Initial draft

Blockchain Technology: Cryptocurrency and other Applications

by
Sandeep Kumar Shukla
Mohan Dhawan
Venkatesan Subramanian

February 1, 2021

Contents

Chapter 1

Introduction

Blockchain becomes a popular technology after its use in cryptocurrency such as Bitcoin and Ethereum. Nowadays it is preferred in various applications because of its decentralization nature and other features such as immutability, availability, public verifiability etc. It achieves the common goal in the decentralized network without the requirement of trusted third party. This section introduces the blockchain technology and its underlying concepts that is secret cipher, hashing and digital signature of cryptography, Consensus protocols of distributed system and pointer and tree of data structure. The discussion of this section is as follows

- Overview and history of the blockchain with its features
- The underlying concepts that are used for the implementation of blockchain.

1.1 Overview

The blockchain is a distributed data structure, which is replicated at various nodes or computers. Like linked list, it is a set of blocks which are connected to each other by a link. In case of a linked list, nodes are connected by pointers and pointers are basically memory addresses. The blockchain has a different notion of linking between nodes and each of these nodes are called blocks. Hence, a blockchain is a series of blocks and each block is connected to its previous block by a link basically block hash, which is a cryptographic technique.

Blockchain is replicated all over because replication gives number of advantages like if one of the replica gets corrupted, the other replicas are there to make sure that the integrity of the information contained in the data structure is maintained. Also replication gives guarantee of integrity of the data. Integrity means that the data once it has been agreed by all the relevant parties to add in the data structure and added, it has not been tampered with. Hence, nobody has come and changed the data and claim that this is the data that was put or add in. This means virtually impossible in a blockchain and that is the main property of the blockchain that it maintains the integrity of the data and as we will see that most of the applications where blockchain is used be it cryptocurrency or be it some other application.

Blockchain is distributed in the sense that the different computers involved in the blockchain platform actually are running distributed algorithms in order to maintain the data's consistency and integrity. The consistency of the data is maintained by a process called consensus. Consensus means that everybody agrees that the data that goes into the data structure is what they agree to put there.

What is blockchain used for? We know that many times we keep logs of events. For example, when somebody accesses your computer, the computer keeps a log of the user names and how they authenticated themselves. Microsoft Windows gives event logs for every event that happens like you open a new program on your machine or something crashes or you get connected to the internet. All these events are kept in event logs thus logs are very important. Similarly, When you do banking transaction bank keeps logs of when you interacted with its banking servers and what you did, what transactions you made.

The main problem with keeping logs without any notion of protection of the integrity is that somebody can tamper with the logs and somebody can delete some of the accesses. And therefore, later on when you check the log, you would not know some part of its history. Therefore, the blockchain is designed in such a way, so that it is an immutable ledger of events, which means it is a log that cannot be changed by a malicious party or by mistake. And therefore, all the data that you put in there could be event logs, it could be transactions, it could be various kinds of accesses and modifications you do to some other thing like a data or you do a property transaction.

All these logs are has to be kept in an immutable ledger and blockchain provides it. The tampering of the data is virtually impossible in blockchain. We do not say it is impossible to tamper, as we will see, as we learn more

that there if you have a very high computational power, which is almost impossible for individuals together.

In case you can gather that kind of computation power you can actually subvert this all the protection and change but since it is virtually impossible, we would say that this is a tamper resistant log. Hence, having these properties, we basically use blockchain as a platform to create and transact cryptocurrency. The bitcoin and ethereum are popular cryptocurrencies.

The first application of blockchain was bitcoin. The whole idea of creating currency, that whose transactions whose creation whose use everything has to be put in a tamper proof log, and without a trusted third party or without a central agency, which keeps track of this logs. After the popularity of Bitcoin, blockchain is considered as one of the integrity tools for distributed applications and applied in various applications such as healthcare, Internet of Things, Supply Chain Management, etc.

1.1.1 History

In 1982, David Chaum first proposed a blockchain-like protocol in his dissertation "Computer Systems Established, Maintained, and Trusted by Mutually Suspicious Groups" [15]. Further Stuart Haber and W.Scott Stornetta [16] in 1991 introduced the concept of linking digital timestamp of documents to make infeasible for the user in sending the document back dated or with later date. Later, along with Dave Bayer, Stuart Haber and W.Scott Stornetta [6] used the merkle tree to make the model efficient. In 2008, Satoshi Nakamoto improved the concept with Proof of Work (PoW), linking blocks without the requirement of trusted third party and called as block chain backbone of Bitcoin (later become blockchain). In 2015, an Ethereum blockchain featuring smart contract functionality was introduced. In January 2018, Hyperledger released the production-ready Sawtooth 1.0 permissioned blockchain and in January of 2019, the first long-term-support version of Hyperledger Fabric (v1.4) was announced. Since then there are various blockchain implementations are released.

1.1.2 Components and Entities

The abstract view of blockchain is shown in figure 1.1. Each block of the blockchain contains header as well as data in the form of merkle tree. The header of the block contains block hash, parent block hash, timestamp, nonce,

etc. The data are transactions, account balances, transaction receipt, etc. The attributes of blockchain header and data varies according to implementations. The different components and entities of the blockchain are discussed in the following.

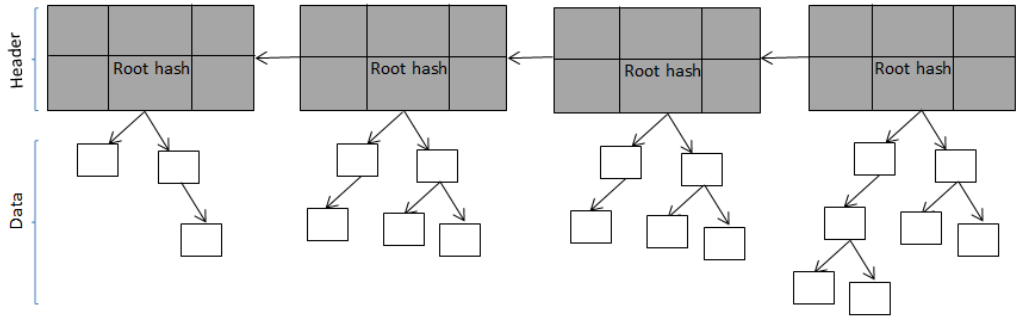

Figure 1.1: Abstract view of Blockchain

Block: Blocks are files where data pertaining to the application is stored. Data in a block are some or recent transactions made in the network and meta-data such as parent block hash, timestamp, nonce, etc. Each block is connected to the previous block through the hash pointer.

Genesis Block: The first block of the blockchain is called as the genesis block which does not have any previous or parent block. All users of the respective network should start with the same genesis block to ensure the correctness of the blockchain. For example, let us assume *Alice* and *Bob* wish to have together a private network but both are having different genesis block. Now, *Alice* generate the second block and send to *Bob* to commit in the chain. When *Bob* try to link the received block with his genesis block, it will give incorrect hash because the parent hash in the received second block and the hash of the *Bob* genesis block are different. In such case they cannot have a common blockchain. In case both have the same genesis block then there will not be hash conflict also they can have the common blockchain application.

Transaction: It is an atomic event that is allowed by the underlying protocol to perform an activity. For example, in cryptocurrency, transactions are the execution of payment transfer like *Bob* sends *Alice*, 10 BTC (bitcoin).

Fork: A project fork happens when developers takes a copy of source code from one software package and start independent development on it [14]. Fork in blockchain is the diverged chain as shown in figure 1.2. Forking in blockchain happens

- When a protocol of the implementation get changed. That is when there is software upgrade that introduces a new rule to the network. In this case, fork can be of two types in blockchain, one is soft fork and another is hard fork.
  - *Soft fork* - It is backward compatible that is a change of rules in the updated software that creates blocks recognized as valid by old software.
  - *Hard fork* - It is not compatible with the older software. In short, it is not backward compatible and all nodes should upgrade the software or protocol to be part of the updated network.
- In a situation that when two or more blocks have the same block height. For example, in figure 1.2, there are two blocks for block number 2, three blocks for block number 3, etc. However, only one block of the respective number can be part of the main chain. Other blocks are called as the forks.

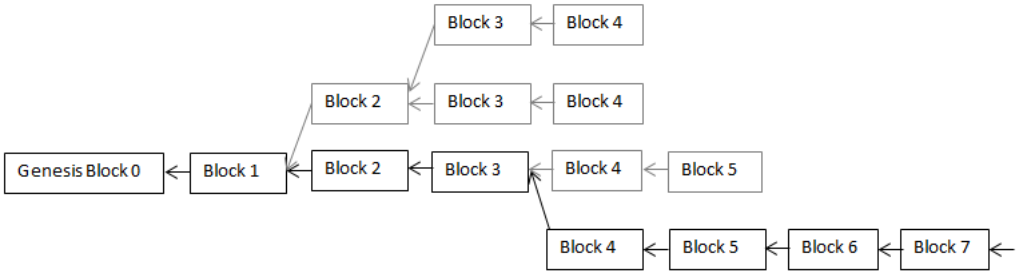

Figure 1.2: Blockchain Fork

Miner: Miners are responsible to generate or mine the blocks. Miner validate the set of received transactions and make a block including all valid

transactions and solve the puzzle for distributed consensus. The mined block will be broadcasted to everyone in the network for inclusion in the blockchain. The consensus protocol varies to one implementation to another implementation.

Nodes: Nodes are the participants of the blockchain network. The participating nodes can be full node or light node. A node which stores blocks header, data and mine blocks is called as full node. A node, which stores only the header of the blocks is called as light node. The third type of node is called as bootstrap node, it is highly available node, node address is hard-coded in the software and it helps other nodes to connect to the network. Mostly these nodes are the blockchain application developer nodes.

1.1.3 Blockchain Types

Though blockchain has various features, there are applications not having sufficient infrastructure to incorporate. The blockchain introduced by Satoshi Nakamato may not fit in to all the applications. Considering the need of blockchain and applications' constraints, the following different types are introduced.

- **Public or Permissionless Blockchain**: This allows any individual or organization to join the network and participate in all blockchain activities such as mining, block/transaction validation, etc.

- **Permissioned Blockchain**: It allows only the authenticated and authorized users to perform read and write operation. The read operation allows a user to query and retrieve data from the blockchain whereas write operation allows to add data onto the blockchain.

  - **Public Permissioned Blockchain**: This allows only authenticated and authorized user to perform the write operation but any user to perform the read operation.
  - **Private Permissioned Blockchain**: It allows the authenticated and authorized user to perform both read and write operation.

1.1.4 Blockchain Features

Blockchain is preferred in various applications since it has the following key features [19].

- *Public Verifiability*: Blockchain allows anyone to verify the correctness of the state of the system that is data in a block to another block is updated correctly and accepted by majority of the nodes in the network. In a distributed ledger, each state transition is confirmed by verifiers however it can be a restricted set of participants. Any observer can verify that the state of the ledger was changed according to the protocol or not and all observers will eventually have the same view of the ledger, at least up to a certain length.

