from scratch to address blockchain data structure, see Section 2.3.3.

We have identified 3 different library to test: the LZMA [37], LZO [38] and LZ4 [39]. All of them are dictionary-based compression library, the first one is the most promising since it is optimized for deep optimization, the others are designed to achieve the best compression and decompression speed.
In order to analyse the performance of these libraries, we have designed and implemented two different experiments which are explained in the Section 2.2.4.
The goal of this phase is to understand which is the best compression ratio achievable with traditional compression libraries.

- Initially we need to identify the factor of the data which influence the compression performance. The Experiment 5.2.1 aims to understand if ordered or un-ordered blocks or different blk size may influence the compression library.
- Secondly we aim to optimize the compression libraries parameters to obtain the best performance. Every library offers different parameters that can be changed and it is necessary to find the best combination to compress our specific data, the Experiment 5.2.2 achieves this goal.

Standard compression libraries achieve a good result however, we aim to a better one. So, we moved to a custom-designed compression algorithm using a white-box approach to Bitcoin data.

4.3 Custom compression approach

In order to design a custom approach, we need to analyse Bitcoin data more in detail. Bitcoin data are structured in blocks and transactions, which are composed of many fields, some with predefined value and the others with high entropy data, like the Bitcoin address or the transactions' id. Also, lots of fields are empty or set by default values.

To build a new strategy, we started with the characterization of the Blockchain data type. We aim to identify data distribution according to the fields' types. Then it is analysed the structure of each fields, depending on the amount of data which has that format different strategies are designed.
In the Experiment 5.3.1 we analyse the entire blockchain, reading every transaction in every block in order to count the number of bytes used by every value. The number of bytes used for flags fields, for transaction index, for block header and all the other fields needs to be measured.
As a background the structure is analysed on a high level in the Section B, the Appendix B describes the structure from a technical point of view and all the fields are described.

The experiment done over data characterization, see Section 5.3.1, shows that more than 70% of Blockchain data are Scripts' fields, see Section 5.3.1, and the article "*Recycling Smart Contracts: Compression of the Ethereum Blockchain*", which is explained in the Section 3.2.5, shows how the Ethereum smart contract presents few redundant structures that can be recycled. Also, the Bitcoin protocol allows only 5-6 type of script with a predefined structure.
In order to replicate the Ethereum experiment on Bitcoin, we aim to analyse the redundancy which characterises the Bitcoin script sequences. We analysed entirely sequences of scripts, we did not split the sequence into instructions. The amount of opcode into a

script sequence are very limited since the majority of the script sequences are occupied by hash field and public key fields which represent the majority of the length.

We design the Experiment 5.3.2 which enumerate the number of transaction, transactions' output and input. It also provides a characterization of the number of scripts which are present more than twice in the blockchain and a study over the average length of these elements.
The characterization would be very useful to try a strategy that aims to replace the script with a pointer to another structure which stores separately the script sequences and replaces them with a shorter pointer into the structure.

Literature contains also a forum topic and an article which aim to optimize field structure: *Test Results for : Data-stream Compression of Blocks and Tx's* [31] and *Inputs reduction for more space in Bitcoin Blocks* [36].
In the forum *Data-stream Compression* Vitalik buterin, the founder of Ethereum, suggested different optimization in the structure which can be used to optimize the structure. However, as far as we know no one tested this solution but it claims to save the 50% of the space.
Thanks to both the ideas we decided to make an analysis which aims to optimize fields to save space throw the elimination of padding zeros, useless fields and bad encoding.

According to the Experiment 5.3.1 the data with a higher impact over blockchain size are transactions, in particular, transactions' input with 75% and transactions' output 20%. Transactions' headers are only 2% that anyway with 2 hundred of data corresponds of 4 GB.
Therefore we focus on the optimization of the following fields: transactions header, transactions input and transactions output.

4.3.1 Transaction header fields

We have optimized the transaction header in the following way:

- version: the Bitcoin protocol implements only two versions of transactions. Currently the version field is composed of 4 bytes, which means that it can distinguish up to 4 million different transaction versions.. The actual field is 4 bytes.
- flag: this flag if present represents the presence of *segwit data*, it is a flag so only 1 bit is necessary, the actual field is optional and uses if present 2 bytes.
- lock_time: this field is most of the time not used [40] and replaced with a default value.

In our solution we have introduced a new 1-byte field that replaces the three fields: version, flag and *lock_time*. An additional 4-byte field is inserted to represent the value of the *lock_time* field, if the latter is different from the default value.
The first field is broken down as follows: 2 bits [0:1] are reserved to indicate the transaction's version, 1 bit [2] replaces the *segwit* flag field, and an additional bit [3] is used to indicate whether the *lock_time* field contains a default value or not.
Where the *lock_time* field contains a value other than the default value, the value is inserted in the 4 bytes following the initial field.
In this solution, 4 bits are left for future implementations. Every Bitcoin fields has the size of n bytes so the minimum size of a field is 1 byte.

4.3.2 Transaction input fields

Regarding the transaction input data we have designed the following solution:

- previous_output: this field has 32 bytes that represent the tx hash + 4 bytes for the output spent, the hash can be substituted with the block height (fixed 4 bytes) plus the transaction position into the list for this reason also, the output number field can be replaced with the *varint field* instead of a fixed 4 bytes field.
- sequence: this field is represented using 4 bytes but it is most of the time set by default.

The solution that we have implemented is to add a flag at the beginning of each input where: flag[0] indicates where the previous output is substitute with the block and transaction position or not, bit[1] indicates where the script is replaced with a pointer or not, bit[2] indicates if the sequence is default.

4.3.3 Transaction output fields

Regarding the output data we have designed the following solution:

- value: this field represent the amount of Satoshi and it has a fixed size which is 8 bytes. The structure of this field is substituted with a variable integer structure which allows more flexibility in terms of size required.
- script_length and script: these fields contain the script and its length respectively. In our solution the script is replaced with a pointer to the field of a data structure in which it is contained the script sequence where the latter is frequent in the blockchain. If the script is replaced with a pointer, the length field will represent the length of the pointer.

In addition, another flag must be added at the beginning of the transaction's output structure. As we said before, the minimum size of a field in the Bitcoin protocol is one byte. The flag we are going to add at the beginning is a flag of 1 byte and represents if the script is replaced or not.

The previous analyses aim to find the best way to compress the entire blockchain data. The solution can be very useful for Archival node and those users who want to maintain a copy of all the data in their local storage.
However, other users may be interested in participating in the ecosystem without the requirement to download the entire chain and also without the cons to lose part of the security.

4.4 Full node optimization

Currently, to participate in the mining process you need a Full node that must download the entire blockchain.
Instead, you can send and receive transactions using a Light client that provides a lean way to participate in the blockchain ecosystem. Unfortunately, the Light client has to rely on other nodes, losing his privacy and delegating security to third parties.

There is also another way to join the ecosystem safely without the necessity to download the entire blockchain. This solution has never been implemented or studied. Our study aims to understand how much memory is required to each user which aims to adopt this method. The description of the solution follows.

The blockchain protocol requires you to verify the validity of each new block and each new transaction. A block is valid when all its fields are well-formatted, the block header includes the hash of the previous block and all transactions contained are valid. To verify if the hash of the previous block is correct, the user must keep the header of the previous block. In addition, a transaction is valid when all values used as input are correct. To check the transaction input you need to check if the sender has the right to spend that UTXO. The structure of UTXOs does not implement a mechanism to verify if it has been manipulated by a malignant entity, so the user must verify the validity of each UTXO by checking if the relevant transaction has been extracted in a valid block. To verify if a transaction belongs to a specific block it is necessary to recalculate the root hash of the Merkle branch and verify if it is equal to the hash written in the block hash. The standard method requires you to store a copy of all transactions, with this method the user can simply store the path of the brother for each UTXOs, so you can easily check if a UTXOs belongs to a block or not.

A client can stay up to date with the blockchain by receiving and verifying each new block and transaction. He just needs to keep a local copy of the header of all blocks, the *./chainstate* folder and additional information about the *sibling path*.
The amount of space required by this solution has never been studied and our experiment aims to provide an accurate evaluation. In the Experiment 5.4 we aim to count the space needed to store additional information about the siblings' path.

Chapter 5

Evaluation

This chapter contains an explanation of the evaluation that we have conducted through numerous experiments, all of which are explained below. The analysis mainly cover three topics, an analysis of the UTXOs distribution, the optimization of available compression library and a customized way to optimize the space based on specific role's needs.

All the experiments and analyses are made using the same *Machine setup*, which is a machine with an Intel(R) Xeon(R) CPU E3-1240 v5 @ 3.50GHz and 32GB DDR4 RAM. For the amount of data used the persistent storage is relevant, therefore a Solid-state disk with 3 TB of capacity is used.
Concerning the software, all the analyses are done using libraries and custom scripts written in Python 2.7 or Python 3.7, specific requirement are listed in every experiment. The results come from analyses made over the Blockchain's data. Functionality and data provided by external parser or external blockchain information services with their own API, do not provide enough information for our need.

5.1 Transactions' output analysis

In this section we have analysed two aspects of the transactions' output: the UTXOs distribution along the Blockchain and the average time within a transactions output is likely to be spend.

In this analysis both the blockchain of Bitcoin and Litecoin are analysed. The Bitcoin data is related to the official Bitcoin's blockchain mined until 10-10-2018 which contains 545 K blocks. Data are divided in two folder, the *"chainstate"* folder and blocks folder, which require respectively 2.8 GB and 199 GB.
The Litecoin blockchain as the Bitcoin date back to 10-10-2018 however the entire Blockchain file size is 16 GB.

5.1.1 UTXOs Distribution

Description

This experiment is an analysis of the Bitcoin and Litecoin UTXOs. The goal is to count how many UTXOs are left along the blockchain and how are they spread along the block. Few blocks contains a lot of UTXOs or UTXOs are spread in different blocks, the analysis aims to answer to this question.

As it is specified in the Section 2.2.4 all the UTXOs are stored in a database structure that is client-software dependent. The official Bitcoin node stores by default the UTXOs inside a folder called *"chainstate"* using a *levelDB* [15]. The entire experiment analyse this structure for the both of Blockchains.

Setup

- download the `chainstate folders` for both Bitcoin and Litecoin;
- install `python2.7` which is necessary to run the Python code developed;
- retrieve the `bitcoin_tools` that is an external python library able to parse part of the blockchain data easily. The library is available here: `https://github.com/sr-gi/bitcoin_tools`. Also, after the download, the library needs to be configured following the instructions contained in the repository folder. It simply requires to setup the `folder path` of our *chain folder* and *chainstate folder*. Also the type of Blockchain needs to be specified to, it can be either Bitcoin and Litecoin.

Strategy

The goal of the experiment is to obtain the exact number of the UTXOs in the blockchain snapshot we have and also to characterize the UTXOs distribution along the blockchain's block.

The *chainstate folder* contains a levelDB that contains the entire list of UTXOs together with other chain state data, like the number of blocks. Every UTXOs is encoded and to read it is necessary a parser. The `bitcoin_tools` contains a method that given the *chainstate path*, it parses all the levelDB, looking for UTXOs and it produces as outcome a list of UTXOs decoded and expressed using the JSON file format.
Once a UTXOs is decoded it contains different information, however we only look for the block age, which is the number of the block where the transactions related to the UTXOs was mined.
We have developed a python script, which parse the JSON file produced by the `bitcoin_tools` and produces a chart with the number of UTXOs remained in each block.

Follow the pseudocode of the python script:

Algorithm 1 Parse the chainstate database

```
1: utxos = parsed chainstate with the bitcoin_tools library
2: // create a list where each value represents the number of utxos in the specific block
3: block_index = [0, 0, 0, ..., 0]
4: for all UTXO ∈ utxos do
5:     block_index[UTXO.height] += 1
6: end for
7: # plot chart
8: create new chart
9: for i = 0; i++; i < chain_len do
10:    chart.insert(x=i, y=block_index[i])
11: end for
```

Legend:

`chain_len` is the number of blocks in the blockchain, 545 000 for Bitcoin and 1.507 M for Litecoin.

Result

The execution time for this experiment is 10 minutes for each run. The result are plotted in the chart 5.1 for Bitcoin and in the chart 5.2 for Litecoin. In those charts each blue dot represents a specific block; the x-coordinate represent the block's index and the y-coordinate represents the number of UTXOs contained in that block.