- *Transparency*: The data and the process of updating the state is visible to the public. However, the amount of information that is transparent to an observer, can differ according to the application requirement.

- *Integrity*: It ensures that information is protected from unauthorized modifications, i.e. that retrieved data is correct. The integrity of information is closely linked to public verifiability. If a system provides public verifiability, anyone can verify the integrity of the data.

- *Availability*: The data availability is important for many use cases. In blockchain systems, redundancy is inherently provided through replication across the nodes.

- *Trust Anchor*: It defines the highest authority of a given system that has power to grant and revoke read and write access to a system. This is applicable only to the permissioned blockchain.

1.2 Cryptography

Cryptography is a method to enable two or more users to communicate over an insecure channel in such a way that an opponent cannot understand what is being communicated. The terms that you hear a lot when we talk about cryptography are plaintext, or cleartext. The message that you want to encrypt is a plaintext, because if you read it, you get the meaning of it. The second thing is encryption or encipher. That is an action you apply

on the cleartext or the plaintext and you encode it in a form so that only people who have a very specific knowledge can actually recover the original plaintext or original message, others will see garbage. It is a very old idea we have, we know that Julius Caesar used Caesar cipher. So this is not something that is very modern. But in modern day, the technology, and the algorithms for encryption or enciphering has changed drastically from the old days. Then after the encryption, what we get, we call them ciphertext. It is basically the encrypted message. And in order to read the message, you have to do a decryption or decipher. And this basically converts the ciphertext to plaintext.

Now, if you look at this as some kind of a data flow as in figure 1.3, then M is your message or plaintext here. And this plaintext goes into an encryption algorithm or here it is shown as an encryption algorithm box. It could be hardware or software and then what comes out is the ciphertext C. The ciphertext is the encrypted version of the original message. And then you want to read it, then you have to pass it through the decryption or deciphering algorithm or the hardware and then comes out the cleartext or the original message. So the following identity must hold. If D applied to C is M, and M is C applied to E. Then M must be recoverable by a composition of D and E. That means, if you apply E first on the message, you get the ciphertext and then on the ciphertext, if you apply D, then you should get back the plaintext. That is the idea, that is the property of all encryption systems.

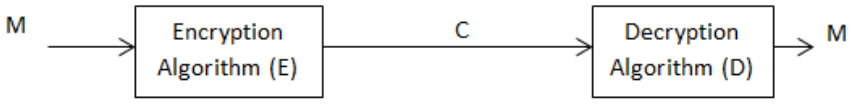

Figure 1.3: Encryption and Decryption

In cryptography, as we said that the method of encryption and decryption is called a cipher. So generally there are two related functions, as we showed in the figure, that there is one for encryption and one for decryption. And in the old days, how you encrypt and how you decrypt it were kept secret, right? That is the only secret that stood in the way of somebody an eavesdropper. Somebody who is trying to capture the message on its way and trying to

understand it, it is secret to them, and therefore, they cannot do it, unless somehow they figure out what the algorithm is. And this is what happened in the 1940s when the Germans used this Enigma code to instruct their naval ships and U-boats. And this messages were going encrypted.

And then the Britishers, which was a team led by Alan Turing, actually constructed a method and device that would reverse the process. And without knowing the algorithm that the Germans were using, they had to actually come up with the algorithm that is being used. And then they had to figure out how to reverse it. And that is how they actually encrypted the Enigma code. But nowadays, we do not keep the algorithm secret, because keeping an algorithm secret is quite difficult. So we have to use some other method to keep the secrecy and that is where the idea of a key comes in. A cryptography key is basically the secret that is an ingredient into the encryption process. And the key is also an ingredient in the decryption process. The encryption key and the decryption key, may be same or different.

Based on whether they are same or they are different we have two types of encryption or decryption method, which are called the symmetric encryption or asymmetric encryption.

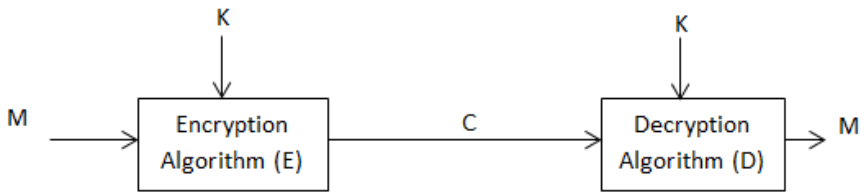

Figure 1.4: Encryption and Decryption

The figure 1.4 contains the second input that is K but it is missing in the figure 1.3. Now we have the encryption in encryption box which takes a message M and a key, in this case K and then outcomes, the ciphertext. When you put that ciphertext C to the decryption algorithm, then you need another key or may be the same key depending on whether it is a symmetric case or asymmetric case and then you get the M. This is called as secret key cryptography. The secret key cryptography means that the key has to be known and kept completely secret from outsiders. It means only the person encrypting it should know the key and the person who is supposed to decrypt it should know the key.

1.2.1 Symmetric Key Cryptography

Figure 1.5 shows the process of symmetric key cryptography with a single key k for both encryption and decryption. Both parties can encrypt and decrypt the message in symmetric key cryptosystem using the same key.

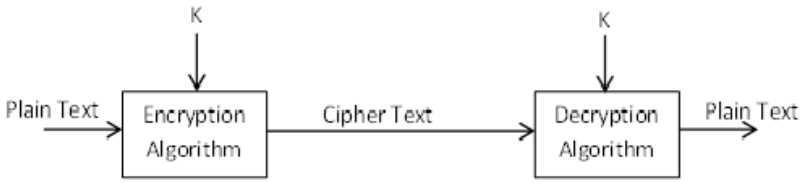

Figure 1.5: Symmetric Key Cryptography

The symmetric key algorithm is also called single key algorithm because you have only one key shared between the receiver and sender. And obviously, two keys are the same. And most of the time this encryption algorithms are also pretty simple. They are simple, but they are tedious. So you take the information that you want to encrypt.

And with the key using the key, you decide on certain substitutions and the part of the message or permutations of the bits or symbols of the message and you do this for many, many rounds. And how you do it depends on the key, the individual bits of the key. And therefore, when you want to decrypt it, you have to reverse the permutations and reverse the substitutions in order to get this message back.

And since you have the key, you know what the substitution and permutation sequences were, and by how much the permutations happened, by how much the substitutions happened. So similarly, you will do the opposite in the other side. One good thing about this is substitution and permutation are pretty easy to do in hardware.

Because you are basically taking bits and pushing them through shufflers and parameters and so on. And you do this many, many times. They are called rounds. And therefore, the secret key cryptography has the advantage of being very efficient and fast when you have a hardware support for doing cryptography. So there are chips that will do this for you in a very, very fast turnaround time. So now how you encrypt with the help the key is based on whether you are doing it continuously as you see bits of the message and you do something with the bits or you divide the message into blocks, chop

the message into multiple smaller messages, and then you apply the key as a whole on each of this chopped portions. So those are the depending on how you do it, whether you want to do it on the stream of bits or on the blocks of bits.

These are called block ciphers and stream ciphers and block ciphers usually divide the message into chunks of like 64 or 128 or 256 or 512 bit blocks. So as mentioned that the symmetric cryptography is very simple and very, very fast.

Examples of symmetric key cryptography are DES, like Data Encryption Standard RC4, IDEA and Blowfish, and most common nowadays is the AES, right. So so AES is the current most common standard and there are variants of AES, which are also used.

The properties of secret key or symmetric key algorithms are as follows.

- First of all, it assures *confidentiality*. So which means that if you have already exchanged the key, and nobody else has the key, then when you send the ciphertext on the way, if somebody reads your ciphertext by tapping on to your network, they would not be able to figure out what what is being sent because unless they have the key they cannot decipher. So the symmetric key cryptography basically achieves that privacy or confidentiality of information, provided the key is kept secret.

- Also if the key is kept secret, then *authentication* is ensured. That is, if you receive a message from somebody, you want to know that it is that person who suppose to send you the message is indeed sending you the message. And somebody is not pretending to be him or her and sending you the message. Now if you have done the key exchange beforehand with the authentic person, then when you receive the message and you apply your key for decryption, and you could and you can decrypt, then that means the anybody who has sent this message must have the key. Now assuming that the key has not been stolen from the sender who you actually exchange the key with, then the message has to be coming from an authentic source. So authenticity is also proven, provided the key is kept really secret between the two parties.

- *Message integrity* means that the message or the data has not been tampered with, nobody has changed the data while the data was in transit. So that is also assured because if somebody replace the message, and then he has to also encrypt the message with the key. Otherwise, you

would not be able to decipher it. When you try to decipher it with a key and it was actually encrypted with a different key then you will get back garbage. So therefore if you can decrypt the message, then it is actually showing that the message could not have been tampered with. Hence the message integrity is also proven if you can decipher the message.

- *Non-repudiation* means that the sender cannot later say, oh no, I did not send this message. Because if the sender has used the key, and he is the one who knows the key and nobody else, then he has to have sent it. Otherwise, there are two possibilities. One, the key is stolen or he is lying.

  Assuming the key is not stolen, then it is a non-repudiable. That is that the sender cannot admit to have sent the message, he cannot deny. So now let us look at non repudiation a little more closely by playing out a scenario. So let us say and in cryptography literature, you will always see these names Alice, Bob, Eve. So usually Alice and Bob are the sender receivers, and Eve is an eavesdropper. So who actually tries to capture the message while the message is on transit and tries to decipher. So Eve is usually the attacker, or threat. Let us see two scenarios.

  In scenario 1, suppose Alice sends a stock buy request to Bob, and Bob does not buy and claims to have never received the message. So at the end, Alice would suffer some loss because the stock might have gone up in price. And then Alice says to Bob that I was supposed to have bought this when the price was low, and I sent you message but you did not buy it. So I have suffered loss. Bob can say that no, no, I never got the message so I never bought that stock.So he can deny to have received the message.

  In scenario 2, suppose Alice sends a stock buy request to Bob. And Bob sends back an acknowledgement that yeah, I am going to do it, I received your request. And then again Bob fails to buy it. And then later when Alice comes and says I have suffered loss for you, because of you. Bob can say that no, no, I did not get the message. Then Alice shows him that look, you sent me an acknowledgement message. And therefore, you must have received the message. Now in case they are not using encryption, or any kind of method of non-repudiable

> communication, then Bob can say that well, it looks like a message from me. It has on its header that it is from my email address, but somebody fabricated it. And then without the use of cryptography for Alice, there is no way that she can prove that the acknowledgement message was indeed from Bob. Maybe somebody, like an Eve might have captured Alice's request in the in the middle of the transit, and then block the message to receive go to Bob. And she faked a message from Bob to Alice that okay fine, I am going to buy it, but she does not do it. So Bob is not the culprit.
>
> So in this case, Bob has repudiability. But in case of the whole thing was done cryptographically, then Bob could not have done repudiation unless he claim that my key was stolen, which is another issue outside the scope. So in a court of law, under the circumstances we described, Alice cannot prove that the acknowledgement message was indeed from Bob. So that is about non repudiation.

Usually one generates the key and shares with another through a secure channel. So that means this key is shared between the encryptor and decryptor. And this sharing is a big problem because how do you let the decryptor know the secret key?

So one possibility is that you do it out of band which means that you use some other means like a telephonic call or some other kinds of method in which you send the key to the receiver, then you send the message and then the receiver will use the key. But if you send the key with the message, then anybody who is listening on the communication channel will read the ciphertext, but then also read the key.

And therefore, they will use the key to decipher because algorithms are not secret, only the keys are secret. So therefore, there has to be a way in which the key is shared between the two parties. And that is one of the problems that the symmetric key cryptography suffers from. And since before you share a key, you cannot encrypt the key because then there is a problem that when you have the key, then you decipher it.

But if you do not have the key, then there has to be a way to share it. And most of the time, this has to done out of band. And which means that there has to be some other way of communicating between the sender and the receiver. So disadvantage, is that how do you exchange the key between the sender and the receiver. And if you want to keep the key in some kind of a, you know database or something, there is a possibility of the key being

stolen or read by somebody.

Because if you keep it in an encrypted form, then the you have to first give the everybody some kind of a key to decrypt them, right. So you have to keep it in plaintext, that is not safe. And therefore, key exchange and key management is a big problem in symmetric key cryptography. And that is where the asymmetric cryptography comes to help because the asymmetric cryptography is used to actually do the key exchange.

And once you have done the key exchange, you can apply symmetric key cryptography. Now you might ask, why should not I then only use asymmetric cryptography? The problem is asymmetric cryptography is not as efficient and it is time consuming. Therefore, you only use it very, very sparingly. And in most cases, it is used for key exchange and after the key exchange has happened you can then apply symmetric cryptography on very large amount of data, because it is fast.

The other problem is that if you have 2 parties, then you have 1 key but if you have 3 parties, then you need, you know $3C2$, which is 3 keys. If you have 4 parties $4C2$ six keys, and if you have 5 parties, then you have, you know $5C2$, which is 10 keys. So this way, the number of keys between parties will also increase very, very quickly in quadratic speed. And that is another problem that you know you have to maintain so many keys for every pair of communicators.

Because if you keep it in an encrypted form, then the you have to first give the everybody some kind of a key to decrypt them, right. So you have to keep it in plaintext, that is not safe. And therefore, key exchange and key management is a big problem in symmetric key cryptography. And that is where the asymmetric cryptography comes to help because the asymmetric cryptography is used to actually do the key exchange.

1.2.2 Asymmetric Key Cryptography

It uses one key for encryption and another for decryption process. Figure 1.6 shows the process of asymmetric or public key cryptography with two keys that is one to encrypt the plain text to cipher text and another to decrypt the cipher text to plain text. For example, the receiver *Bob* generates the key pair (K_{Pub} , K_{Pri}) and shares the public key K_{Pub} with sender *Alice*. *Alice* use the public key (K_{Pub}) of *Bob* for the encryption process and *Bob* uses his private key (K_{Pri}) for the decryption process. As the name implies, public key is for the public that is any one can get the key and encrypt the

required message but only the owner of the key can decrypt ciphered message using the private key known only to him. There are different asymmetric key algorithms available such as RSA, Elliptic Curve, Diffie Hellman Key exchange, etc. Asymmetric key cryptography is mostly for symmetric key exchange purpose.

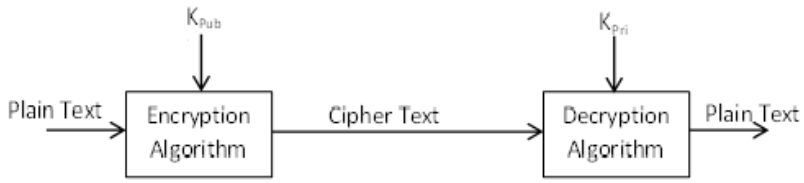

Figure 1.6: Asymmetric Key Cryptography

Therefore, public key cryptography is something that is almost essential in actually doing cryptography today. In practice, for example, when you communicate over the web, and you do ecommerce on the web, you use HTTPS, which uses Secure Socket Layer (SSL) and there the key exchange happens through public key cryptography. Once the key exchange happens it is called a session key.

And then that session key is used as a symmetric key between the two parties. And public key cryptography in the beginning, allows them to establish that secret key between the two parties.

So what is a public key cryptography as we said before? In public key cryptography, there are two keys. One is kept secret. The other one is kept public or known to everybody. And this pair of keys here we are calling it $K_{R(pub)}$ (public key) and $K_{R(pri)}$ (private key). Suppose Alice want to send Bob a message and Bob tell Alice his public key.

Bob can tell anybody his public key, and that time, what Alice do is encrypt with that public key, transcend Bob the ciphertext. Since Bob have the private key and nobody else has a private key corresponding to this public key, only Bob can decipher the message. And Bob not tell this key to anybody, keep it very closely secret. Therefore, the key, Bob has two keys or we can say it is a two part of a key system.

One part, the public part, Bob can announce to the world that if any of you want to send message to me, then send me encrypted with this key. And the way it is designed that only only person who can actually unencrypt the

message or decipher the message is one possessing the private key. Therefore, it has to be only Bob who can read it. So that is the idea of public key cryptography. Now the question is, how is it used to establish a shared secret, because public key cryptography is usually based on a very complex mathematical problem compared to secret key cryptography or symmetric key cryptography where bit shuffling and bit permutation, this kind of stuff is used. So it is very fast to implement. Whereas in this case, normally what we do is we try to solve a very complex mathematical problem, rather than a bit shuffling kind of thing, simple things. Therefore, it is not very fast. So we have to eventually use a symmetric key cryptography. But to establish the secret key between the two parties, we need to use public key cryptography. So that is what public key cryptography is normally used for.

So I am sending a message over the Internet to my friend. Here in the internet, there are lurking behind many of the computers, there are people who can capture my message. And then if they know the key, then they will decipher it. So I have to somehow make the system such that the symmetric key that I am going to use to communicate very large amount of messaging or file uploads and all that I will not let anybody else know the key. So public key cryptography helps me by the following.

If you use public key cryptography, then you never send the secret key over the network. So what you do is you establish the secret key by using public key cryptography without having to ever sent the key on the network. So everybody creates the key on their own computer. And because of the mathematical process that is followed, the key this party establishes and the key the other party establishes becomes the same. So that becomes a shared secret without having gone through the network. So the problem statement here would be that Alice has a channel for communicating with Bob and they want to use this channel to establish a shared secret. However, Eve is able to learn anything that is sent over the channel. So if Alice and Bob has no other channel to use, how can they establish a shared secret?

As discussed, public key cryptography is hard mathematical problem. So for example, RSA is dependent on the fact that if I give you a very large integer N and the N can be factored into two prime numbers, p and q, finding those p and q to factor N is a very hard problem. It is called the integer factorization problem and figuring this problem is not very fast and if N is a very, very large number like 2048 bits. It is a very large number, let us say. And then you have to know what are the two prime numbers that produced this number by being multiplied, it is a hard problem. And

this kind of hard problems are used for public key cryptography. We can call this as one way function because multiplying p and q to produce N is easy. And then if you do not tell people p and q, and just give the N for people to know the p and q is hard. So reversing the process is difficult. So public key cryptography as you know that the person who wants to send you the message, you basically send it encrypted with the public key. And then, when you decipher the message, you do it with the private key.

1.2.3 Hash Function

It is a function that takes arbitrary size input and produces the fixed size output. In equation 1.1, a function takes arbitrary size of string sequence with the combination of 0s and 1s and produces output of size n.

$$f\{0,1\}^* \rightarrow \{0,1\}^n \tag{1.1}$$

A hash function, especially cryptographic hash function should satisfy the following three properties

- Pre-Image Resistance: Hash function should be an one-way function that means it should be computationally hard to reverse a hash function. As given below, from the hash output H, it should be hard or infeasible to bring out the message M.

  $$f\{M\} \rightarrow H$$
  $$M \not\leftarrow f\{H\}$$

  Also, if a hash function f produced a hash value z, then it should be a hard to find any input value x that hashes to z.

- Second Pre-Image Resistance: Given an input and its hash output, it should be hard to find a different input with the same hash. For example, in the below, from input M_1 and its equivalent hash H_1, it should be hard to find message M_2 which will produce same hash output H_1.

  $$f\{M_1\} \rightarrow H_1$$
  $$f\{M_2\} \nrightarrow H_1$$

This property of hash function protects against an attacker who has an input value and its hash, and wants to substitute different value as legitimate value in place of original input value.

- Collision Resistance: It should be hard to find two different inputs of any length that result in the same hash. Difference between second pre-image resistance and this is: In the second pre-image resistance, message and hash value is fixed but in case of this property, attacker is free to choose any two inputs, which can bring out same hash value. If any hash function is collision resistant then it is second pre-image resistant also.

In any hash function, collision do exist because the output set is limited when the input set is high. For example, in figure 1.7, it shows that the input set is mapped to the output set. The size of input set is higher than the output set. Hence, there will always be a possibility that collision will occur. According to Birthday paradox, if the output set size is n with possible $0, 1$ combination then there is 50% chance that in $2^{\frac{n}{2}}$ input, collision will occur. However, it is computationally hard to achieve the collision because values in output space has the equal probability of occurrence.

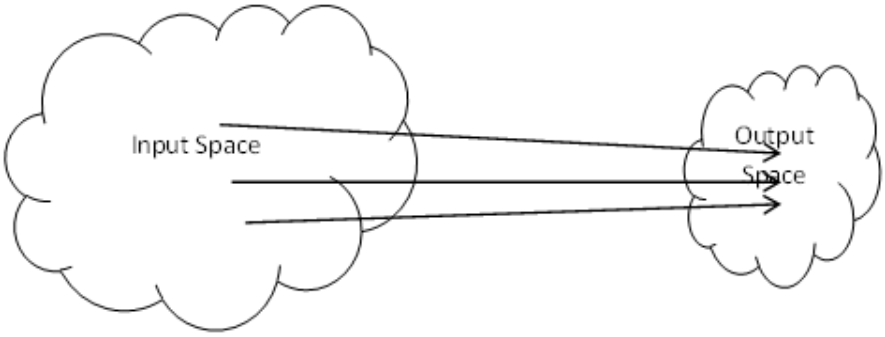

Figure 1.7: Hash function mapping

Verification The output of hash function h is to verify the integrity of the message. To verify the integrity, give the message and the hash output to a function V as in equation 1.2, which will identify whether message is valid

or not that is modified or not.

$$V(M,H) = \begin{cases} Valid & \text{If } h(M) == H \\ Invalid & otherwise \end{cases} \tag{1.2}$$

There are different cryptographic hashing algorithms available, commonly used are Secure Hash Algorithm (SHA) versions. Other hash algorithms are Message Digest (MD), RIPEMD, etc.