Bitcoin

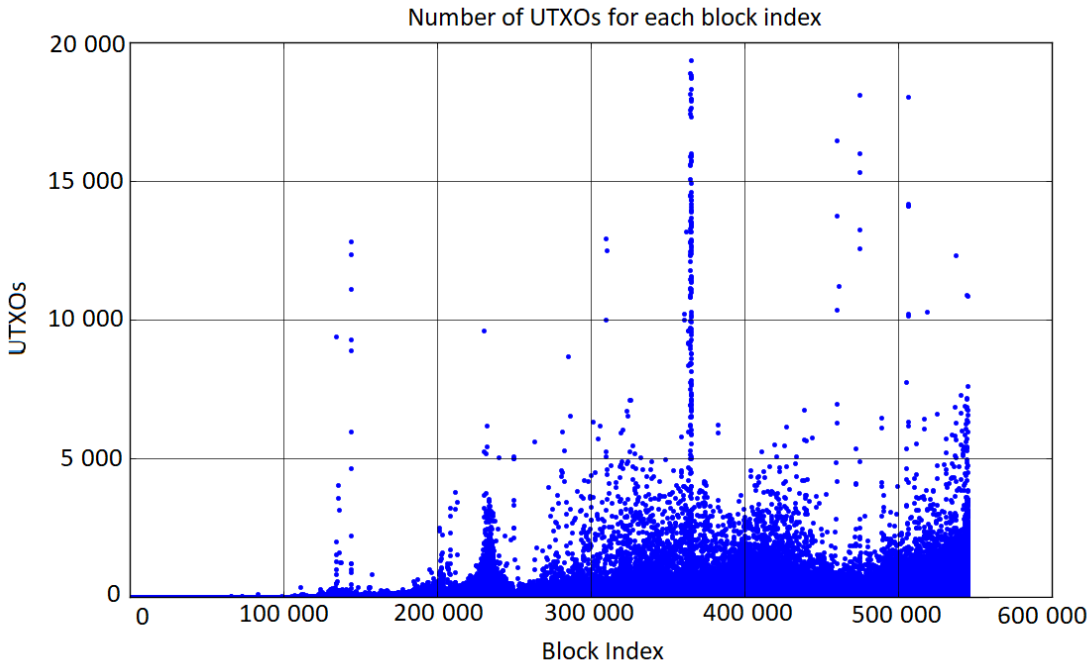

Figure 5.1: The chart shows the age of all the Bitcoin's UTXOs, until block's index: 545 000. The blue point at block index 225 000 shows that the Bitcoin block at that index still contains about 9 500 unspent transactions' output.

The total number of UTXOs are 50 349 323 and they are not concentrated only whithin the recent blocks but they are spread along the entire blockchain. Only the 19.2% of all the blocks contains only spent transactions' output, the other 80.8% contains UTXOs. What we can see from the chart is that after the first 100 K blocks, the majority of the blocks contains UTXOs. However, some specific range of blocks contains a huge amount of UTXOs, compared with the average.

Litecoin

The same analysis is done for the **Litecode** blockchain and the result is shown in the Figure 5.2. From the analysis, we know that there are 22 364 371 UTXOs distributed within 520 106 blocks that represent the 34.49% of the entire amount of blocks.

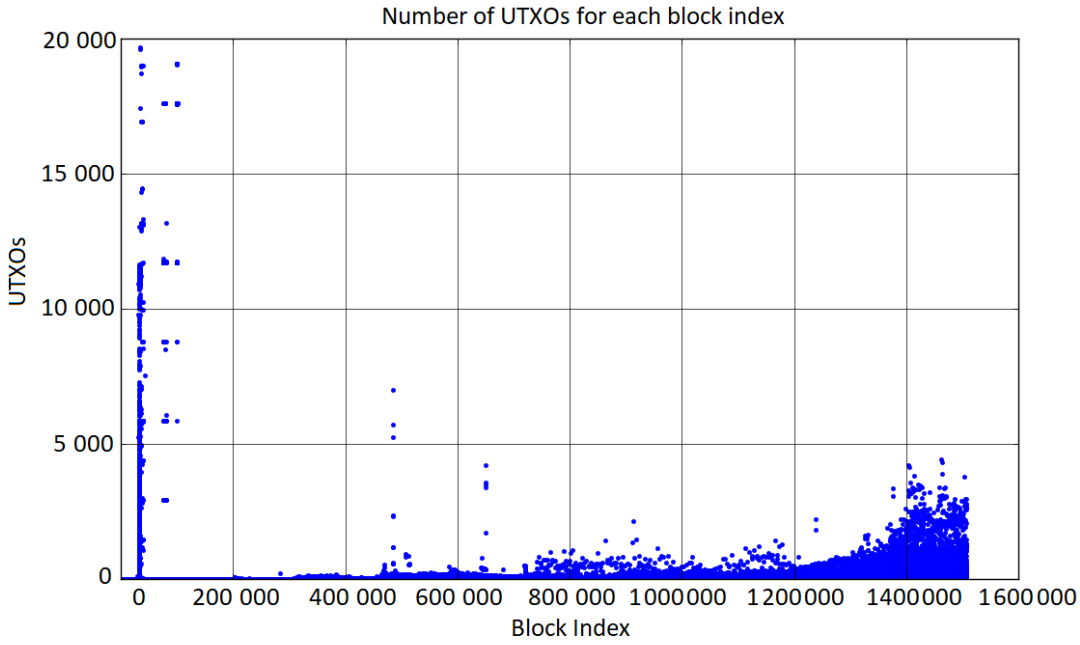

Figure 5.2: The chart shows the age of all the Litecoin's UTXOs, until block's index: 1 507 816. The blue point at block index 480 000 shows that the Litecoin block at that index still contains about 7000 unspent transactions' output.

For both the charts, there cannot be two dot aligned vertically and dot structures which seems to be aligned are just dots that represent near blocks. Due to the high horizontal density these dots seems aligned, however they are not.

5.1.2 Spent Transactions' output life span

Description

The goal of this experiment is to characterize the life span of the transactions' output, which is the time that passes between the transaction creation and the transaction's output expenditure. The experiment aims to analyse both UTXOs and transactions' output already spent. The time is expressed using the number of blocks.
The value changes also along the blockchain with the increasing of transactions' output considered in the analysis. Also, we are looking for the blocks number which guarantees that a transaction is likely to be spend with the probability of 90% and 95%.
Certainly the value will be updated along with the advance of the analysis, which starts from the block zero to the end at the last block, so the results needs to show also this behavior.

Then, we plot in another chart the lifespan behavior of all the transactions' output using a Cumulative Distribution Function.

Setup

- `./blocks`, the entire block's folder is necessary since we need to parse all the data inside the blockchain to retrieve the age of each spent and unspent transactions' outputs;
- `Python v3.7` is necessary to run the python script developed and the library we have used;
- download and install the `bitcoin_chain_parser` [41], this is a python3 blockchain parser necessary in order to parse the blockchain's data in the right sequence. The parser is retrievable at the following link: `https://github.com/alecalve/python-bitcoin-blockchain-parser`.

Strategy

For this experiment we need to characterize the age behavior of the transactions' outputs inside the entire Bitcoin blockchain.

We can define the life span of a specific transaction's output as the difference:

$$lifespan = bh_j - bh_i$$

where bh_j is the block height where the tx_output is spent and the bh_i is the block height where its related transaction was created.
Now, the main difficult here is to retrieve the age of a transaction's which creates a specific transaction's output from the transaction which spends that transaction's output.
In other words there is not reference to the life span of the transaction's output at its expenditure time. Therefore the life span of a specific transaction's output is written neither in the blockchain data nor in the *chainstate folder* and it needs to be retrieved indirectly.

The theoretical approach would suggest to parse the entire blockchain from the beginning. Using python we need to create a dictionary with elements formed in this way: $\{key =$ `tx_id`, $value =$ [`block_heigth_creation`, $[x_1, ..., x_k]$] $\}$ where x_i is the block_height spent of the i-th output and k is the number of outputs of that transaction. Then every time the script encounters a new transaction it has to update the age of each transaction's outputs used.

However, this way is not feasible from a memory requirement point of view since it requires to maintain billions of key-value pairs in a structure which has to provide a fast random access to its stored data. Moreover, data are growing up during the parsing analysis.

The solution designed aimed to reduce the information that needs to be maintained simultaneously in the machine memory, with the cost of few more calculus and the maintenance of two separated structures.
One structure maintains the result we are looking for, transaction's outputs and the time which is passed between their creation and their expenditure. This data structure is updated on real-time during the analysis, since once a transaction's output is spend, it is inserted in the first structure and it can be deleted from the second structure which contains only the not yet spend transactions.
The structure which collect the result is named A and has the following structure: $A = [x_0, x_1, ..., x_h, ..., x_n]$ where n is the maximum age that a tx_output can has, which is

equal to the maximum length of the chain. Then each value at position x_h represent the number of tx_outputs spend within h blocks from its creation time.

The second structure named T aims to maintain the temporary information, which is the time of a transaction and the number of its unspent outputs, everything keyed with the transaction's id. The structure is made in the following way: $T = \{tx\_id_1 : [h, o]\ , tx\_id_2 : [h, o]\ , ..., tx\_id_n : [h, o]\}$

- `tx_id` is the transaction's index;
- `h` is the height of the transaction creation;
- `o` is the number of outputs that the transaction has.

The analysis parses the entire blockchain, sequentially and analysing all the transactions contained in each block. Every time that a new transaction is created the script adds an entry related to this transaction in T with its height creation block and number of output. Then the script calculates the age of every output and uploads the information in A. Moreover in order to limit the amount of memory necessary, every time a transaction's output is spent, the counter `o` of the transaction is decreased and once it become zero the entire transaction entry is deleted from T, it means all its output were spent. The algorithm used is shown in the pseudocode 2.

Algorithm 2 Parse the transactions outputs life span

```
1: import Blockchain from python_blockchain_parser
2: blockchain = new Blockchain($blocks_path)
3: T = {}
4: A = [0_1, ..., 0_n]
5: sample_counter = 0
6: for all block ∈ blockchain do
7:    h = block.height
8:    for all transaction ∈ block do
9:       T.addtransaction.id, [h, transaction.number_of_outputs
10:      for all tx_in ∈ tx.inputs do
11:         age = h - T[transaction.id].height
12:         A[age] += 1
13:         T[transaction.id].outputs_number -= 1
14:      end for
15:   end for
16:   if h == sample_counter + 10 000 then
17:      sample_counter = h
18:      save(A) # save the counter in the memory
19:   end if
20: end for
```

Legend:

T dictionary with the format: tx.id, [block height, number of tx.outputs]

A array where the position represent the number of blocks and value represent transaciton's output spent after index blocks.

Then in order to see the trend of the parameter calculated, the script needs to provide a time meaning to the analysis. Therefore the script takes a sample of the data structure A every G blocks analysed. Trivially the smaller G is the more precise the analysis is. At the end of the analysis different samples is saved and each $sample_i$ is an array where each element x_h is the number of tx_output spent after h blocks from the beginning of the chain until the block: $G * (i + 1)$.
The plot we want to obtain aims to show the trend of 3 different data:

- the **avg life span** line shows the average amount of block after that a transaction is going to be spent. This value is a cumulative representation and every dot represents the value estimated from the beginning of the chain until the block aged like the x-coordinate of the dot.
- the **95% and 90% CDF** lines represent the number of blocks after which a transaction is likely to be spent with a probability of respectively 95% and 90%.

Algorithm 3 shows how to produce the plot 5.4.

Algorithm 3 90% and 95% confidence interval

```
last_block = 540 000
for i = 0; i++; i < last_block do
  x_i = i
  #transactions = sum(sample_i)
  weighted_sum_avg = 0
  for j = 0; j++; j < i do
    weighted_sum_avg = weighted_sum_avg + sample_i[j] * j
  end for
  y_avg_i = weighted_sum_avg / #transactions
  weighted_sum_90 = 0, partial_counter = 0
  for j = 0; j++; j < i do
    weighted_sum_90 = weighted_sum_90 + sample_i[j] * j
    if partial_counter ≥ 90% of #transactions then
      y_90_i = weighted_sum_90 / partial_counter
    else
      partial_counter += sample_i[j]
    end if
  end for
  weighted_sum_95 = 0, partial_counter = 0
  for j = 0; j++; j < i do
    weighted_sum_95 = weighted_sum_95 + sample_i[j] * j
    if partial counter ≥ 95% of #transactions then
      y_95_i = weighted_sum_95 / partial_counter
    else
      partial_counter += sample_i[j]
    end if
  end for
  plot(X, Y)
end for
```

In order to provide another useful type of data another script is designed to produce the cumulative distribution function regarding the transactions probability within a certain amount of blocks. The pseudocode is available in the algorithm 4.

Algorithm 4 Cumulative distribution function of the transactions' output along the entire Bitcoin blockchain.

```
retrieve the result structure A produced with the algorithm 2
max_block_age = 540 000 #age of the BTC blockchain analysed
#transactions_number = sum(A) # somma tutti gli elementi nell'array A
partial_sum = 0
for i = 0; i++; i < max_block_age do
    x_i = i
    partial_sum = partial_sum + A[i]
    y_i = (100 * partial_sum)/#transactions_number
end for
plot(X, Y)
```

Result

The first figure 5.3 represent the life span analysis over the entire Bitcoin blockchain. The plot is created using the algorithm 2 for the preliminary result and the algorithm 3 for the last analysis and plot.
The second chart 5.4 represents the cumulative distribution function of the transaction's output life span. The chart is generated using both the algorithms 2 and 4.
Another important consideration is that in both these experiments UTXOs are not considered since it is difficult to predict their spent age expectation, which is by the way the goal of this experiment.

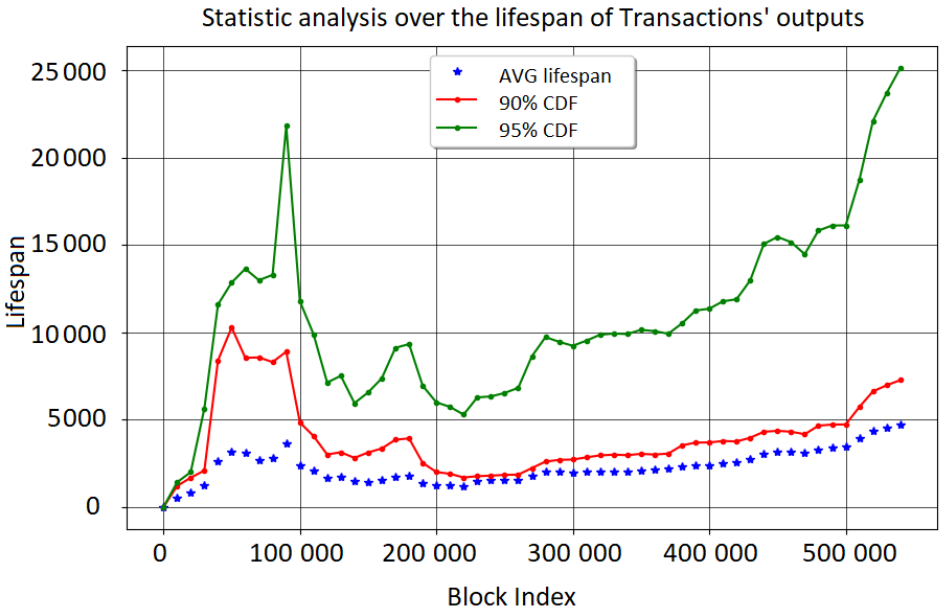

Figure 5.3: tx_output Life span analysis over transactions' outputs. The blue point at block index 100000 shows that transaction outputs, created from block 0 to block at that height, are on average, spent after about 2500 blocks. The red point at block index 100000 shows that the 90% of all the transaction outputs, created from block 0 to block at index 100 000, are on average, spent after 5000 blocks. The green points have the same meaning of the red points, however they represent 95% of all the transaction outputs.

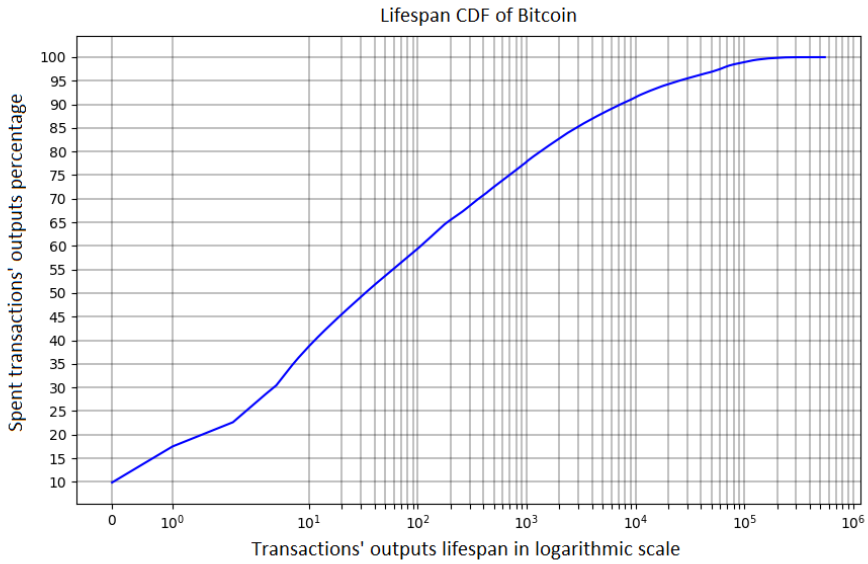

Figure 5.4: CDF of Bitcoin transactions' outputs. The index point 100 shows that 60% of all transactions created are spent within 100 blocks from their creation. Also, the point at index zero shows that 10% of the transactions' output are created and spent within the same block.

5.2 Traditional Compression

This sections aims to identify the best way to compress blockchain data, which is viewed as a black-box, using standard compression library.
In the initial part different compression libraries, available in the network are compared together to identify which is the best one regarding blockchain data.
Secondly, different compression libraries parameters are tried to push the upper limit of the available technology.

5.2.1 Compression library Comparison

Description

This experiment aims to identify which are the best compression libraries technology to compress blockchain data. The blockchain data are viewed as a block-box, however the structure in which data are stored can be customized for the purpose.
The reason why we need to modify the storage structure is due its structure and the way in which blockchain's blocks are stored. An explanation of the structure is available in the Section 2.2.4.

As described in the Section 4.2 section, dictionary based compression library have the right characteristics to perform well in the blockchain data compression. Here, the test aims to compare the performance of three different compression libraries, which are:

- *LZMA* is designed to achieve the best savings ratio result.
- *LZ4*, *LZO* using different compression strategies they are designed to be fast during both compression and decompression phases.

The dictionary based compression library needs to make the preliminary analysis on the data, with the most widely view available over the dataset, see Section 2.3.3.
However, due to the blockchain size and the maximum size of the input accepted by the library available, the test needs to chunk the blockchain data in different pieces. The input file size in the library cannot be more than 2 GB compressed in a single round. For this reason and in order to provide a way which is able to compress the entire blockchain data, we need to understand also the variation produced in the libraries' performance giving different chunk of data.

So, the experiment will test the compression performance of different libraries, over different formatted chunk of blockchain files. Blockchain's blocks are usually stored in blk files non ordered, see Section 2.2.4.
Therefore, it is tested the variation of performance using ordered and non ordered block.
Also, the same test is done in three different blockchain type: Bitcoin, Litecoin and Dogecoin, which are both are fork of the same project.

Setup

Firstly it is necessary to import and configure the blk_creator python script created before and explained in the Appendix A. The strategy section will explain how it is used to produced the chunk of file used for the test.

The entire blockchain data is also necessary, in particular Bitcoin, Litecoin and Dogecoin in order to repeat the same test for all of them.

It is necessary to download and install the compression library written in Python. The follow instruction will achieve it:

LZMA *user$ pip install backports.lzma* [37]

LZO *user$ pip install python-lzo* [38]

LZ4 *user$ pip install lz4* [39]

Strategy

For this experiment is necessary to produce different tester file and we have produced them maintaining the Blockchain Bitcoin structure since it is more efficient and easier to be integrated in future implementations. The structure is the same of blk00XXX.dat files used to store the entire blockchain, as explained in the Section 2.2.4. The size of the original files are 128 MB and block are stored inside un-orderly.

In the first phase it is necessary to create the sample files that are going to be used as input for the compression tests. Therefore, a Python script able to build the blk file using blokcchain data was created and its implementation is explained in the appendix: A.
Basic inputs and outputs of the code is respectively a bunch of blocks from a specific blockchain data, which can be Bitcoin, Dogecoin or Litecoin. It does not matter if blocks are ordered or not and the outcome would be a blk file of a requested size, that would be the sum of the size of all the single blocks provided.

In order to test all the possibilities discussed, the following criteria are used to create the sample files. Sample size cannot be greater than 2.5 GB since the used python libraries do not allow input files bigger than that. The following size are chosen: [100 MB, 200 MB, 500 MB, 1.0 GB, 1.5 GB, 2.0 GB].
Also, only for the Bitcoin samples we have created different samples to distinguish ordered blocks from the un-ordered. For Dogecoin and Litecoin it would not be possible since their blockchain is not so big, sample of 2 GB are greater enough to represent 20 to 25% of their entire blockchain size.
Another important things we have considered is that the distribution of the blocks we have inserted inside the blk file should represent the real Bitcoin blocks entropy in order to avoid a not reliably result the blk_creator uses randomness to pik the blocks that needs be inserted. Also, it is not allowed the repetition of same blocks, once a block is inserted it cannot be inserted anymore otherwise the compression ratio will benefit from this and it is not a realistic.
Every experiment are also repeated multiple times for each configuration, 10 for Bitcoin data and 5 times for Litecoin and Dogecoin data. Different file sample are also produced accordingly.

The parameter we have measured in the experiment are the following: the compression ratio and the time necessary to do it.

The algorithm 5 contains the pseudocode used for the experiment.

Algorithm 5 Compression test

```
import blk_manager as blk_manager
size = [100 MB, 200 MB, 500 MB, 1 GB, 1.5 GB, 2 GB]
c_lib = [LZMA, LZ4, LZO]
for all s ∈ size do
    # Ordered blocks
    files [ ] = blk_manager.get_orderedFiles(size=s, sample=10)
    for all lib ∈ c_lib do
        lib_avg_time = 0, lib_avg_space = 0;
        for all f ∈ files do
            t_i = time.now
            file_compressed = lib.compress(f)
            avg_time += time.now - t_i
            avg_space += 100 * (size(file_compressed) - size(f)) / size(f)
        end for
        print(avg_time / 10, avg_space_saved / 10)
    end for
    # Unorder blocks
    files [ ] = blk_manager.get_unorderedFiles(size=s, sample=10)
    for all lib ∈ c_lib do
        lib_avg_time = 0, lib_avg_space = 0;
        for all f ∈ files do
            t_i = time.now
            file_compressed = lib.compress(f)
            avg_time += time.now - t_i
            avg_space += 100 * (size(file_compressed) - size(f)) / size(f)
        end for
        print(avg_time / 10, avg_space_saved / 10)
    end for
end for
```

Result

A single run of this experiment takes 1 day, this amount of time is due to the *LZMA* library which is very slow and which takes 30 minutes to compress a single sample of 2 GB. The following three figures represent the experiment done for the three different blockchains' data.
The first plot, Figure 5.5, shows the result of both ordered and un-ordered blk files.
The following two charts represent the achievements for Dogecoin in the Figure 5.6 and Litecoin in the Figure 5.7.

The first chart 5.5 shows that for Bitcoin the ordered blocks are compressed better then un-ordered blocks, independently from the library chosen. The best result achieved for the un-ordered blocks is reached with the *LZMA* library which is able to achieve a saving percentage of 22.5%, clearly it is worst than the 30% savings achieved with the same library but using the ordered blocks.

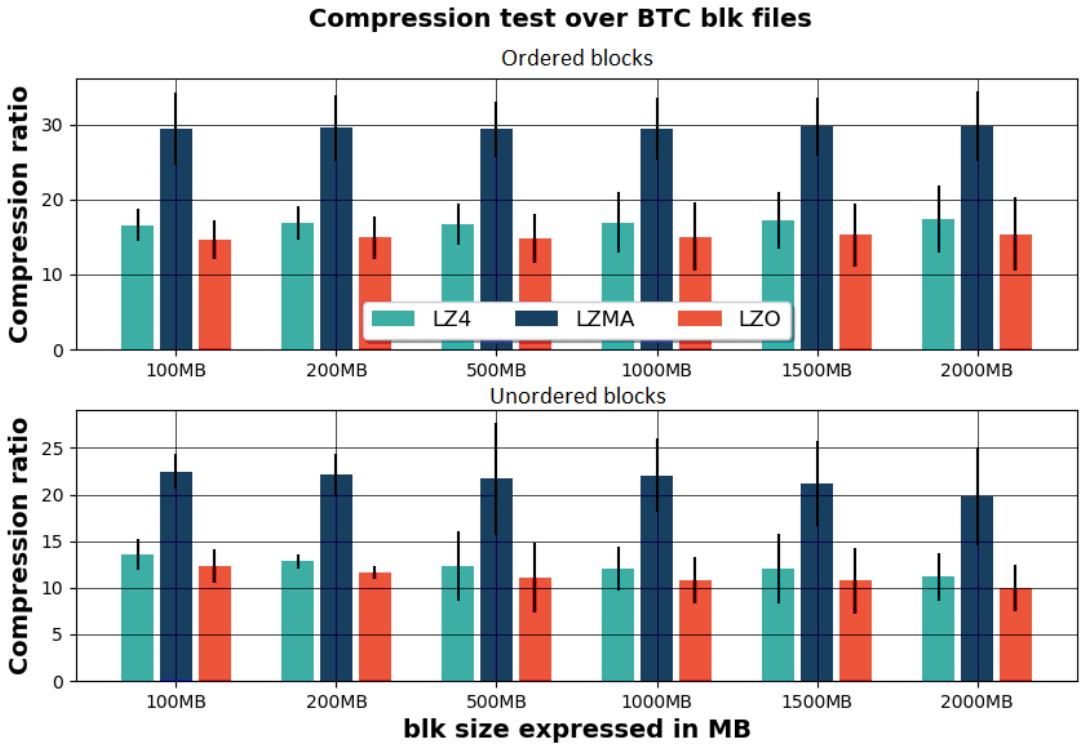

Figure 5.5: Compression test over Bitcoin blk files. In the upper part of the chart, the blue bar at index 200 MB shows that with the LZMA compression library the experiment achieve a compression ration of about 29.5%, done over file with size 200 MB made by a list of ordered blocks. On the right side of that bar you can see the compression ratio achieved by the LZO compression library, and also the performance achieved by the LZ4 library on the left side.

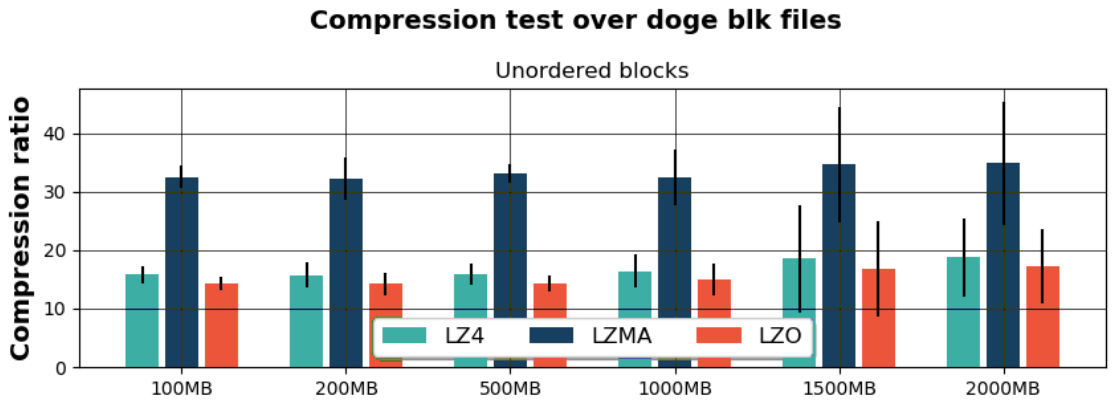

Figure 5.6: Compression test over Dogecoin blk files.