1.2.4 Digital Signature

Digital Signature is to verify authenticity of the digital message or documents. Through this, we can ensure that the message is sent or signed by the respective user and it is not tampered. Digital Signature uses asymmetric key cryptography and hash function for preparing signature and verification.

Digital signature is important because we always want to know in blockchain, whoever is doing transaction, the transaction before being recorded, one has to verify that it actually being done by somebody who claims that he is doing the transaction. So therefore, everybody has to have a public key and private key to be part of the blockchain ecosystem. So you have a public key and you have a private key. Everybody knows your public key if they want to know whereas private key or secret key is kept with you. So if you sign a transaction with your private key, digital signatures are basically some kind of an encryption done to prove that you know the private key. So anybody who can open decipher it with your public key knows that it is you who did this.

So whenever you put out a signature, then people would know that you did the transaction and if your were supposed to do the transaction, then that is a valid transaction. If you were not supposed to do the transaction, it is your signature, then the transaction will be considered invalid and will not go into the permanent blockchain.

How do the digital signatures work?

First, you have to have a function for generating keys. So you say that I wanted 256 bit key. So then you call the corresponding functions generate keys. It will give you a key pair, that is secret key or private key and public key. So then to sign a document, you take a message or a message digest, and then you encrypt it with your private key.

To verify that it is your signature, all I have to do is I have to check whether with your public key, I can decipher the message. And then if I can do that, then I know that this signature is valid. So there should be an is Valid function. So anybody who wants to do the create key pair has to use this function generate keys. Anybody who wants to sign has to use the secret key and use the sign function. And anybody who wants to check whether it is indeed the valid signature has to use the verify function.

So only valid signatures, the digital signature must have the property that only the valid signatures will verify accept others will fail to verify if I am saying I am signing but I am using somebody else's secret key. Hence no one is able to forge somebody's signature.

Unless my private key is stolen, no body can do my signature. So this is very important part of the blockchain technology that I should be able to have a verifiable digital signature and I have to keep my private key secret. Also I have to be able to apply the signature on a message or a message digest. Now one important thing that I must, we must remember is that the algorithms that we use for all cryptography like generating keys and, and so on, requires a good source of randomness.

Now we know that the creation of random numbers is very difficult. And many times the function that produces random numbers turns out to be not very random. And that has been a source of problem in many cryptographic techniques. And people have been able to decipher keys by using the fact that the randomness that you thought was random, was not very random, and they could know, they could reconstruct it.

So it is very important to have a good source of randomness. Without that the whole construction of cryptography digital signature will fail. Also when you sign, you can sign a message, or you can sign the digest of the message. Because digest of the message is always smaller than the message. So that makes your signature faster and you can also sign a hash.

The bitcoin blockchain uses the ECDSA that is the Elliptic Curve Digital Signature Algorithm. Because RSA uses the large integer factorization as its hard problem on which the whole construction is based. Now it is known that if there is Quantum Computing then factorization of large integers will be easy. And so in that sense, Bitcoin as well as other cryptocurrency implementations has made a good choice, making ECDSA as your signature standard for digital signature in bitcoin blockchain.

1.2.4.1 Elliptic Curve

An elliptic curve defined by $y^2 = x^3 + ax + b$ is a plane algebraic curve defined over finite field, where a and b are constants and x and y are variables that is co-ordinate of the curve. It supports two operational properties

- Point Addition : Addition of two points such as $(x_3, y_3) = (x_1, y_1) + (x_2, y_2)$
- Point Doubling : Multiply a point two or more times such as $[2](x_1, y_1)$, where the point (x_1, y_1) is multiplied twice.

$$P_1 + P_2 = P_3$$

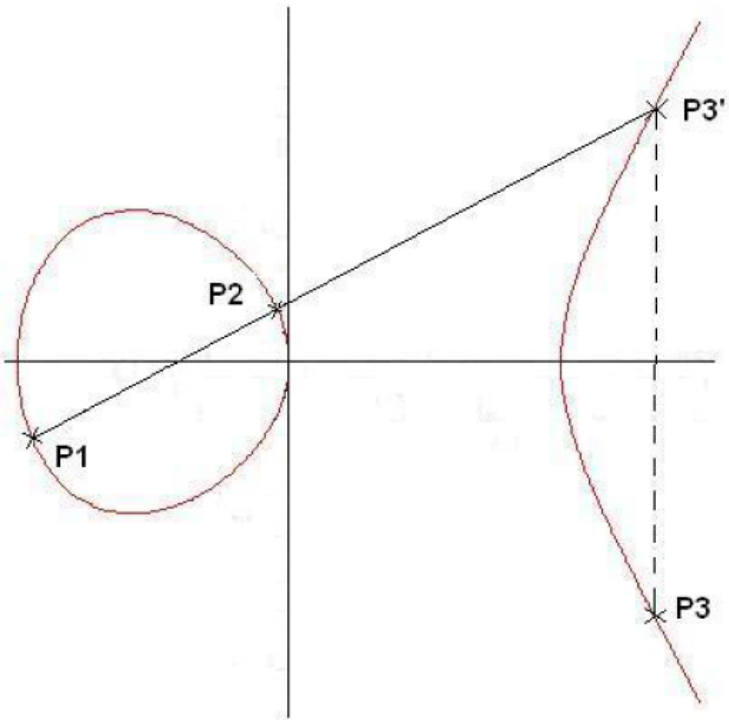

Figure 1.8: Point Addition

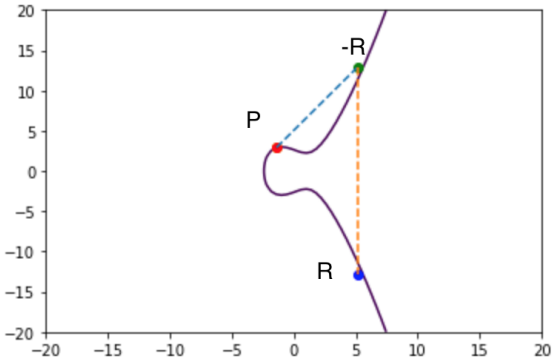

Figure 1.9: Point Doubling

Figure 1.8 and 1.9 shows elliptic curve with point addition and doubling respectively. The process of computing addition and doubling are shown in algorithm 1 and 2. The variables $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ are points on the curve E with the condition that $P_1, P_2 \neq \infty$. The variable m is to define the slope of the line on the curve.

Algorithm 1 Point Addition

Require: P_1 and P_2
Ensure: $P_3 = (x_3, y_3)$
  if $x_1 \neq x_2$ **then**
    $m = \frac{y_2 - y_1}{x_2 - x_1}$
    $x_3 = m^2 - x_1 - x_2$
    $y_3 = m(x_1 - x_3) - y_1$
  end if
  if $x_1 = x_2$ AND $y_1 \neq y_2$ **then**
    $P_1 + P_2 = \infty$
  end if

1.2.4.2 Elliptic Curve Digital Signature Algorithm (ECDSA)

This is used by Bitcoin and other cryptocurrencies to ensure that funds can only be spent by the rightful owners. The process of ECDSA needs the pre-sharing of the public parameters. For example, Alice wants to send a signed message/transaction to Bob. Initially, they must agree on the curve

Algorithm 2 Point Doubling

Require: P_1 and P_2
Ensure: $P_3 = (x_3, y_3)$
  if $P_1 \neq P_2$ AND $y_1 \neq 0$ **then**
    $m = \frac{3x_1^2+A}{2y_1}$
    $x_3 = m^2 - 2x_1$
    $y_3 = m(x_1 - x_3) - y_1$
  end if
  if $P_1 = P_2$ AND $y_1 = 0$ **then**
    $P_1 + P_2 = \infty$
  end if

parameters $(Curve, G, N)$, where G is the base point, N is the order of the Curve. *Alice* choose the Private Key d_A which is between $[1, N-1]$ and a public key curve point $Q_A = d_A XG$. Using these parameter, *Alice* computes the signature based on the steps given in algorithm 3 and *Bob* verifies the signature using algorithm 4.

Algorithm 3 Signing Algorithm

Calculate e = HASH(m) where HASH is a cryptographic hash function, such as SHA-2.
Let z be the L_n leftmost bits of e, where L_n is the bit length of the group order n.
Select a cryptographically secure random integer k from [1,n-1]
Calculate the curve point $(x_1, y_1) = k \times G$.
Calculate $r = x_1 mod n$. If $r = 0$, go back to step 3.
Calculate $s = k^{-1}(z + rd_A) \bmod n$. If s = 0, go back to step 3.
The signature is the pair (r, s).

It is mandatory to share the public curve parameters will all participating nodes to perform the signature and verification. In case of cryptocurrency, it is not so easy to share with all since there are massive number of participants. Hence, the curve parameters are hard coded in the implementation so that all participating nodes will have the same curve parameters.

Elliptic Curve Security: The security of the elliptic curve is based on elliptic curve discrete logarithmic problem (ECDLP). It means, if an adversary knows the point P and Q even then it is difficulty to find k, if $Q = kP$, where

Algorithm 4 Verification Algorithm

Verify that r and s are integers in [1,n-1]. If not, the signature is invalid.
Calculate e = HASH(m) where HASH is a cryptographic hash function, such as SHA-2.
Let z be the L_n leftmost bits of e.
Calculate $w = s^{-1} \bmod n$.
Calculate $u_1 = zw \bmod n$ and $u_2 = rw \bmod n$.
Calculate the curve point $(x_1, y_1) = u_1 \times G + u_2 \times Q_A(x_1, y_1) = u_1 \times G + u_2 \times Q_A$. If $(x_1, y_1) = O$ then the signature is invalid.
The signature is valid $r \equiv x_1 \pmod n$, invalid otherwise

k is private key. The size of the elliptic curve determines the difficulty of the problem.

1.2.5 Memory Hard Algorithm

In cryptography, a memory hard function (MHF) is a function that costs significant amount of memory to evaluate. A memory hard function f needs time T and space S for ordinary computation, but $T' >> T$ if $S' < S$. The intention of the memory hard algorithm is not to allow additional processor to any system without upgrading the memory.

1.2.5.1 ROMix

It is a memory hard algorithm with a hash function, input and other parameter. First part of ROMix shown in algorithm 1.2.5.1 that takes an input value X, generates the hash of it repeatedly for N iteration and stores in an array variable V_i. With input X and hash output $(V_0, V_1, V_2, ..V_{N-1})$, the next part of the ROMix shown in algorithm 6 randomly access the array values V to ensure that they are all stored in Random Access Memory. The input value X is given to a bijection function *integrify*. The random output j of the *integrify* function is index to locate the value V_j. The hash value of the input (X) along with V_j will be computed and assigned to X. This process will go for N iterations and final mix output is B.

This function ensures that the required memory S need to be allotted so that the mix generation will be fast that is the output can be computed in time T. In case the required memory is not allotted or less memory S' is

Algorithm 5 ROMix - Chain Hash computation

Require: Input X, N
Ensure: Hash Array V
  for i = 1 to N **do**
    $V_i \leftarrow X$
    $X \leftarrow H(X)$
  end for

Algorithm 6 ROMix - Random mix generation

Require: Input X, V, N
Ensure: Mix output B
  for i = 1 to N **do**
    $j \leftarrow integrify(X)\ mod\ N$
    $X \leftarrow H(X \oplus V_j)$
  end for
  $B \leftarrow X$
  return B

allotted ($S' < S$) and the hash values produced by algorithm are not stored then output can be computed in time T' where $T' >> T$.

1.2.5.2 HashCash

This is the first Proof of Work (PoW) system to control the email spam and it was originaly proposed by Cynthia Dwork and Moni Naor in 1992 [8]. Later it was named by Adam Back [5]. Hashcash is a non-interactive, publicly auditable and trapdoor-free cost function with unbounded probalistic cost.

- Non-Interactive: Client generates its own challenge instead of server sends the challenge to the client because there is nochannel for the server to send a challenge.
- Publicly Auditable: A cost function can be efficiently verified by any third party without access to any trapdoor or secret information.
- Unbounded Probablistice Cost: In general client take forever to compute, though the probability of taking significantly longer than expected decreases rapidly to Zero. For example, a user throw a fair coin

k number of times but not able to get the head. The throws k does not have any bound that is it can go upto ∞. The bounded one has the limit on which there is an upper bound for example choosing a key from key space.

- Trapdoor free: The fastest algorithm for computing partial collisions is brute force. There is no challenge as the client can safely choose his own random challenge, and so the hashcash cost-function is a trapdoor-free.

In Hashcash, the problem is finding the partial hash collision that is the left k bits out of full length (l) should be 0 ($\{0\}^k\{0,1\}^{l-k}$). This can be achieved only through the brute force. The hashcash is used in bitcoin a the mining function.

1.2.5.3 ASIC Resistance

It is an Application Specific Integrated Circuit that can be connected as the external processor to quickly complete the process. The cost of the ASIC devices are cheaper while compared with CPU. The miners involved in cryptocurrency use such devices to increase the speed of computing the desired hash. For example, in Proof of Work (PoW), miner needs to generate a hash value that is less than the target to successfully mine the block. If ASIC is used along with regular CPU then the respective hash value can be computed fast when compared with attempting only on single CPU. Malicious user can add more ASIC devices and mine the blocks consecutively to sabotage the system. The ethereum cryptocurrency uses memory hard algorithm to resist the ASIC devices to participate in the hash generation.

1.2.5.4 New Identity

The participating nodes of the cryptocurrency should have to identity to participate in the blockchain activities such as transaction, mining, etc. The identity is derived from the key pair of the user. The public key pk becomes your public name and sk is kept as your hidden identity or key to your identity. So if you want to prove your identity you have to use sk to sign. Now if pk looks random, nobody needs to know who you are. So therefore, one of the big problem in blockchain is that a person can create hundreds of different identities. For example, Alice can create as number as many as key pairs, and can use that to create multiple identities and do various things

in under the various identities. And therefore people would not be able to know that Alice is the person who is doing all this.

Therefore, anybody can make any new identity at any time and make as many as they want. And there is no central point of coordination. And that was by design, because the anonymity of users was important for the originator of the bitcoin or other cryptocurrencies blockchain. And these identities are also called addresses in bitcoin. Because anybody who sends money to Alice, he sends it to Alice public key and that is why it is called an address. And it is not connected to Alice real world identity. Nobody is checking. There is no other linking or anything. So therefore, you cannot discover ever who is Alice. However, there may be some investigative work people can do. And if you are not clever enough, then by linking together the various addresses that you are operating, how what activity you are doing, and you are transferring money back to one particular identity and so on, they can somehow infer certain things.

And then once they know one of the identity, one of the public key's identity, real world identity, then all the other ones that are also created by you and somehow they establish the link between all this, they can make inference. So in that sense, bitcoin blockchain is not fully anonymous. It is called pseudo anonymous. The researchers have shown that you can actually do various kinds of data mining to see which of the addresses are probably being maintained and operated by the same entity and things like that. But even then, discovering the real identity will require external investigative work that information cannot be found in the blockchain itself.

1.2.6 Proof of Work

Proof of work (PoW) is a form of cryptographic proof in which one party (the prover) proves to others (the verifiers) that a certain amount of computational effort has been expended for some purpose. Verifiers can subsequently confirm this expenditure with minimal effort on their part. The concept was invented by Cynthia Dwork and Moni Naor in 1992 as a way to deter denial-of-service attacks and other service abuses such as spam on a network by requiring some work from a service requester, usually meaning processing time by a computer. The term "proof of work" was first coined and formalized in a 1999 paper by Markus Jakobsson and Ari Juels [8] [10]. Later, PoW is popularized by Bitcoin as a foundation for consensus in permissionless blockchains and cryptocurrencies, in which miners compete against each

other to confirm transactions, append blocks and mint new currency.

So what is happening with proof of work? the proof of work basically allows you to select nodes according to the proportion of computing power they have for the entire infrastructure. So to prove that they have that computing power they have to compete for right to create the block and it should be moderately hard to create new identities to gain more computing power. For example, you you can create 1000s of ID's but then that computational resources will be used by all of them and therefore, total computational resources you have will be the same so, that basically, focuses us on this on avoiding the Sybil attack. So let us see how it is really done.

Remember random hashing, we say that if you have a fixed random input r, and you challenge others to give you an x, so that hash function as in equation 1.3 that takes random number r concatenated with x as input produces the output that belong to a set y where y is the target set. In blockchain, you take the block that means take the previous hash and all the transactions in that block, and then you concatenate x to find the output that belong to a set y. This is called as hash puzzle that is finding an x.

$$h(r|x) \leq y \tag{1.3}$$

The target set y is from the set of possible hash values. Let us say we are using 256 bit hash values, then our output space is 2^{256} and that is very large. The target set of the hash value is basically less than, let us say, so many leading zeros followed by any numbers. So which means; that you are saying that not any number from 2^{256} possible numbers can be selected. Only the ones with so many leading zeros, which mean relatively smaller numbers, any x that will satisfy this will be winning for the hash puzzle. The hash puzzle has this property that we discussed earlier, is that the only way to solve this is through brute force. You start by saying that, x = 0000 will be this. And then when you try to hash if the hash does not come out, to be in that range, then you try 000001 and so on. And then you have to, you may have to try all of them.

Now, many miners let us say there are 3 miners each having 30% of the resources. So there is a probability that all three will solve them almost at the same time. And then when they solve it, they think that they have won. Because, there is no central authority to say that you are the winner. So, all three will have their blocks as competing blocks for being added to the blockchain. So therefore, there will be a race condition.

So, now everybody's competing hash so, the every second now, I looked this up and in December 2019, as you can see, that over almost like 110*1012 hashes per second. So, this is Tera hashes. So, that many hashes are computed per second by the blockchain ecosystem. Now, all these hashes are not computed by single node all nodes are trying. So, this number of hashes per second is actually the total cumulative number of hashes computed.

And some of them will be doing much less number of fascists and win. The point here is that the proof of work is a very difficult brute force computation and there is a lot of computation that goes on at every mining node that is trying to solve this hash puzzle. Now, the problem is that in the beginning, everybody was using desktop computers and then the hash rate was much less as you can see even within like this is from January 19 to December 19. In January 19, the hash rate was 40 million, tera hashes and so if you look at this, is 44 million. Here we are looking at almost 110 million. So, within a span of a year the hash rate has increased. This means that people are throwing in a lot more resources, a lot more parallel computation, a lot more GPUs and so on. To do this hash computations and therefore, we are seeing a surge in the hash rate of the entire network. So, therefore, what will certainly happen is that in the beginning, the let us say I keep I give you a hash puzzle, and you can solve on an average within 10 minutes, then after you throw in more computational power.

You can try parallely more many more hash combinations, many more nonce x combinations, and therefore, you will be solving it faster. And then more computation you give more computational resources you give you can compute even more efficiently. So therefore, what happens is that periodically the nodes automatically recalculate the targets set. If puzzle is solved quickly then the target set will be reduced otherwise increased to an restricted extend according to the blockchain implementation.

1.3 Distributed System

It is a system whose components are located in multiple nodes of the network and these components passes message between one another to achieve the common goal. The advantage of distributed system is improve the efficiency and performance. Distributed database is also a distributed system and blockchain is a distributed database. Distributed database have different storage devices, which will be controlled by different processors. It may

be in different geographical location or at one location but in different systems. Distributed database can increase the data processing speed since it supplies data in parallel and available near to the site of greater demand client. It also ensures the reliability and availability of data on demand. If any one storage device fails, data can be accessed from other storage devices. All storage will have the identical data or part of the data based on the requirement of end user. Distributed database can be of Homogeneous or Hetrogeneous. In Homogeneous, all storage device will use the same software such as operating system, data structure and database management system. In case of heterogenous, each storage device will its own software and storage schema. Blockchain uses homogeneous system. Using the following methods distributed database [4] can be configured.

- Data Replication: All storage will have exactly the same copy of data. User can access data from any of the storages. Update of data can happen in
  - Synchronous: The data changed at one storage device will get updated immediately on all other storage devices.
  - Asynchronous: The data will get updated for every period of time not immediately.
- Vertical Partitioning: The storage devices will have different data which are vertically partitioned. For example, data of one field or attribute will be in one storage and another attribute in another storage device.
- Horizontal Partition: The storage devices will have different data which are horizontally partitioned. For example, record of User $\mathcal{A}$ will be in one storage and User $\mathcal{B}$'s record in another storage.
- Combinations: Some data may be stored centrally and others may be replicated.

However, blockchain has to be configured with data replication. Blockchain uses the synchronous method of data update with bounded delay. Integrity of the data is a major issue in the distributed database especially while replicating dynamic data. In static data, no update of data allowed and it is fixed. In case of dynamic data, it keeps on changing based on the application requirement. In dynamic environment, one storage update its data and

submits to other storage for their update. Now, the issue arise that how other storage devices knows that the storage device is sending the original update not fake. Also, how one storage can ensure that the same data is sent to other storage. This makes the storage to resist in update of data. Such problem occurs in all distributed systems.