All the charts also show that the *LZMA* library works clearly better, with a factor of two, over the compression ration.

Also, the performance obtained in the compression of Btc ordered blocks is not clearly distinguishable in the difference of file size. The 100 MB size seems to have 1% point

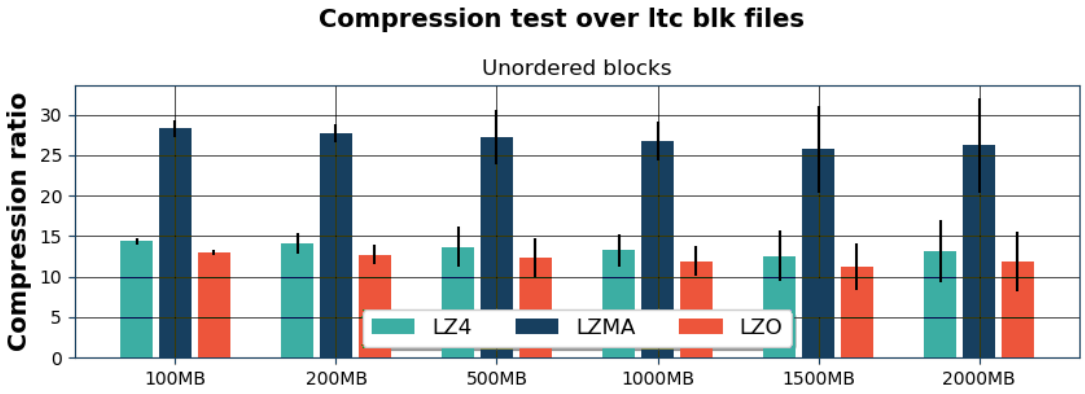

Figure 5.7: Compression test over Litecoin blk files.

less then 2 GB samples, however with the increasing of the file size it is increased also the variance.

5.2.2 Optimizing Compression parameter

Description

The compression libraries we have used in the previous experiment 5.2.1 have different configuration parameters, this experiment aims to characterize the compression performance varying the parameters.

For this experiment we have evaluated the parameters only with the Bitcoin data sample. Also, since *LZMA* library has a lot of parameter, trying all the combination would takes weeks due to the amount of time required by a single file compression with this library. We have chosen to optimize the compression only for the most promising sample, which are the ordered Bitcoin blocks of 2 GB.
The parameters we have chosen to be evaluated for every library during the test, are listed below:

- *LZ4* has two parameters: dictionary size and compression level:
  - compression level $\in$ [0, 5, 9, 16;
  - dictionary size $\in$ [64 KB, 1 MB, 4 MB].
- *LZO* has only the level parameter so we used the values $\in$ [1, 5, 9].
- *LZMA* we have decided to modify the compression level, the dictionary size and the match finder. Following specific pattern.
  - compression level $\in$ [1, 5, 9], dictionary size 64 KB and match finder = MF_BT2
  - compression level $\in$ [1, 5, 9], dictionary size 16 KB and match finder = MF_BT3
  - compression level extreme $\in$ [1, 5, 9], dictionary size 1.5 GB and match finder = MF_BT4

Setup

Ten sample Bitcoin files are necessary, it can be token from free from the previous experiment or they can be created using the blk_creator, which is a software explained in the Appendix A. The samples needs to contain 2 GB of ordered Bitcoin blocks.

Then as the previous experiment the following instruction need to be followed in order to install the required compression library:

LZMA *user$ pip install backports.lzma* [37]

LZO *user$ pip install python-lzo* [38]

LZ4 *user$ pip install lz4* [39]

Strategy

The test aims to test different library configuration parameters.
Analysis mainly consist of setting up every different possible configuration and for each one it is necessary to repeated it 10 times changing the input file. The file needs to have the same structure characteristics, but different content in order to avoid performance behavior due to sample size or content instead of specific parameters itself.

We have build a dictionary with all the configuration parameters that needs to be tested. Then for each configuration the compression analysis is done over ten different file and the average of the time and compression time results is done.
The algorithm 6 contains the pseudocode of the experiment.

Algorithm 6 Compression customization

```
1: ConfigList = [LZ4 lve:9 extreme, LZMA lve:3 extreme dictionarySize:1Mb, ...]
2: Tester_Files = [blk01.dat, blk02.dat, ..., blk09.dat]
3: for all conf ∈ ConfigList do
4:    time = 0
5:    space_saved = 0
6:    for all file ∈ Tester_Files do
7:       t_i = time.now
8:       file_compressed = conf.library.compress(file, conf, conf.parameters)
9:       time += time.now - t_i
10:      space_saved += [100 * (size(file_compressed) - size(file))] / size(file)
11:   end for
12:   avg_time = time / number of files
13:   avg_space_saved = space_saved / number of files
14:   print(configuration: conf, time: avg_time, space_saved: avg_space_saved)
15: end for
```

Result

The charts in the Figure 5.8 shows the compression ration achieved by different configuration with red bars in the figure above and its related time with the blue bars in the figure below.
Also the chart is divided vertically in two sections, the Figure on the right below has a

different time axis values. The reason is that all the analyses done with parameters on the left side require not more than 400 seconds to be done, instead the other analyses require 100 times more time, so using the same scale it would completely hide all the initial tests done.
However, the compression ration achieved between the configuration on the right side is comparable with the analyses displayed in the left side figure.

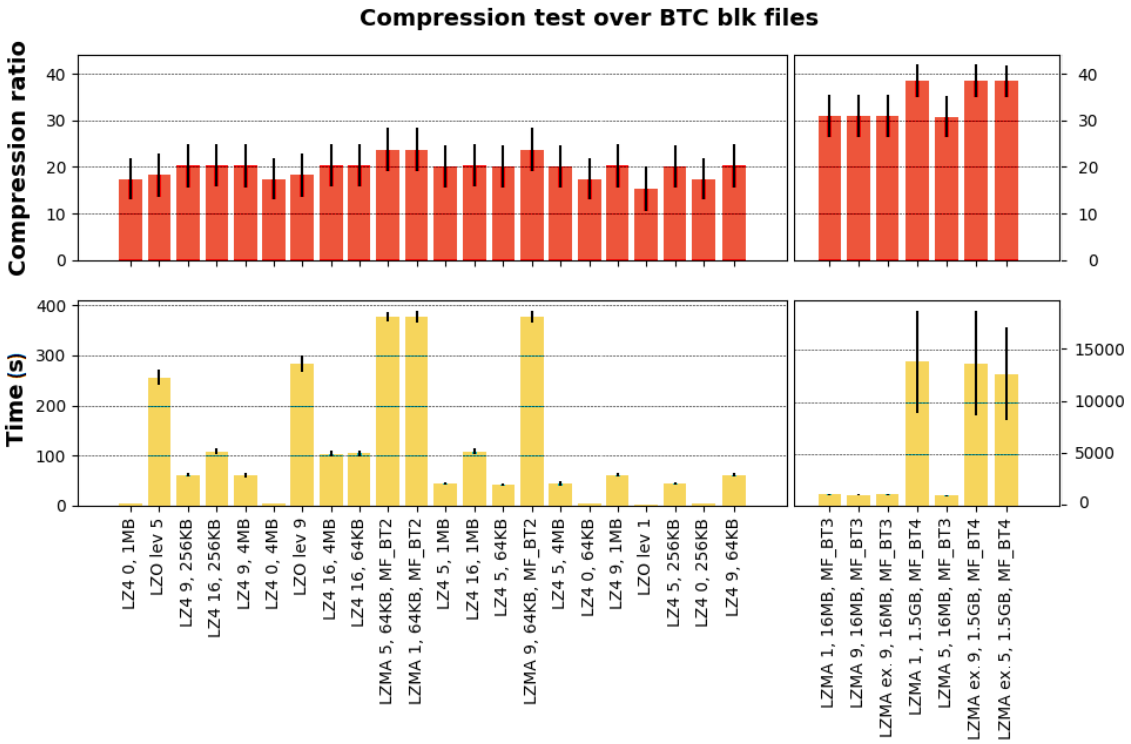

Figure 5.8: Parameters optimization of traditional compression libraries. The fourth red bar shows that the LZ4 compression library, with a level compression of 16 and a dictionary size of 256 KB, reaches 20% as compression ratio, taking about 100 seconds, which is displayed in the blue bar below.

The main finding is that the best result in terms of compression ratio is achieved by the *LZMA* library with the following setup: the extreme parameter, a dictionary of 1.5 GB and the MF_BT4 match finder, reaching a 38% of savings space percentage.
However, the best compression requires the most demanding method in term of time consumption, which require more then 4 hours in order to compress a single file.

5.3 Data analysis for efficient Compression

Until now with the previous experiment the legacy compression library was studied. A different approach in this section is applied, the blockchain structure is not viewed as a black-box and the structure is analysed to retrieve a possible optimization.

In the following experiment it is made a characterization of the space used in the blockchain dividing different value type. Also, a specific analysis is done to estimate

the entropy of the script. Both these analysis will lead us to the motivation for the last experiment which is able to combine the findings to achieve a better compression result.

5.3.1 Blockchain Data Distribution

Description

Blockchain's data are strongly structured, the data are stored within customized fields with value that are limited within specific range. The goal of this experiment is characterizing the amount of data saved in the blockchain divided by field's type.
Also, we want to study the values distribution of specific fields to estimate if a different field's structure can benefits the amount of memory used.
As it is explained in the Section 2.2.3 Bitcoin data structure can be divided in the following section:

- *block header*: `magic_number, block-size, block's header, number of transactions`;
- *block content* which is a list of transactions, where every transaction has the following structure:
  - *transaction's header*: `version number, witness flag, input number, output number, witness data and lock time.`
  - *transaction's body*
    - `transaction's input`
    - `Input script`
    - `transaction's output`
    - `Output script`

Setup

- Download the entire blockchain of the Blockchain you want to analyze. In this case the Bitcoin blockchain.
- The complete blockchain is necessary since the parser will iterate sequentially over all the data measuring the size of each field we aims to analyse.
  The data can be retrieved in the data folder of a Bitcoin client with the entire blockchain downloaded.
- Download the Alecalve python blockchain parser, used also in the experiments 5.2.1 and 5.2.2.
- Modify the library accordingly to the Appendix C, since it is necessary to insert some additional method to retrieve some some specific data which is otherwise omitted in the standard library.

Strategy

Since we want to count the amount of space used by the data of the blockchain divided by the fields type, it is necessary to parse the entire blockchain. For each block encountered we need to retrieve the block header, and the size of each field in its body section.

The script maintains different counters which are going to be updated during the scanning process of the blockchain.
A Python script is designed, it will create and update all the counters mentioned before and to parse the entire blockchain it uses the modified version of the Alecalve parser, see Appendix C.
The pseudocode of this script is available in the algorithm 7.

Algorithm 7 Space field counter

```
1: tx_size, tx_input, tx_output = 0, 0, 0
2: block_header, transaction_header = 0, 0
3: script_in, script_out = 0, 0
4: for all block ∈ blockchain do
5:   block_header += block.header() #return the size of the block's header in bytes
6:   for all tx ∈ block do
7:     tx_size += tx.size()
8:     for all input ∈ tx.inputs do
9:       tx_input += input.size() - script.size()
10:      script_in += script.size()
11:    end for
12:    for all output ∈ tx.outputs do
13:      tx_output += output.size() - script.size()
14:      script_out += output.size()
15:    end for
16:  end for
17: end for
18: plot a pie chart with the following data: tx_input, tx_output block_header,
    transaction_header script_in, script_out
```

Result

Results are showed using a pie chart, where the whole circle represent the entire size of the blockchain analysed and every slice represent the space used by a specific type or group types of data.

Two different chart is displayed since they represent respectively two different range of blocks in the blockchain. The choice comes from two factors, the first one is that the first part of the chain is characterized by a very large amount of blocks that encapsulated a only few transactions, which brings to have a number of block header comparable with the number of transactions, which does not represent the right blockchain distribution of data but just the initial phase of the chain. We have identified this range from block zero to block 200 000. The second factor is that in the block 478 000 it was introduced the *segwit* type of data.
Therefore, the first chart represent the range of blocks from the first stable point until the introduction of the segwit data and the second chart represent the data from the *segwit* introduction until the end of the blockchain.
To have a better understanding of the following pie chart we are going to make a list of the label of the pie charts and the related fields taken in consideration for that specific field. All the name used represent the original Bitcoin's name used in the Bitcoin protocol and expanded in the Appendix B.