1.3.1 Decentralization

The first thing that we have to understand is the question of centralization versus decentralization. For example let us say transactions of your bank account are kept in a database. Even you can also have one replica of the database for fault tolerance or for backup, but the point is that the entire data is stored in a central location or in within the authority of a central, trusted body, like your bank. This obviously has its own problem first of all, centralizing everything gives you this notion of, vulnerability in the sense that if that particular data is somehow lost by a cyber attack or some other way, then you are going to lose all the information. Second is that the central authority has full command over what can be done on the data. And you have to somehow trust that central authority not to manipulate the data or delete some entries or do something like that.

Also, let us take another example, Know Your Customer (KYC) data with a central authority. Now, if the central authority maintaining the database decides to delete your entry, then you no longer have any access, nobody will be able to access your data and therefore, you will be not recognized through the KYC process. Now, unless there is a alternative way of doing KYC, you will be in trouble. That is the problem of the centralization, too much power in the hands of a central authority.

Therefore, Satoshi Nakamoto he or she or them, worried about these centralization of monetary infrastructure and monetary information in the hands of few banks and we should actually create a currency that is completely decentralized. So, there is no central authority which creates that money or a central authority, which, keeps the keeps track of the money or authenticate people to use their money. So that is the basic, so, you know, political economic underpinning of the introduction of the Bitcoin blockchain.

So centralization and decentralization versus decentralization is a basic concept that underlies the blockchain technology. Now, centralization has many advantages.

For example, it is easy to manage. A bank will not have to worry about,

you know, having replicas and privacy issues. Because if you have many replicas, you have to make sure that the data is kept private, all that kind of stuff. It is easy to provision. So for example, if you want to create a new account, you have to just make change in the central database. If you want to delete an account or then also you have to change in one place, it is easy to ban.

Similarly, if you have a centralized authority, maintaining for example, your DNS domain name service, then you can easily ban certain domains from being found, and it is easy to distribute responsibility. So centralization has all these advantages.

However, decentralization has some of these disadvantages, for example, it is harder to manage if data is distributed all over the place. It is harder to distribute work in the sense that you cannot say, you know, you have this responsibility, and you have that responsibility, because nobody is being commanded by a central server or central authority. And it is harder to ban something so you cannot delete information so easily. And it is harder to provision.

But centralization, as a single trusted party, is its biggest weakness in terms of not only the fact that they might actually do things in a unilateral way, which is not good for the consumers. But also the fact that if it gets cyber attack or any kind of attack, then also it can actually have a devastating consequences. Whereas if the data is decentralized, and all the functions are decentralised then it is much harder for an attack or for a central authority to you know take over this functionalities or change the functionalities and so on. So, therefore, blockchain is based on the notion of trust.

So, the question is when you have a central authority, you have to trust that authority and normally in the past we have been doing that. So, we have been trusting the authority that is managing the DNS, we are trusting the authority that manages the digital certificates, we are trusting the authority that is managing our accounts. But at present, we are seeing that a lot of the times the central authority becomes overly authoritative, they ban things. They actually can change things that to put you into trouble so therefore, the trust has to be created from a decentralized system.

Now, in a decentralized system when you have many players and no player is more important than the other players, which is the case in the centralized situation, there, creating trust is also a challenging task. Because you may have a number of players in the system a number of actors in the system, who are also malicious. So, you have to assume that in the ecosystem that

some who are actually together doing the computation, keeping track of data, deciding what is valid and what is not valid are malicious and so, therefore, your system should be designed in such a way so that in spite of that you have the ability to trust the system.

So you have to derive trust from untrusted participants or an untrusted actors for that you may have to make certain assumptions. For example, what percentage of people are untrustworthy? And what percentage of people are trustworthy if that assumption sometimes have to be met in order to trust the entire system. If you did not trust anybody, 100% of the people, then you cannot build a trusted system out of 100% untrusted actors. Let us see how bitcoin's blockchain bring trust and how do we derive trust in the system.

How is blockchain decentralized? The blockchain cryptocurrency applications maintain the ledger. The ledger is basically the transaction records. In blockchain, we do not maintain the ledger in a central only one location. It is replicated among 1000s of different computers and different what we call nodes. Now who has authority over which transactions are valid?

So, in a regular banking situation, you see that the transaction validity is checked by the bank. In case of blockchain, actually, every participant who is interested in data mining or keeping the ledger is going to check the transaction validity. So, if somebody wants to act maliciously, but others will not necessarily act maliciously, at least, the majority will not act maliciously we assume. And therefore, the transaction validity should be within the logic that we have come up with. Now who creates new bitcoins?

So we know in a banking system, the new currencies created by the central authority like Reserve Bank of India, or Federal Reserve in the case of in the United States, but in the bitcoin blockchain, the bitcoins are not created by any central authority. Here whoever actually wins the competition to create the new block. And if that block is actually accepted as the new block, then they get some reward.

And that is the only way to create a new bitcoins. So therefore, it could be anybody who wins and not every time same person wins. Therefore, bitcoins are created by many, participants. Now, who determines the rules of the system change? So bitcoin has now certain system rules that everybody that is every participant is supposed to follow. So that is how the program they write to run the mining process or check for transaction validity.

Now, how do you change the rule? The question is that you might want some rule change because you found some deficiency in the system. And that is when the rule changes. When rule changes the bitcoin current part

of the blockchain that so far you have developed may become obsolete. So, you may have to fork the blockchain. So, that is also not done centrally because this forking process takes place kind of automatically through the entire dynamics of the system.

1.3.2 Consensus

The consensus protocol is required for agreement among a number of processes (or agents) for a single data value. Some of the processes (agents) may fail or be unreliable in other ways, so consensus protocols must be fault tolerant or resilient. A consensus protocol tolerating failures must satisfy the following properties.

- Termination - Eventually, every correct process decides some value.
- Integrity - If all the correct processes proposed the same value v then any correct process must decide v.
- Agreement - Every correct process must agree on the same value.

The problem of reaching agreement in the distributed system can easily be understood from the Byzantine Generals Problem.

1.3.3 Byzantine Generals Problem

The problem of byzantine fault is expressed abstractly as the Byzantine Generals Problem [11]. A Byzantine army having multiple divisions with its commanding general are camped outside an enemy city. The generals can communicate with one another only by messenger to decide upon a common plan of action. There are chances that one or more generals may be traitors, trying to prevent the loyal generals from reaching agreement. The generals should ensure the following using a method to reach an agreement.

- All loyal generals decide upon the same plan of action: This is possible only when every loyal general obtain the same information.
- A small number of traitors cannot cause the loyal generals to adopt a bad plan.

Lets consider a byzantine army problem which is impossible to recover from failure when we use messenger messages. The army has a commander with two units headed by the lieutenants as shown in figure 1.10 plan for a battle . The commander can give command to units lieutenant either attack or retreat and to win the battle both unit should go for attack. In figure 1.10, commander gives the *attack* command to both lieutenant but lieutenant 2 is giving message to lieutenant 1 that he received *retreat* command. This will happen when lieutenant 2 is compromised. In such case no collective decision possible. In figure 1.11, commander gives two different command to the lieutenants that again lead to disagreement. The important notion to notice here is that lieutenant 1 cannot identify who is compromised and take agreed decision to proceed.

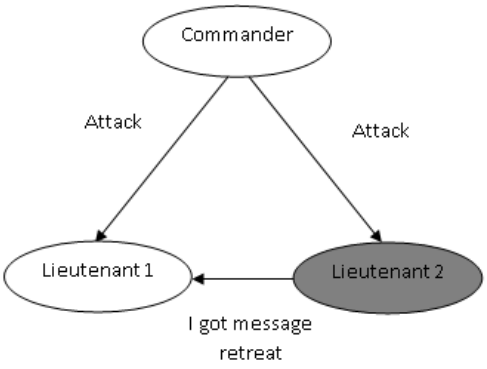

Figure 1.10: Byzantine problem - Malicious Lieutenant

In both scenario, lieutenant 1 have two messages that is one attack and one retreat. Hence, no decision can be taken since there is no majority. The same problem is possible when there is no message from the participating Lieutenant.

1.3.4 Faults

The distributed system have multiple components and there are chance that some component will fail or faulty. A fault can be crash or byzantine. The crash fault occurs due to the device failure. The Crash fault tolerance (CFT) is one level of resiliency, where the system can still correctly reach consensus if components fail for example Raft consensus protocol. The byzantine failure exhibit an arbitrary behavior such as sending conflicting information

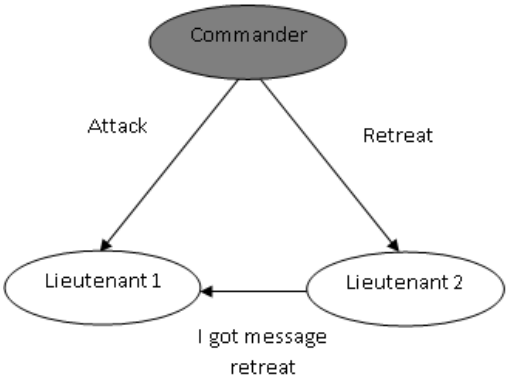

Figure 1.11: Bitcoin problem - Malicious Commander

to different parts of the system. Byzantine fault tolerance (BFT) is more complex and deals with systems that may have malicious actors.

1.3.5 RAFT

RAFT (Raft) is a consensus algorithm developed by Diego Ongaro and John Ousterhout in 2014 [13]. The nodes part of the RAFT consensus protocol can be any one of the following

- Leader - After getting majority in the election, the node become leader.
- Follower - In the beginning all nodes are the follower.
- Candidate - Node can announce itself as the candidate for the election to become leader.

In RAFT protocol, the client can communicate only with the Leader in case any client sends request to the follower node then it will redirected to the leader node. The RAFT protocol divides time into small terms and each term is identified by a increasing number, called term number. Every node part of the protocol will maintain the term number and it will be passed while communications between nodes. Every term starts with an election to determine the new leader. The candidates request for votes from other follower nodes to gather majority and the candidate node secured majority becomes the leader for the current term. In RAFT protocol, nodes does following two communications [9].

- *Request Vote*: This request will be sent by the Candidate nodes to get votes during an election

- *Append Entries*: It will be sent by the Leader node for replicating the log entries and also as a heartbeat by any nodes to check if a server is live. If there will be reply for the heartbeat then the server is live otherwise down.

1.3.5.1 Leader Election

A leader election takes place when a Follower node sends heartbeat to leader but it times out. In such case, the Follower node changes it state to Candidate state, votes for itself and sends *Request Vote* to get majority and attempt to become the Leader. If it gets the majority then it will be changed to Leader node otherwise it will turn back to the Follower node. In two cases, a node can vote for the Candidate node.