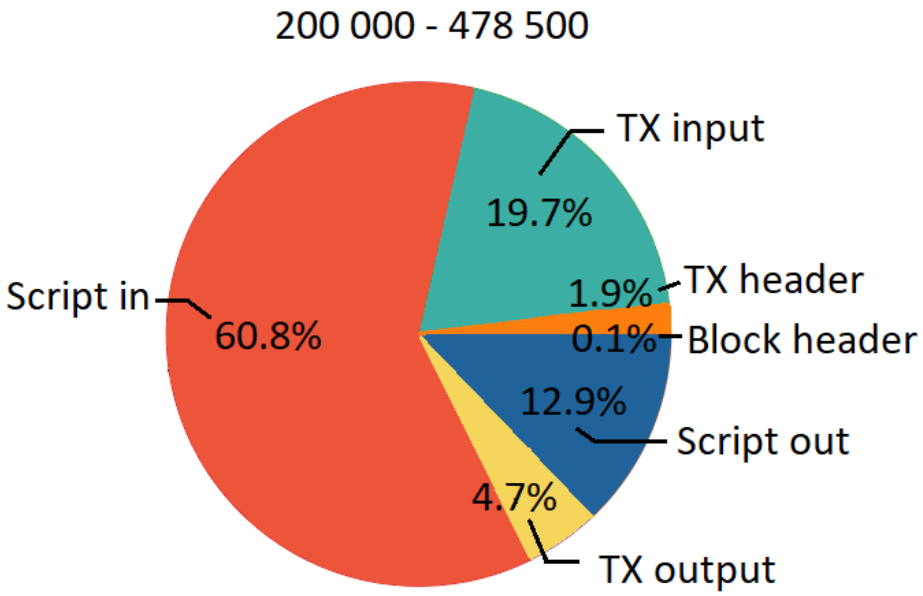

Figure 5.9: Bitcoin's Fields size distribution in the first part of the chain, from block 200 000 to block 478 000.

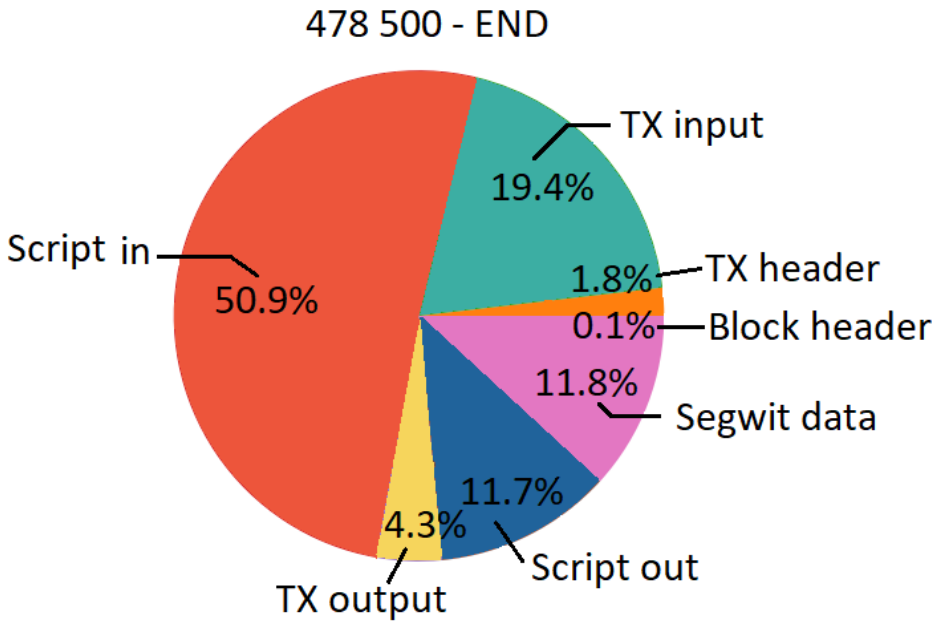

Figure 5.10: Bitcoin's Fields size distribution in the last part of the chain, from block 478 000 to the end.

- **Block header**: `version`, `prev_block`, `merkle_root`, `timestamp`, `bits`, `nonce`, `transactions_count`;
- **Tx header**: `transaction version`, `flag`, `tx_in count`, `tx_out count`, `tx_witnesses`, `lock_time`;

- **Tx input**: `previous_output, script_length, sequence`;
- **Script In**: `script`;
- **Tx output**: `value, pk_script length`;
- **Script Out**: `pk_script`.

As we can see, the majority of the blockchain data is represented by transactions, which are mainly formed by *Input Script* for 60.7% and by *Output Script* for 13.0%. Transactions' header and blocks' header are almost irrelevant since together represent the 2% of data.
From the introduction of the *Segwit* data, a 10% of data are transferred from the *Input script* to this new form of script field. So, the field distribution almost maintain the same distribution along the first and the second part of the blockchain.

5.3.2 Scripts Distribution

Description

The goal of this experiment is to evaluate the entropy of the most space relevant data type in the blockchain. Based over the previous experiment 5.3.1 the fourth most bigger sections are the *script* both *input* and *output*, the *segwit data* and the transactions' input header.
The transactions' input header is mainly composed by flags or version fields, therefore their entropy are going to be evaluated separately from the entropy of the script's fields. *Input scripts*, *Output scripts* and *segwit data* contain the same types of data, which are Bitcoin's opcode. This experiment aims to characterize the frequency of scripts, that means to figure out the number of occurrence of every single script.
In other words it is going to measure how many different script are written in the Bitcoin blockchain, an equivalent analysis is done during the preliminary phase of a dictionary compression algorithm. The most common script are later replaced with shorter pointers.

To see which is the structure of a Bitcoin script read the Section 2.2.1.
According to Bitcoin white paper every new payment should create and use a new Bitcoin Address [1] for privacy reason. Therefore we do not expect high redundancy in the script fields since every script should contains a new address, which cause a different script. However, we want to measure it, since the length of the script can make the process of replacing them with pointer, worthy.

Setup

- download the entire blockchain data, not both the blk and the *chainstate* folder but just the blk data folder. The data can be retrieved in the data folder of a Bitcoin client with the entire blockchain downloaded.
- download the Alecalve blockchain parser [41].
- modify the Alecalve blockchain parser introducing the function explained in the section: C. This is necessary to retrieve specific data during the parsing of the data, which are not allowed by the offered parser.

Strategy

The strategy aims to parse sequentially the entire blockchain, starting from the bottom. During the process it needs to count the number of occurrences of every specific script. The structure used for the goal is a key-value database, where the key are the script's and the value are the occurrence of every, which are updated during the parsing process.

Script are millions and represent the majority of data in the Bitcoin blockchain we need to analyse. Therefore, it is not feasible to insert all of the script in a standard key-value structure and different strategies were applied to achieve it.
The structure used is a LevelDB [42], developed by Google and used also by the Bitcoin client to store the blockchain data.
First of all, the code creates two structures, the first structure, called *db_in* stores the *Input script* and *Segwit data*, since they are two different type fields to represent the same information, so scripts used can be equal.
The second structure, named *db_out*, is used to store the *Output scripts*.

In the worst case each script is unique and used only once, which means that our database structure must maintain 100 GB of unique keys. If this is the case, even in a slightly better scenario, the structure will be too slow.
In order to further reduce the keys in the database, we have developed a pruning strategy that aims to remove unnecessary keys. We have made the hypothesis that it is more likely to find similar data and scripts nearby. So we assume that the older a script is, the less likely it is to find another occurrence of the same script.

So, first we changed the structure used to store also the time the script was created, within the value fields. Then, every time the structure used becomes slow, we did a flushing operation. We flush the old keys of the database that have a value greater than 2 in another structure containing only the keys, and delete the older keys that have no more than 1 occurrence.
The reason is that we are interested in characterizing the occurrence of the script, to estimate whether a strategy based on compression by replacing the value with a pointer will be noteworthy.

So, in the end, are used two Python dictionary to maintain the first newest information and the other two LevelDB, *db_in* and *db_out* to maintain the frequent scripts.

Algorithm 8 Script occurrence analysis

```
1: from python-bitcoin-parser import Blockchain
2: InScript =  # Dictionary
3: OutScript =  # Dictionary
4: chain = Blockchain(blockPath)
5: for all block ∈ chain do
6:    t_in = 0, t_out = 0
7:    for all transaction ∈ block do
8:       for all input ∈ transaction.inputs() do
9:          t_0 = time.now
10:         InScript[input.Script()] += 1;
11:         t_in = time.now - t_0
12:      end for
```

```
13:     for all output ∈ transaction.outputs() do
14:       t_0 = time.now
15:       OutScript[output.Script()] += 1;
16:       t_out = time.now - t_0e
17:     end for
18:   end for
19:   if t_in > threashold then
20:     flush frequent scripts from InScript into db_in
21:     delete old scripts from InScript
22:   end if
23:   if t_out > threashold then
24:     flush frequent scripts from OutScript into db_out
25:     delete old scripts from OutScript
26:   end if
27: end for
```

Result

The results shows the size of the two LevelDB, the *db_in* which contains the *Script input* and *segwit sequences* that has more then 1 occurrences in the blockchain and the *db_out* which contains the same information but for the *Output script*
We have count the number of blocks, transactions and both input, outputs scripts in order to estimate the average frequency of each elements.

number of blocks	545 000
number of transactions	347 389 259
transactions' inputs	888 049 876
transactions' output	945 016 460

The following table contains the size of each database we have obtained, which contains only scripts that occur at least two times, the number of unique keys in the database and the average size of each elements.

database content	size	database unique keys	average element length
script in + segwit	177 M	4.1 M	37.6 bytes
script out	1.3 GB	40.5 M	24.8 bytes

The entire analysis of the Bitcoin blockchain takes 3 days using the described setup.

During the entire analysis only the frequent scripts are identified however, in order to limit the computational time of the algorithm, it is not saved the exact amount of frequency for every script. A possible upgrade would be to design a way which is able to store the exact number of occurrence of every script, which can be very useful once this script needs to be replaced with pointer and the more frequent script can be substituted with shorter pointer.

5.3.3 Smart Compression

Description

This experiment aims to test a customized compression strategy, designed using the result obtained in the last two experiments: 5.3.2 and 5.3.1. The new approach is to

optimize the fields of the blocks data structure, to replace the redundant scripts with pointer and to further compress the obtained data using the best compression library, which was identified in the experiment 5.2.2.

This customized compression is made by three phases, which regards different aspects of blockchain's data that can be compressed.

1. **Fields optimization** which aims to redefine the structure or the way in which specific fields are stored in the memory looking for a space reduction. As we have seen in the section 4.3 there are different fields that can be optimized. With a sequential parsing of the chain, this phase will read the data written in the legacy structure and providing the same data using the optimized structure. The encoding used needs to be loss-less to perform a reversible process.

2. **Elimination of data redundancy** aims to reduce as much as possible the redundancy within Bitcoin data. As seen in the previous experiment, the *Scripts fields* are the most space demanded and also lots of scrips are redundant. Replacing the redundant scripts with a pointer to a structure that contains the real value will reduce the necessary space.

3. **Legacy compression** is applied at the end to produced the final version of the data, which aims to compress fields not already compress by our strategy. The library used to compress the data is the same that the Experiment 5.2.2 has evaluated to be optimum, we expect the data perform in the same way for optimized and not Bitcoin.

Setup

In order to made the customized compression the following things needs to be done:

- The entire Bitcoin blockchain data needs to be stored using the official format, the "blk00XXX.dat" files. On the contrary the UTXOs folder is not necessary, since the goal of this experiment is just compressing the Bitcoin data and the UTXOs structure is used only to increase the speed and usability of Bitcoin software.

- install the Python LevelDB library which is able to manage a LevelDB structure that is used by the code to take advantage by an external data-structure to store data that cannot fit into a in-memory structure.

- download and install the python bitcoin blockchain parser which is retrievable using the following link [41].

- modify the bitcoin blockchain parser code following the instruction available in the Appendix C.

Strategy

The entire smart compression is made by 3 sequential steps.

In the field optimization phase the code needs to compute the frequency analysis over the *Scripts*. In particular the python code parses sequentially the blockchain data and

builds in real-time a structure that contains only *Scripts* which occurs at least two times in the blockchain. The data analysed are *Input script*, *Segwit data* and *Output script*. The first two are stored together in a LevelDB named *db_in* and the Output script are stored in a database named *db_out*.

Then, during the compression step the script that are recognized as frequent are replaced by the key value that is related to the same script in the *db_in* or *db_out* structure. According with the *Huffman coding*, see Section 2.3.2, the length of the key should be choosen accordingly with the frequency of the value. The higher the frequency of a value, the shorter its key should be. However, the frequency analysis, see Section 5.3.2, is capable to just recognize a redundant script from a unique one. Therefore, this experiment uses keys with the same length which is decided accordingly with the numbers of Scripts that needs to be indexed. *Script outputs* which has at least 2 occurrences are more then 40 M, so the length of the keys is 4 bytes allowing to index 4 billion elements.

Also, the code implement an intermediate in memory structure to achieve the data recurrence analysis. The speed of a LevelDB cannot be compared by an in-memory Python dictionary. With our configuration the code takes 5-6 days to run, other implementations take longer. The analysis is done using this intermediate structure, a dictionary where key are data and tuple which contains the number of occurrences and the number of the last block where this data is found. However, since the structure cannot be too big, after a certain amount of blockchain blocks analysed the `older` data are deleted if there are not enough occurrence, or inserted in the database and then deleted.
The operation of flushing the data in the database increases considerably the amount of time, also the control of the presence of the data, the recovery of its possible value and the updating with the correct occurrence takes more time.

For the first part of the analysis it can be used the same code of the experiment 5.3.2 replacing the pointer value according with the explanation.

The **Second step** consist of parsing the entire blockchain from the beginning until a decided point and replacing recurrent data by pointer of 4 bytes and redesigning the non optimized fields.
Data are analysed splitting data in blocks, retrieved sequentially using the Bitcoin blockchain parser. Once a blocks is provided all its part are analysed and written sequentially, according with the new encoding in the data block.

Also, it is inserted a 1 byte flag in every transaction's header, transaction's input and output. The meaning of the flag is different according to its position and it is composed by 8 flags, one for every bit. The specific meaning of these flags are explained below.
According to the Section 4.3 there are multiple fields optimized. The list of all the modifications follows:

- transactions' header:
  - version: Bitcoin has only two transactions' header version B, so only 1 bit of information is necessary. This field has currently 4 bytes.
  - flag: this flag represents the presence of Segwit data section, since it is a flag it is a binary information however the field is optional and currently occupies 2 bytes.
  - lock_time: this field is not used most of the time and it is replaced by default value.