- A node can vote for the Candidate node only when term number of the Candidate node is greater than other Candidate nodes in the cluster and the same Candidate node will be elected as Leader node since all other nodes in the cluster follow the same protocol.

- The *Request Vote* also contains information about the candidate's log to figure out which one is the latest. If the Candidate requesting the vote has less updated data than the Follower then teh Follower does not vote for the said candidate otherwise it may give the vote.

1.3.5.2 Log Replication

The client request will be stored in the log of the leader and then forwarded to the Follower nodes of the cluster or network. The log contains the following information :

- Command specified by the client to execute

- Index of the log.

- Term Number at the time of entry of the command.

The Leader node sends the logs to all Follower nodes using the *Append Entries* to synchronous their logs with the current Leader. Once the majority of the Followers in the cluster or network successfully copy the new entries in their logs, it is considered committed and the Leader also commits the entry in its log to show that it has been successfully replicated. Later, the leader executes the client request and responds back with the result. The client entries will be executed in the order they are received.

Leader Crash: There is a possibility that Leader node crash in between the term. This will lead to inconsistency in the log entries of the Follower nodes. The newly elected Leader will match index number in the Leader and Follower, Follower then overwrite with the new entries supplied by the Leader to match the Follower with the Leader.

1.3.6 Byzantine Fault Tolerance

We will discuss two methods to solve the byzantine fault tolerance: one is Oral message based solution and second is signature based solution.

1.3.6.1 Oral Message based solution

: This method simply sends an oral message through a messenger with following assumptions.

- Every message that sent is delivered correctly.
- The receiver of a message knows who sent it.
- The absence of a message can be detected.

We assume that there are $n = 3m+1$ generals, where m are traitors. Here "oral message" means each general is supposed to execute some algorithm that involves sending messages to the other generals, and we assume that a loyal general correctly executes his process. The processes are given in the algorithms 7 and 8. In algorithm 7, the commander sends his value (target or retreat) to the lieutenants and lieutenant uses it. In algorithm 8, the commander send his value to lieutenants, later each lieutenants act as a commander and send the received value to other lieutenants. The algorithm 8 will be called recursively till m becomes 0. Finally the decision will be taken based on the majority value received. Figure 1.12 shows an example

of $m = 1$. In the beginning, commander sends message x to lieutenants later each lieutenant become commander and sent the received message x to other lieutenants. But, the traitor lieutenant 3 sends message y to other lieutenants as received message. Now for the decision, lieutenant 1 will see the received messages $(v_c = x, v_2 = x, v_3 = y)$ and majority will be the decision and similarly lieutenant 2 will take the decision based on the received messages $(v_c = x, v_2 = x, v_3 = y)$. As per the majority concept, lieutenant 1 and 2 will follow the message x and traitor message y will be discarded. While sending the message, lieutenants should index the message with sender identity to remove the ambiguity. This can happen only when m is more than 1. Importantly, this solution does not work when three generals in the presence of a single traitor.

Algorithm 7 OM(0), Oral Message

1: The commander sends his value to every lieutenant.
2: Each lieutenant uses the value he receives from the commander, or uses the value RETREAT if he receives no value.

Algorithm 8 OM(m), when m > 0

The commander sends his value to every lieutenant.
for each i **do**
  Let v_i be the value Lieutenant i receives from the commander, or else be RETREAT if he receives no value.
  Lieutenant i acts as the commander in Algorithm OM(m - 1) to send the value v_i to each of the $n - 2$ other lieutenants.
end for
for each i **do**
  Each $j \neq i$, let v_j be the value Lieutenant i received from Lieutenant j in step (2) (using Algorithm OM(m - 1)), or else RETREAT if he received no such value.
  Lieutenant i uses the value majority $(v_1, v_2.....v_{n-1})$.
end for

There are two natural choices for the value of majority$(v_1, v_2..., v_{n-1})$:

- The majority value among the v_i if it exists, otherwise the value RETREAT

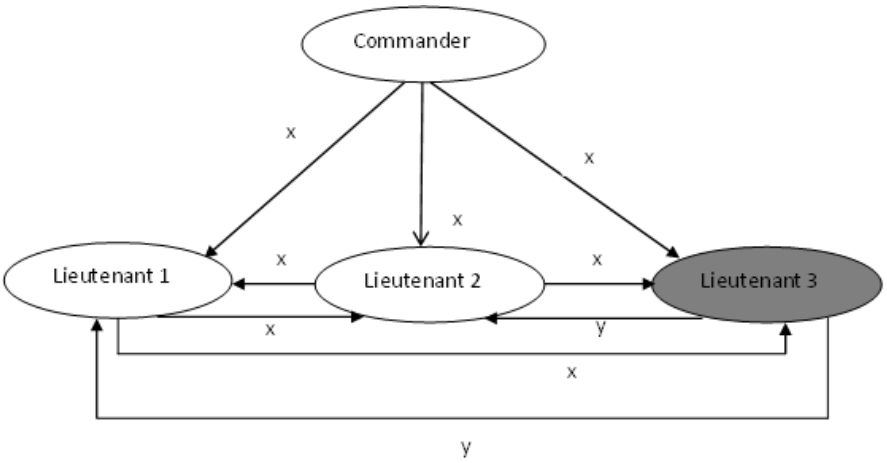

Figure 1.12: BFT Process of $OM(1)$

- The median of the v_i, assuming that they come from an ordered set

1.3.6.2 Signed Messages based solution

All generals should send the messages with their signature. In addition to assumptions in oral message based solution, following assumptions are also added.

- A loyal general's signature cannot be forged, and any alteration of the contents of his signed messages can be detected.
- Anyone can verify the authenticity of a general's signature.

It is important to note that no assumptions about a traitorous general's signature. In particular, his signature can be forged by another traitor, thereby permitting collusion among the traitors. In our previous solution, four generals are required to cope with one traitor however it no longer holds in the signature based solution. In fact, a three-general solution which was not possible in oral message based solution does exist here. We now give an algorithm that copes with m traitors for any number of generals. This will be useless if there are fewer than $m + 2$ generals. Algorithm 9 shows the process of signature based byzantine fault tolerance solution.

In algorithm, the commander sends a signed message to each of his lieutenants. Each lieutenant then adds their signature to that message and sends

it to the other lieutenants, who add their signatures and send it to others, and so on. This means that a lieutenant must effectively receive one signed message, make several copies of it, and sign and send those copies. It does not matter how these copies are obtained; a single message might be copied and sent, or each message might consist of a stack of identical messages which are signed and distributed as required. Algorithm assumes a function *choice*, which is applied to a set of orders to obtain a single one. The only requirements we make for this function are

- If the set V consists of the single element v, then choice$(V) = v$.
- Choice$(\emptyset)$ = RETREAT, where $\emptyset$ is the empty set.

Algorithm 9 SM(m), Signed Message

Initially $V_i = \emptyset$
The commander signs and sends his value to every lieutenant.
for each i **do**
  if Lieutenant i receives a message of the form v: 0 from the commander and he has not yet received any order **then**
    He lets V_i equal $[v]$;
    He sends the message $v : 0 : i$ to every other lieutenant.
  end if
  if Lieutenant i receives a message of the form $v : 0 : j_l : .. : J_k$ and v is not in the set V_i, **then**
    He adds v to V_i;
    if $k < m$, then he sends the message $v : 0 : j_l : .. : j_k : i$ to every lieutenant other than j_l.....j_k.
  end if
end for
for each i **do**
  When Lieutenant i will receive no more messages, he obeys the order choice(V_i).
end for

In algorithm 9, $x : i$ denote the value x signed by commander i and $v : j : i$ denotes the value v signed by j, and then that value $v : j$ signed by commander i. At first, commander signs and sends his value to every lieutenant. Lieutenant i receive the message v, include in the set V, if it

is not there already and send the received message to others lieutenants after signing it. In case, Lieutenant i receive the message v of the form $v : 0 : j_l : .. : J_k$ and v is not in the set V then add it. Also Lieutenant i further circulate the message by adding its signature if $k < m$. After the time out or indication of no message, Lieutenant i will pick the choice from the set V. Figure 1.13 shows the process of signature based solution with three generals. Even though, two choices are possible for each lieutenants they will obey the order received by commander but lieutenants can identify the traitor that is commander, who has signed for conflict messages.

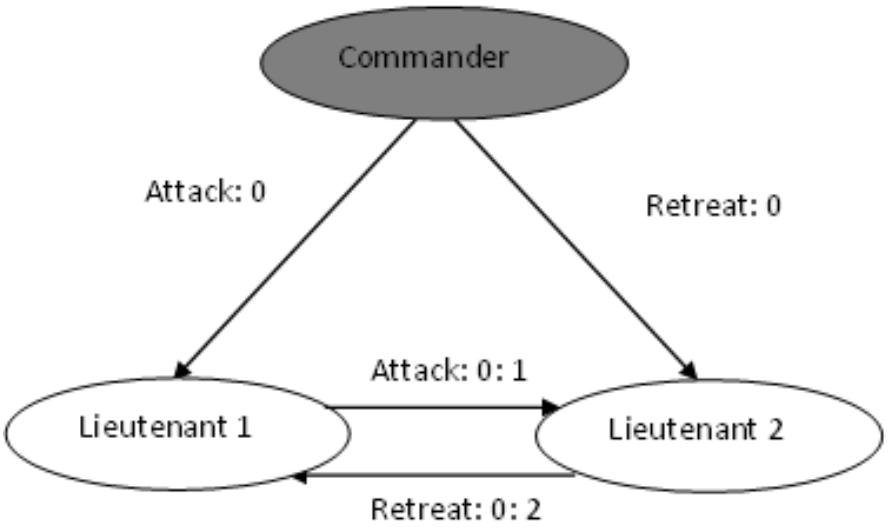

Figure 1.13: BFT Process of $SM(1)$

1.3.7 Practical Byzantine Fault Tolerance

The Practical Byzantine Faulty Tolerance (pBFT) is the real time implementation of the Byzantine Fault Tolerance. It uses practical Byzantine state machine replication that tolerates Byzantine faults (malicious nodes) through an assumption that there are independent node failures and manipulated messages propagated by specific, independent nodes [7]. The state machine replication means having the same state machine with pre-defined process in all participating nodes. The result must be deterministic provided execution of operation in a given state with a given set of arguments and

they must start in the same state. The pBFT will work for asynchronous distributed system with bounded delay, duplicate or deliver messages out of order. The pBFT designed to work in asynchronous systems with following steps.

- A client sends a request to invoke a service operation to the primary.
- The primary multicasts the request to the backups.
- Replicas execute the request and send a reply to the client.
- The client waits for 1 reply from different replicas with the same result; this is the result of the operation.