The designed solution aims to use only 1 byte flag placed in the beginning of the transaction, bit[0:1] is used to indicate the transaction version number, bit[2] is a bit flag to replace the segwit flag, bit[3] is a flag to indicate whether the `lock_time` has the default value and if not the field `lock_time` is copied after the bit flag.

- transactions' input data:
  - previous_output: this field has 32 bytes that represent the tx hash + 4 bytes for the output spent, the hash can be substitute with the block height (4 bytes) plus the transaction position into the list. So, the output number field can be replaced with the *varint* field to replace the fixed 4 bytes field.
  - script_len and script: if the script is frequent, these value are replaced by the key which represent the script in the *db_in* structure and the length of the key. Otherwise these fields are left invariant.
  - script: this field is represented using 4 bytes but it is most of the time set by default.

  The solution that we have implemented is to add a flag at the beginning of each input where: flag[0] indicates where the previous output is substitute with the block and tx position or not, bit[1] indicates where the script is replaced with a pointer or not, bit[2] indicates if the script is default.

- transactions' output data:
  - value: this field is substituted by a variable integer field.
  - script_len and script: are maintained if the script is unique, othersiwe are replaced according with the same procedure of the input script field. In this case if the script is frequent the key is an index of the *db_out* structure.

  A flag needs to be added at the beginning of each transaction's output.

- transaction's body: In the transaction's body, we can found *transactions' input*, *transactions' output* and in the last version also the *Segwit data*. We compressed also them, with the following rules.
  The original *segwit data* is sequences of fields with var_int leng and the related scripts. We added a 1-byte flag to each script at the beginning to indicate where we replaced the script with a pointer or not. Then the var_int leng could be remain the same as before if the script is not substituted otherwise it will be replaced by the length of the pointer and the script is replaced by a pointer.

The **Third step** is a standard compression applied over the data we have produced. The Python code applies the LZMA compression library.

Result

The result achieved is based over the partial blockchain compression, it is not compressed the whole blockchain but blocks from block zero to the last block minus 20 K blocks. We have estimated that this is the part of the blockchain that can be compressed in a common situation where users aims to maintain a fast access to the most likely useful

data and compress the other part.
As we can see in the Figure 5.11 the size of the compressed section of the blockchain is 167 GB, in the end of the process we reduce the data up to 92.98 GB which correspond to a reduction of 44.3%. This result is better than what traditional compression has achieved, which has been tested in the Experiment 5.2.2 and has led to a compression of 38%.

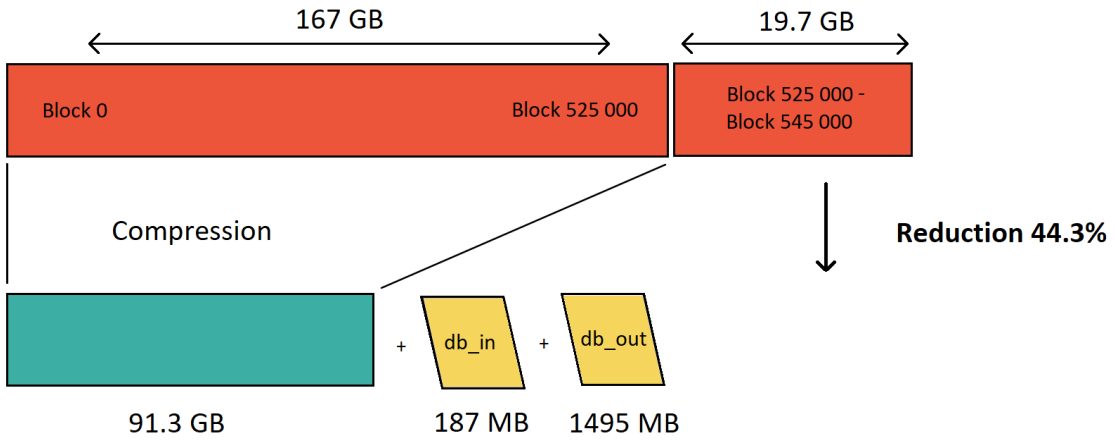

Figure 5.11: The chart shows the difference from the initial blockchain size, represented by the red blocks and the compressed blockchain represented by the green blocks, which are the file compressed plus the two additional database: *db_in* and *db_out*.

In the Figure 5.12 trends of file compressed during the entire process are plotted. It is possible to visualize how the file size grow with the number of blocks, how the field redesign and pointer substitution reduce the size and also how the other compression phase performs with the LZMA compression library.

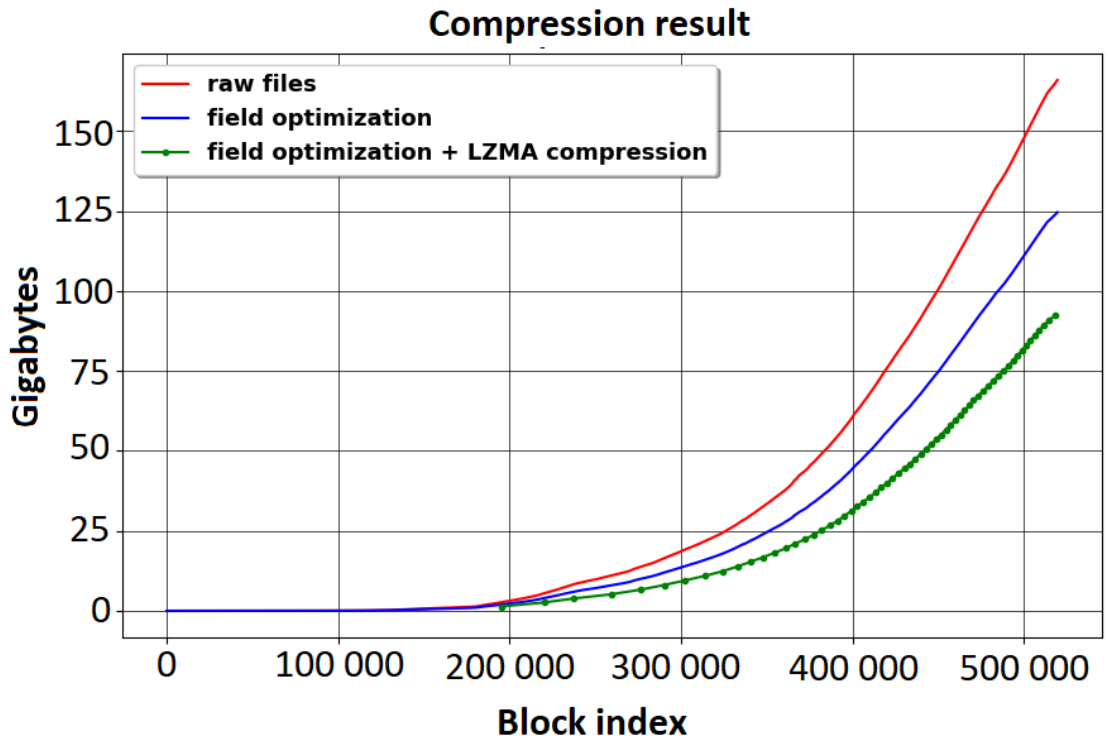

Figure 5.12: Compression result along the process with the 3 different stage of compression, no compress, field optimization, legacy compression. The green red point at block index 400 K shows that the raw blockchain size from block 0 to the given block index is about 6.250 GB, at the same block index the size of the optimized chain is about 4 GB and the green dot shows that the compressed file is just 3.5 GB.

5.4 Space optimization for full node

Description

The experiment aims to estimate the space requirement to maintain enough data to allow a full-node to be able to verify new blocks and transactions without the entire blockchain downloaded.

Currently a Bitcoin node which aims to verify a new transaction or a new block must have a trusted version of the previous block in the chain and a way to verify if all the contained transactions are valid. Also, to verify a transaction's validity all the transactions' input needs to be verified. Therefore it is necessary to maintain a trusted version of the `./chainstate` plus a way to verify if a UTXOs actually belong to a trusted blocks.
The last action is done thanks to the Merkle Tree's root hash which is saved in every block, however the sibling path is necessary, and usually it is retrieved from the stored data but we aim to store it separately to avoid the constraint of maintaining the entire blockchain.

This experiment aims to verify the precise amount of space required by the strategy above, which has to stores all the blocks' header, UTXOs and all the *Sibling Path* related to the UTXOs.

Setup

The following actions are necessary:

- Download the `./chainstate` folder of a Bitcoin node.
- Then, download also the `./blocks/index` folder.

Strategy

The experiment calculates the space requirement as follow:

- `./chainstate`: the UTXOs' folder, it can be just query the size of the folder;
- blocks' headers: the size of the block's header is fixed so the size is = headers_Size * number_of_blocks;
- sibling path: can be calculated multiplying the Merkle tree node's size by the deep of the specific transaction's Merkel tree minus one.

In order to retrieve the last information, we need to know the sibling path associated to every UTXOs. Each UTXOs is associated to a transaction, which is mined in a block and the number of transaction mined in the same block determine the deep of the associated Merkle tree.
So, firstly we parse all the UTXOs and for each UTXO we can decode the block's height. Then, we parse the `./blocks/index` folder and for each blocks we can read the number of contained transactions.

Algorithm 9 Full-node space requirement

```
1: cs = open(./chainstate)
2: idx = open(./blocks/index)
3: blocks = [0...0]
4: for all UTXO ∈ cs do
5:     blocks[UTXO.block_height] += 1
6: end for
7: blkTx = [ ]
8: for all block ∈ idx do
9:     blkTx[block.height] = block.transactions
10: end for
11: tot_size = 0
12: for all i ∈ range(0, blockchain-len) do
13:     tot_size += hash256-len * cs[i] * (log2(idx[i]) -1)
14: end for
```

Legend:

blocks is a list where the index is the block's height and the value is the number of UTXOs for that blocks;

blkTx is a list where the index is the block's height and the number of transactions contained in the block;

Result

In the Figure 5.13 is shown the result of the experiment.
The chart shows the cumulative distribution function of the space used by the sibling path of all UTXOs.

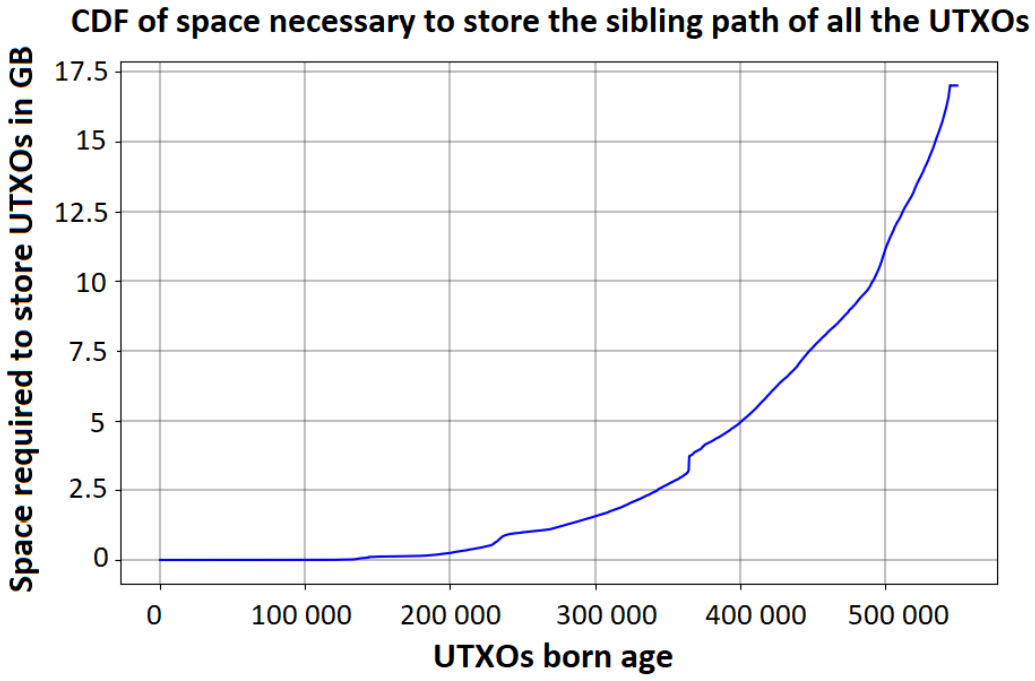

Figure 5.13: The chart shows the amount of space required to save the sibling path for every UTXOs. The point at index 400 K shows that it is necessary 5 GB to store the Sibling path of the UTXOs from the beginning of the chain until the block at that height.

In the x-axis we have the blockchain block height, the function represent the space necessary to store the *Sibling path* of every the UTXOs.

As we can see from the chart most of the data is produced from the latest part of the chain, where the blocks contains more transactions and so each sibling path requires more space.

We can also take in consideration the data that comes from the second experiment that show that with 95% of probability a UTXO is likely to be spend within 25 K blocks, so if a miner decides to maintain in the memory only the data related to the last 25 K blocks the amount of memory is only $\sim$ 3.735 GB over the entire amount of data which is 17.011 GB. The entire *./chainstate* folder, which contains the information related to the whole chain is 2.8 GB, differently the *./chainstate* folder which contains only the last UTXOs is: 811 MB.

Chapter 6

Results Evaluation

This project analyses different aspects of blockchain data, focusing on space optimization. In particular, the experiment aims to lower the space required by the Archival, Full and Auditor nodes. Despite the majority of the experiments are done over Bitcoin data, the insight concern all the coins developed on the Bitcoin blockchain structure.

6.1 Transactions' output

Transactions' outputs are one of the most critical section of the blockchain protocol and structure. They are the value that can be exchanged and the client nodes need to maintain them securely in a fast and secure structure until their expenditure.
The first part of our analysis aims to study their behaviour, while the goal of the first experiment aims to characterize the UTXOs lifespan. All the nodes which aim to participate actively in the network store UTXOs in a folder named *chainstate*, which allows a fast access method to verify all the incoming new blocks and transactions. However, as described in the Section 3.2.1 and our experiment confirm that the *chainstate* folder contains many transactions' output that will probably never be spent. The Bitcoin blockchain presents some transactions' outputs, dating back to the creation of the coins, which were never spent and are still loaded in the *chainstate* folder.
These UTXOs are produced by the transactions' output whose owner has lost the key, or the one whose script stores just data and would never be spent. UTXOs can also be produced by the one with a too low value to be spent, which causes the *"Dust"* problem. However, there is no way to distinguish them from UTXOs managed as assets and maintained voluntarily for a long time.

Our analysis has quantified the amount of UTXOs on Bitcoin and Litecoin blockchain.

- Bitcoin: analyzing a chain long 545 K blocks, we have figured it out that 80.81% of the blocks contain at least 1 UTXO. The initial part of the chain is almost UTXOs free however, the remained part is characterized by UTXOs uniformly distributed.
- Litecoin: the chain is made by 520 K blocks and 34.49% of them contains at least 1 UTXOs.

Both the blockchains analysed presents an incremental increase of unspent transactions' output in the last part of the chain. This behaviour is normal since recently transactions' output has a standard physiological period.
However, chains contain also particular sections characterized by a huge amount of

UTXOs. This behaviour can be produced by a software bug that causes the loss of users' key or by protocol bug that creates not spendable transactions' output. Both the cases are indistinguishable from a user perspective therefore, users cannot just remove them from the *chainstate*.

The second experiment is focused on transactions' output lifespan characterization. During the initial stable phase of the blockchain, the average lifespan of a UTXOs is just 1400 blocks, with the ageing of the chain, the lifespan grows linearly reaching 5000 blocks in the newest chain's part.
The 90% of the UTXOs follows the behaviour of the average measurement, at least at the beginning. However, it reaches an age of 7500 blocks, 2500 more than the average. Also, the lifespan trend which represents the 95% of all the UTXOs is far from the average, starting with a lifespan of 6000 blocks it reaches 25 K blocks. This behaviour shows that a little part of UTXOs is spent a lot of time later than the majority of the UTXOs.
To have a preciser view over this data in the chart 5.4 we have plotted the cumulative distribution function of the age of the UTXOs contained in the most recent snapshot of chain we have analysed, which is at block 545 K.

6.2 Data compression

This section analyses all the result obtained from the study done over blockchain data compression. Using a black-box approach, in the experiment 5.2.1 we have initially tried to compress the entire blockchain data.
We have used different dictionary-based compression libraries, which are the only general-purpose library that can fit the blockchain data structure. In particular, we have tested 3 different libraries which are *LZMA*, *LZO*, and *LZ4*.
The experiment, using all the libraries mentioned, has tested the compression over different formats of files indeed the size of files vary from 100 MB to 2000 MB also, the blockchain blocks' which compose the file can be ordered or unordered.

Initially, we have used the default libraries configurations for all the tests and the result shows that the *LZMA* library performs better overall tests.
Also, comparing the different blockchains, the Dogecoin achieves the best performance in terms of compression. Indeed the best result for the Bitcoin blockchain is a savings of 30% instead, it is possible to achieve a compression of 35% in Dogecoin and only 28% for Litecoin.
Furthermore, comparing the organization of the block inside files we have figured it out that ordered blocks perform better according to the test made over Bitcoin blockchain. However the compression performance is constant, independently from the file dimension considered, and it is 30% with the *LZMA*, 16% for *LZ4* and 14% for *LZO*.
All the test results are also affected by a variation which is, in the Bitcoin ordered files, around $\pm 5\%$.

After the preliminary tests, we decided to improve the performance focusing on the most promising result. Therefore we analysed files with a size of 2 GB made with ordered Bitcoin blocks. Each experiment is repeated over 10 different samples, which are made with the mentioned characteristics but differ in the content, indeed each sample is filled taking blocks from a different section of the blockchain.

Different libraries configuration are tested and the result shows that in a few seconds it is possible to achieve a saving space of 17% using the *LZ4* started with the lowest compression level available.
The best space compression is achieved thanks to the *LZMA* library, that reaches a saving space of 38.5% and a variation of ±3.5.
From the parameter point of view, it achieves the best performance setting the *MF_BT4* as tree parameter and the maximum dictionary size which is up to 1.5 GB however, the compression level is not relevant once the other two parameters are already chosen.
As a side effect of the use of the *LZMA*, we have the time usage indeed it requires 3.5 hours for each 2 GB sample file.
Others libraries are not comparable in terms of compression however, *LZO* and *LZ4* are optimized for time-saving and they use less than 5 minutes to compress the data on the worst case.

To achieve better compression performance, we moved to a white-box approach.
Firstly we have done an analysis to characterize the composition of the data, which are the nature of different data and how are they distributed. The experiment 5.3.1 shows that in the Bitcoin blockchain transactions' takes the majority of the space, in particular in the last part of the chain transactions' inputs produce the 82.1% of all the amount of data also, the transactions' outputs produce the 16%. Moreover, the script data both in input and output represent the 74.4% of all the chain data, see Section **??**.
We figure it out that with the optimization of filed in the script input or output we can save gigabytes of data for each byte reduced in one of these fields, see Section 5.3.1. Transactions' input and output are almost a Billion in the Bitcoin blockchain and the efficiency in terms of storage is crucial.

Thanks to the experiment 5.3.2 we have also verify the entropy of the scripts.
Even if a precise analysis is very difficult to do since the data that needs to be enumerated are very scattered along the entire chain, the experiment provides a quantitative idea.
The experiment shows that the Bitcoin chain has 888 Million of transaction input script and 4.1 Million of them are at least duplicated. The 4.1 Million of Script are not unique and they represent the 0.005% of the entire amount of scripts, the average length of these elements is 37.6 bytes.
On the other hand, the chain has 945 Million of transactions' output and 40.5 Million of them are duplicated at least 1 times. The average length of these elements is 24.8 Bytes.

After this analysis we have built a complete fully customized compressor which aims to compress all the blockchain data using:

- fields optimization;
- store the redundant script in another designed structure and replacing frequent scripts with the pointer to the new structure;
- another compression is done by a legacy method. In particular, we have used the optimized *LZMA* compression library.

Since the future application of this compression is to compress the chain part which is not likely to be used sooner the experiment was tested only over this fraction of the chain. In particular, we left the last 25 K blocks for faster access.
The compressed chain has a size of 167 GB and the compressor obtained a savings of 44.3 % which is the best result reached. The final size of the chain is 91.3 GB but it needs to be used in conjunction with two databases of size 187 MB and 1495 MB.

The chart 5.12 shows how each step of the compression affects the final result. Starting with a raw file of size 167 GB thanks to fields optimization and pointer replacement we can reduce the space to 125 GB then, thanks to the legacy compression we can save other 34 GB achieving 93 GB of file size.

6.3 Secure light client and auditor node

Thanks to the last improvement and data design we provide a secure and light way to participate in the blockchain ecosystem.
The temper-proof and safety are achieved without relying on an entire local copy of the blockchain. The user needs to maintain a chain made only by the header of the blocks, the *chainstate folder* plus a few additional information to achieve the goal.

In order to participate in the ecosystem of the blockchain is necessary to be able to verify the incoming blocks and transactions. The blocks verification reflects over multiple transactions verification.
To verify a transaction is necessary to prove the validity of its input which needs to be mined inside a valid block. A light client in order to make this operation relies on trusted nodes without the ability to check the correctness of the information received. However, it is possible to store the *Sibling path* of every UTXOs and relying on them during the verification process.

Storing the Sibling path of every UTXOs allows the user to verify wherever an incoming transaction uses only valid input. She can verify the claimed UTXOs relying on her trusted headers chain which is also updated on real-time.

The experiment 5.4 in the chart 5.13 shows the amount of memory needed to store all the UTXOs sibling path. It results in just 17 GB to store the UTXOs for the entire blockchain. However, in the initial part of the analysis we have seen that 95% of the transaction spend UTXOs not older than 25 K blocks, which means that if a user decides to store the information related only to the most likely spendable UTXOs, discarding the others, it just needs 2.8 GB of memory plus the mandatory *chainstate* which is only 811 MB.

Chapter 7

Conclusion

The last 30 years have seen some of the most rapid technological advancements in human history and the blockchain has the potential to be the next major disruption technology. The blockchain has two main abilities: it introduces the concept of security in all information and transactions, while entrusting it to a reliable and secure protocol.

The blockchain was introduced in 2009 with Bitcoin, and it is the technology behind all of today's cryptocurrencies. Financial institutions were the first to notice it, still as a new payment system. However, since 2009 the blockchain has captured the interest of other markets as well, and it is now improving many industries such as supply chain, health care and IoT, because of its decentralization, persistence, auditability and anonymity [1].

However, the blockchain still has two main issues, which limit its adoption and usability. Our work improved the state of the art, in order to solve both of them.

The first issue is linked to the huge amount of data that a blockchain application produces. These data need to be replicated in all the devices involved in the ecosystem, producing an enormous waste of resource. As a matter of fact, we have presented a methodical analysis over the Bitcoin protocol and a statistical analysis over its data value. The search for patterns aimed to figure out a possibility of optimization. The goal was to find the best lossless compression method. After a first attempt, conducted with the available compression libraries, we decided to design a custom method, which led to the reduction of the space used by the Bitcoin blockchain almost to half, specifically up to 44.3%.

The second issue regards the absence of a provably secure thin protocol, capable of guaranteeing security and privacy. The only existing way to participate in the network is through the download of the entire blockchain. Consequently, this issue creates an initial barrier for smartphones and IoT devices. Our study has shown that it is possible to break it down, by lowering the disk space requirements to 4 Gigabytes.

What we know for sure about the future is that it is going to be filled with smartphones and IoT devices. Taking this fact for granted, both solutions found through this project, not only bring true and effective improvements to the topics of light secure protocol and blockchain compression but also pave the way for future efficient implementations.

Appendix A

How to manage blk files

This Appendix describes how to produce different samples from Bitcoin blockchain data, or from any blockchain that saves its data maintaining the block structure. All Bitcoin data is stored in structured blocks as described in the Section 2.2.4. However, in order to do compression experiments, we need samples.
The following text explains how to produce samples from Bitcoin data and what is the main constraint to consider. The explanation is not limited to Bitcoin data but applies to all blockchains based on the Bitcoin structure, including Litecoin and Dogecoin. Some of the techniques described may also be useful for other types of blockchains that have the block structure representation.

The algorithm we have created is intended to produce several samples of blockchain data. Once you need to do an experiment you need to replicate the result and to do so you need several samples. The algorithms describe how to extract a blk file from the blockchain structure of Bitcoin.
One of the main features that can relate to the production of the sample is the fact that the blocks inside it can be ordered or not, for reasons of compression this choice may affect the analysis and therefore we have chosen to create two ways to create this sample to check if there are differences in the use of it.

All blocks, needed to produce samples, can be retrieved by analyzing Bitcoin blk files or by using the bitcoin database structure to retrieve a specific part of the blockchain data.
Both ways are interesting, we should look at the Bitcoin structure which is made as follows: as we described in the Section 2.2.4, we know that a blk file is a sequence of `Magic number`, `Byte length` and `Raw data block`.
Another important thing that needs to be considered is that it is necessary to avoid data replication within the same sample since the experiment for compression performance can be affected.

Both algorithms will use the open-source library called *Blockchain python parser* which is available at the following link *https://github.com/alecalve/python-bitcoin-blockchain-parser*; we refer to the version of 15 March 2018.
The following two sections explain how to produce ordered and unordered samples of different sizes. Each type of sample must have 10 different versions, which has the same characteristics but has different contents.

A.0.1 Samples with Ordered blocks

The main problem is that the blocks are not organized inside blk files and to retrieve the correct order it is necessary to read the metadata structure which Bitcoin uses to store the actual position of each block starting from the block's index. However, there is an open-source project which provides a function to return an iterator on ordered blocks. This tool provides a method called `def get_ordered_blocks(self, index, start=0, end=None, cache=None)` which is able to provide an ordered sequence of blocks, it takes the first and the last index as parameters.
However, the iterator return block as python object but we need the binary content of it so it is necessary to modify the code as follows: inside each *Block* object it is stored the `hex` content, we add the method `get_raw_data()` inside the block class as it is shown in the code below. The following Python code shows the *block* object of the class and the method added.

```
class Block(object):
        """
        Represent a Bitcoin block, contains its header
        """

        def __init__(self, raw_hex, height = None):
                self.hex = raw_hex
                self._hash = None
                self._transactions = None
                self._header = None
                self._n_transactions = None
                self.size = len(raw_hex)
                self.height = height

        def get_raw_data(self):
                return self.hex

        ...
```

The following algorithm explains how to create 10 different samples of 5 different size, filled by ordered blocks.

Algorithm 10 blk ordered creator

1: `blockchain` = new Blockchain($blocks_path) # from blockchain_parser
2: `i` *: start blocks index*
3: `s` *: size desired*
4: `buffer` = [ ]
5: **for** `block` ∈ `blockchain.get_ordered_blocks($index_path, s,` ∞`)` **do**
6: buffer += *magic number*
7: buffer += len(`block`) *# in a specific format*
8: buffer += `block`.get_raw_data()

```
    if len(buffer >= size desired) then
      break
    end if
  end for
  save(buffer) as blk0000.dat
```

A.0.2 Samples with unorderd blocks

This section describes the procedure that can be used to create, as the previous method 10 different samples with 5 sizes, filled by unordered blocks. Here the process is simpler because the code uses the method `get_blocks(blk_file_name)` of the *Bitcoin python parser* specifying a blk file name and it will return an iterator over contained blocks, which are not ordered.

The raw-data return by the method used does not contain the magic number and the length of the block so the algorithm adds them in order to create a blk file that is fully compatible with standard Bitcoin blk files.

Algorithm 11 blk unordered creator

```
#from blockchain import get_blocks
blk_name : blk name start
s : size desired
buffer = [ ]
while true do
  blk_name = new name
  for all block ∈ get_blocks(blk_name) do
    buffer += magic number
    buffer += len(block) # in a specific format
    buffer += block.get_raw_data()
    if len(buffer >= size desired) then
      save(buffer) as blk0000.dat
      end
    end if
  end for
end while
```

In order to use this algorithm for other blockchains is necessary to change the magic number inserted accordingly.

Appendix B

Structure of Bitcoin blocks and transactions

This appendix aims to show the technical structure of Bitcoin's blocks and transactions.

Transaction

The table B.1 shows the general format of a Bitcoin transaction. The same format is used to store a transaction within a block.
Other formats are used to broadcast and transmit transactions inside the network since additional information were added [43].

Field Name	Description	Field Size	Data Type
version	Version number	4	int32_t
flag	If present always 0001, indicates the presence of witness data	0 or 2	optional uint8-t[2]
tx_in count	Number of transaction inputs (never zero)	1+	var_int
tx_in	A list of at list 1 transaction's inputsB.2 or source for coins generations	41+	tx_in[]
tx_out count	Number of transaction's outputs	1+	var_int
tx_out	A list of transaction's outputs B.3	9+	tx_out[]
tx_witnesses	A list of witnesses, one for each transaction input	0+	optional tx_witness[]
lock_time	The block number or timestamp at which this transaction is unlocked. 0 means not locked, <500M means block number and >= 500M means timestamp.	4	uint32_t

Table B.1: Bitcoin transaction structure.

The following two tables represent the transactions' *Inputs* fields structure B.2 and the transactions' *Outputs* fields structure in the table: B.3.

Field Name	Description	Field Size	Data Type
previous_output	32 bytes of the hash of the referred transaction + 4 bytes for the index of the specific output referred.	36	char[32]+uint32_t
script_len	The length of the script	1+	var_int
script	Computational Script for confirming transaction authorization	1+	uchar[]
sequence	todo	4	uint32_t

Table B.2: Bitcoin transaction input structure.

Field Name	Description	Field Size	Data Type
value	Transaction value, expressed in value of Satoshi.	8	int64_t
pk_script length	Length of the pk_script	1+	var_int
pk_script	Contains necessary information that a Bitcoin Script use to claim the ownership of this output	?	uchar[]

Table B.3: Bitcoin transaction output structure.

Block

In the table B.4 it is shown the structure of a Bitcoin block used to store it in the memory.
It is made by a fixed size section which represent the firsts fields, that contribute for 80 bytes plus the last two fields which are variable. They are a variable integer field necessary to represent the number of transaction plus the entire list of transaction which is up to 128 MB according to the Bitcoin protocol.

Field Name	Description	Field Size	Data Type
version	Block version information	4	int32_t
prev_block	The hash value of the predecessor of this specific block	32	char[32]
merkle_root	The hash of the root node of the merkle tree that caontaind all the hash of the transactions contained in this block	32	char[32]
timestamp	A unix timestamp recording the creation of this block	4	uint32_t
bits	The calculated difficulty target being used for this block	4	uint32_t
nonce	The nonce used to generate this block	4	uint32_t
txn_count	Number of transaction entries	1+	var_int
transactions	List of transactions	?	tx[]

Table B.4: Bitcoin block structure [15]

Appendix C

Alecalve blockchain Python Parser

This is the project of an open-source library created by the author called *Alecalve* for the analysis of raw data stored by Bitcoind, which is the most used Bitcoin client.

The data stored by Bitcoind uses a specific encoding and structure that is not part of the Bitcoin protocol, this library offers an easy way to iterate on the contents at the block level, transaction and values in it.

However, despite the many features offered, this library does not provide the ability to analyze raw data and does not provide a suitable reading for script type fields.
The most complex thing to do when analyzing Bitcoin data is to recover the correct order of the blocks, they are in fact stored in a random way. The correct order can be found by reading the database called *./chainstate* where for each block index is stored the exact location in memory of the beginning of the bytes of that block. This library offers this functionality in the form of an iterator of ordered blocks.

Below are described some ways to use the library in a simple way and to modify it appropriately so that you can read the data that are not normally made available.

C.1 Install

The project can be downloaded from the GitHub repository at the link: *https://github.com/alecalve/piton-bitcoin-blockchain-parser*. The version we are referring to is in Commits of 10 February 2018. However, future versions have not altered the validity of what we find described below.

The library requires Python 3 to work, in addition to the *plyvel* library which allows interfacing with the LevelDB database used by Bitcoind.
However, all the instructions to install the library are contained in the Readme.md file of the repository. Once installed, it can be freely modified without any problems.

C.2 Structure and modification

We will briefly describe the structure of the code and then show how we can modify it according to our needs.

As we can see from Figure C.1 the `blockchain_parser` path contains lots of Python file and the name are meaningful.

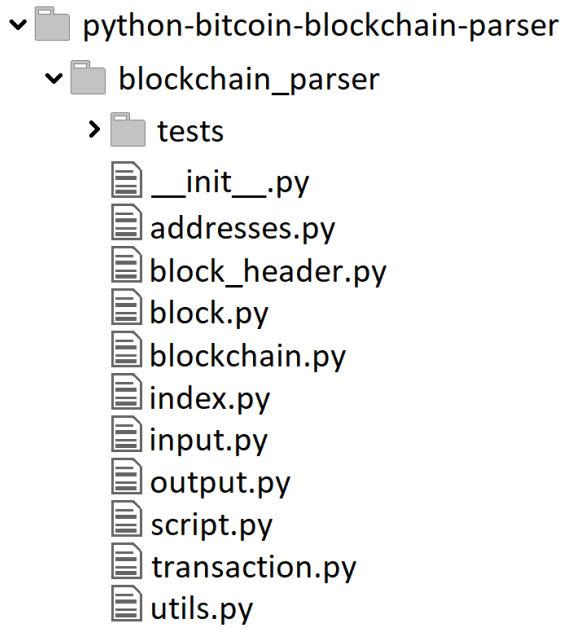

Figure C.1: The figure shows the Bitcoin blockchain python parser path organization.

The first class that needs to be used is *blockchain* written inside the `blockchain.py` file. The class needs to be initialized with the Bitcoind data path. Then it provides two useful methods:

- *get_unordered_blocks*: which return an iterator of *blocks* object unordered;
- *get_ordered_blocks*: this method provides an iterator as the previous one, but it accepts two indexes as a parameter which is the starting index and the last index of the blocks we need.

Both methods return an object of type block, which is a python object described in the `block.py` file.
This object contains the raw bytes for its representation, so when you need to parse the raw data add a method within the block class.

The object also contains a list of transactions, transactions are represented as a python object which is described in the file `transaction.py`. Here, unlike in the block, the raw data is not saved, in fact they are used during the initialization of the transaction object but then they are discarded.
To retrieve specific information that is not saved by default, such as the length of a certain variable field in terms of bytes, it is possible to intervene on the method of object initialization. In the method `__init__()` the raw data are processed and the various fields are initialized, here just introduce the variables we want to read and initialize them during this process. The python code sequentially iterates on the bytes and is very easy to understand and modify.

Bibliography

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system." https://bitcoin.org/bitcoin.pdf, 2009.

[2] Statista, "Market capitalization of bitcoin from 1st quarter 2012 to 4th quarter 2018 (in billion u.s. dollars)." https://www.statista.com/statistics/377382/bitcoin-market-capitalization/, October 2019.

[3] "Trends in cryptocurrencies and blockchain technologies: a monetary theory and regulation perspective."

[4] P. De Filippi and B. Loveluck, "The invisible politics of bitcoin: governance crisis of a decentralized infrastructure," 2016.

[5] "Can blockchain strengthen the internet of things."

[6] "Medrec: Using blockchain for medical data access and permission management."

[7] S. C. for Enterpreneurship and Technology, "Air applied innovation review, issue no.2 june 2016," 2016.

[8] K. Wüst and A. Gervais, "Do you need a blockchain?," in *2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*, pp. 45–54, IEEE, 2018.

[9] "How much data do we create every day the mind blowing stats everyone should read - forbes." https://www.forbes.com/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read/6a057dd260ba, May 21, 2018.

[10] M. Dameron, "Beigepaper: An ethereum technical specification." https://github.com/chronaeon/beigepaper/blob/master/beigepaper.pdf.

[11] Z. Zheng, S. Xie, H.-N. Dai, X. Chen, and H. Wang, "Blockchain challenges and opportunities: A survey," *International Journal of Web and Grid Services*, vol. 14, no. 4, pp. 352–375, 2018.

[12] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, pp. 382–401, July 1982.

[13] "Bitshares blockchain." https://github.com/bitshares-foundation/bitshares.foundation/blob/master/download/articles/BitSharesBlockchain.pdf, Nov 29, 2018.

[14] "Bitcoin is an innovative payment network and a new kind of money." https://bitcoin.org/en/.

[15] "Bitcoin wiki." https://en.bitcoin.it/wiki/, 19 November 2017, at 20:12.

[16] "Bitcoin - wikipedia script." https://en.bitcoin.it/wiki/Script, 26 May 2018, at 12:56.

[17] "Bitcoin developer: Transactions." https://bitcoin.org/en/transactions-guide#introduction, May 2019.

[18] "Bitcoin developer: Scripts api." https://bitcore.io/api/lib/script, May 2019.

[19] "Bitcoin - segregated witness (consensus layer)." https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki, 2015-12-21.

[20] "The long road to segwit: How bitcoin's biggest protocol upgrade became reality." https://bitcoinmagazine.com/articles/long-road-segwit-how-bitcoins-biggest-protocol-upgrade-became-reality/, Aug 23, 2017.

[21] "Bitcoin - mining." https://en.bitcoin.it/wiki/Mining, 25 June 2018, at 21:48.

[22] "Operating modes in bitcoin." https://bitcoin.org/en/operating-modes-guide.

[23] "Bitcoin fullnode." https://bitcoin.org/en/full-node.

[24] K. Sayood, *Introduction to Data Compression*. Elsevier, 2006.

[25] "Fundamental data compression," Oxford: Butterworth-Heinemann, 2006.

[26] S. Liu, "Bitcoin statistics." https://www.statista.com/statistics/647523/worldwide-bitcoin-blockchain-size/, October 2019.

[27] "Bitcoin release note 0.11.0." https://github.com/bitcoin/bitcoin/blob/master/doc/release-notes/release-notes-0.11.0.md, Aug, 2015.

[28] A. Gervais, S. Capkun, G. O. Karame, and D. Gruber, "On the privacy provisions of bloom filters in lightweight bitcoin clients," in *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC '14, (New York, NY, USA), pp. 326–335, ACM, 2014.

[29] E. L. ¡elombrozo@gmail.com¿ Johnson Lau ¡jl2012@xbt.hk¿ Pieter Wuille ¡pieter.wuille@gmail.com¿, "Segregated witness (consensus layer)."

[30] S. Delgado-Segura, C. Pérez-Sola, G. Navarro-Arribas, and J. Herrera-Joancomartí, "Analysis of the bitcoin utxo set," in *International Conference on Financial Cryptography and Data Security*, pp. 78–91, Springer, 2018.

[31] P. Tschipper, "[bitcoin-dev] test results for : Datasstream compression of blocks and tx's."

[32] E. G. Sirer, "How to compress bitcoin."

[33] J. Bruce, "The mini-blockchain scheme," *White paper*, 2014.

[34] "Mimblewimble white paper."

[35] B. B. F. Pontiveros, R. Norvill, and R. State, "Recycling smart contracts: Compression of the ethereum blockchain," in *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pp. 1–5, IEEE, 2018.

[36] M. Zima, "Inputs reduction for more space in bitcoin blocks," 2018.

[37] "Lzma compression library." `https://github.com/peterjc/backports.lzma`, 2019. version: todo.

[38] "Lzo compression library." `https://github.com/jd-boyd/python-lzo`, 2019. version: todo.

[39] "Lz4 compression library." `https://pypi.org/project/lz4/`, 2019. version: 2.1.5.

[40] G. Walker, "Learn me a bitcoin - locktime." `https://learnmeabitcoin.com/guide/locktime`, Aug, 2019.

[41] A. L. Calvez, "Bitcoin python parser." `https://github.com/alecalve/python-bitcoin-blockchain-parser`, 15 March 2018. Accessed: 2018-06-06.

[42] "Leveldb - non relational database developed by google." `https://github.com/google/leveldb`, Aug, 2019.

[43] "Bitcoin - wikipedia transaction." `https://en.bitcoin.it/wiki/Transaction`, 5 January 2019, at 02:16.