A client c requests the execution of state machine operation o by sending a $< REQUEST, o, t, c > \sigma_c$ message to the primary, which is chosen from the replicas. In the message, $REQUEST$ is to indicate the message is request type, t is the timestamp to ensure exactly once the execution of client requests and σ_c refers the message is signed using client key. The primary automatically multicasts the request to all the replicas. A replica sends the reply to the request directly to the client. The reply has the form $< REPLY, v, t, c, i, r >$ where v is the current view number, t is the time stamp of the corresponding request, i is the replica number, and r is the result of executing the requested operation. The client waits for $f + 1$ replies with valid signatures from different replicas,and with the same t and r, before accepting the result. This ensures that the result is valid, since at most f replicas can be faulty. If the client does not receive replies soon enough, it broadcasts the request to all replicas. If the request has already been processed, the replicas simply re-send the reply; replicas remember the last reply message they sent to each client. Otherwise, if the replica is not the primary, it relays the request to the primary. If the primary does not multicast the request to the group, it will eventually be suspected to be faulty by enough replicas to cause a view change, which is to change the primary. In a view change process, replicas suspects primary will send view change message to all other replicas with the state that which requests have not been committed yet. If any replica gets $f + 1$ view change request then it makes itself as primary and send new view to all other replicas. The important properties of the pBFT are

- *Safety*: The algorithm provides safety if all non-faulty replicas agree on the sequence numbers of requests that commit locally.
- *Liveness*: Clients eventually receive replies to their requests provided at most $\lfloor \frac{n-1}{3} \rfloor$ replicas are faulty. Or all replicas are arrive at that state.

1.4 Data Structure

It is a method to store the data in an organized way to enable efficient access and modification. Blockchain uses hash pointer and Merkle tree for security as well as efficient access of data. This section discusses about the data structures used in the blockchain.

1.4.1 Hash Pointer

The hash pointers are also useful like any pointer that you use in C, C++ programming to create binary and other variants of trees. Pointer is a concept in programming language and data structure to point the stored data. This helps to locate and retrieve the data. Hash pointer is also a pointer with an additional feature to ensure the data is not modified and it comprised of two parts:

- Pointer to where some information is stored
- Cryptographic hash of that information.

With the help of a hash pointer, we can retrieve the information and verify that the information or data is not changed. Figure 1.14 shows the hash pointer for the information block $\mathcal{B}$. In figure, the information $\mathcal{B}$ is used to compute the hash pointer to $\mathcal{B}$. In case $\mathcal{B}$ is modified then hash pointer available cannot point or match the modified $\mathcal{B}$.

Figure 1.15 shows the chaining of blocks using hash pointer. In figure 1.15, if the data block $\mathcal{A}$ is modified in later stage then it will be detected while traversing because data block $\mathcal{B}$ holds the hash pointer of data block $\mathcal{A}$. Similarly for all blocks, the hash pointer prevents modification. The only way to change the block content is through completely changing the block sequence or blockchain.

Figure 1.14: Hash Pointer

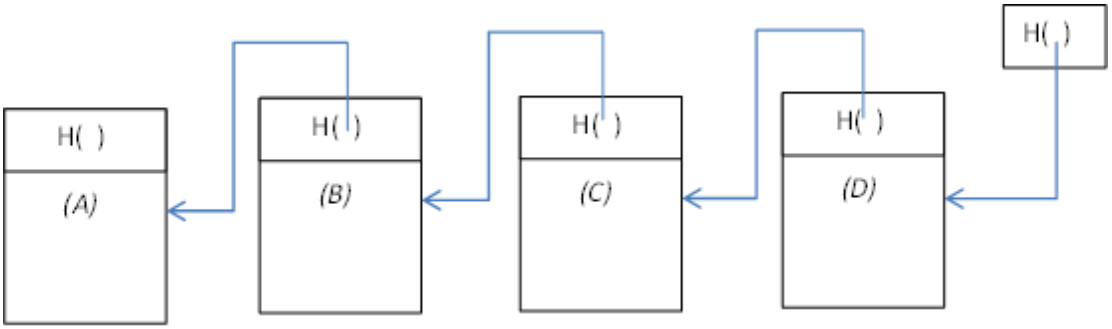

Figure 1.15: Hash Pointer Chain

Now the important question to be answered is how the cryptographic hash value can become the pointer. In programming languages, we use the hexadecimal memory address as the pointer of size 16 or 32 bits but in cryptographic hash pointer it will be 256-bit or 512-bit. The database should be designed in such a way to hold the pointer as 256 or 512 bit for example leveldb used in Bitcoin and Ethereum cryptocurrency implementation.

Hash pointer in blockchain is for two purposes one for the implementation of merkle tree and another for the block linking like a linked list. For example, figure 1.15 shows the usage of hash pointer for linking the blocks in blockchain. In case of merkle tree, node pointers are the hash pointers.
Advantage The hash pointer based merkle tree can prove the membership in $O(log\ n)$ time, where n is the number of leaf nodes.
Disadvantage This cannot be used for the data structure which have cycles. This is because there is no start point in cycled data structure. We should find hash of a starting node and proceed further but in case of cyclic data structure we will not have the starting node that is node without predecessor. In an acyclic data structure, we can sort starting from the leaves or

near the tail that do not have any pointers coming out of them, compute the hashes of those and then work our way back, towards the beginning. But in a structure with cycles, there is no end that we can start with.

1.4.2 Merkle Tree

In cryptography, merkle tree or hash tree is a tree, where leaf nodes are a hash of the data block and non-leaf nodes are hash of its child nodes. The concept of hash trees is named after Ralph Merkle who patented it in 1979 [2]. Figure 1.16 shows the sample merkle tree for the data blocks m_1 *to* m_8. Leaf nodes of the tree labeled with hash value of the data blocks m_i and non-leaf nodes are labeled with hash values of its child nodes labels such as $d' = h(f||g)$, etc. This will recursively computed till the root node where $root = h(a''||b'')$. The hash used in the merkle tree is the cryptographic hash function. Hash trees can be used to verify any kind of data stored, handled and transferred in and between peer network nodes are undamaged and unaltered. Merkle tree is used in blockchain to store the transactions, account balances, etc.

The advantage is in order to locate a block, if you have a chain then you have to do a completely linear search like you have to see if you have the hash pointed to this, then you have to compute hash of the previous blocks to see which one is just before it. And then you have to compute the hash before that to know which one is before that and so on and so forth. So that is a lot of work.

If instead of chaining them you have in the tree structure, and then you have the hash to this, then you know that this block has not been tampered with. So and then you locate this, and then you locate this, and you locate this, and then you locate this. So it is a logarithmic search for a particular block of data. So to locate a block of data, you need to look at log n number of items. And whereas in a chain, you have to look at linear number of items.

The advantage of Merkle tree is that you know with n levels you can handle 2^n blocks of data. Then only need n depth is required, which means logarithmic depth. The membership can be verified in logarithmic time. The membership means suppose you want to know whether this particular data or piece of data is present in the tree. Then a logarithmic search to the trees depth from the root to find that.

Now there is a variant called sorted Merkle tree where the members are also sorted. In that case, verifying non membership is also easy. So here verifying non membership is going to be expensive, whereas here verifying

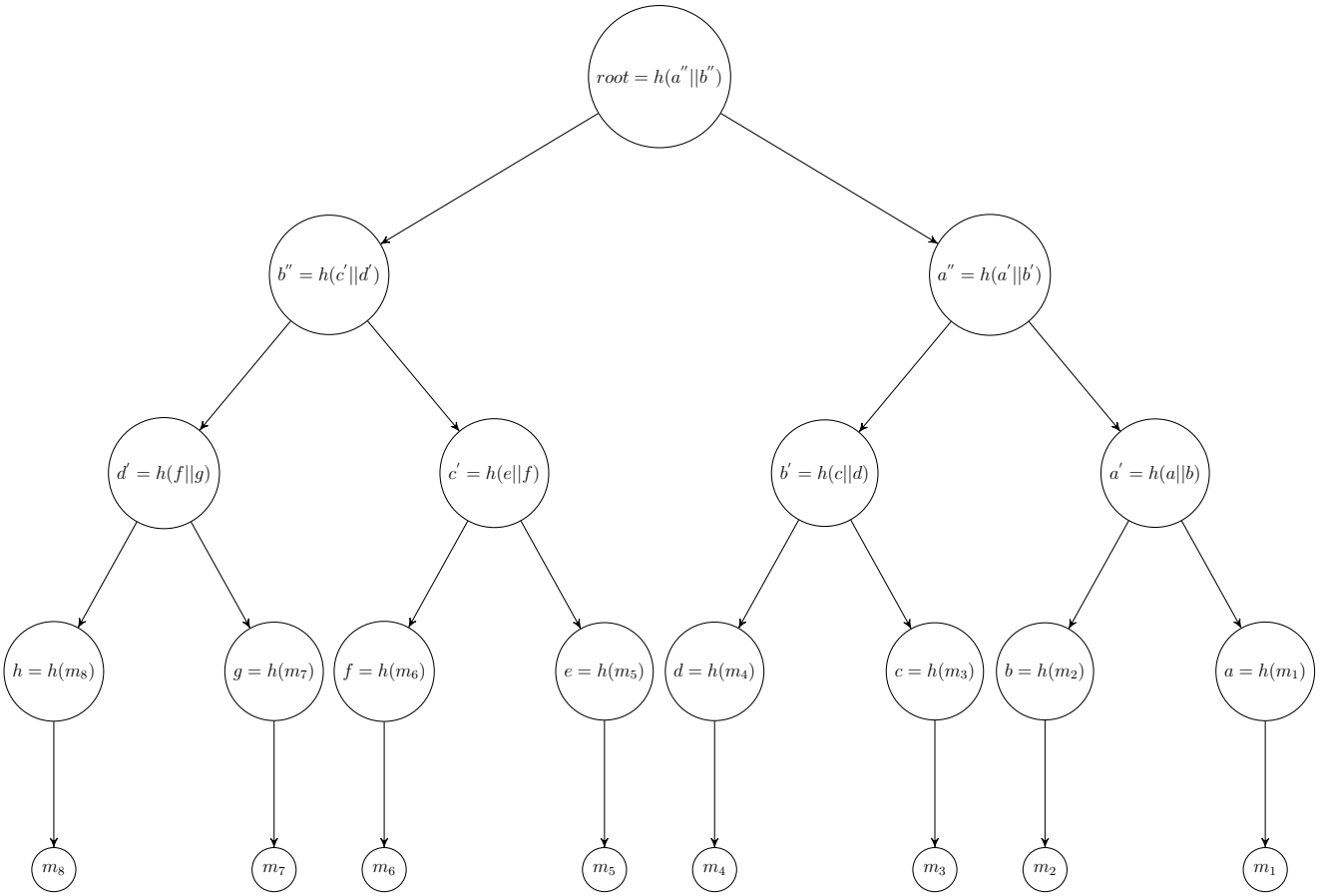

Figure 1.16: Merkle Tree

non membership is also going to be *logn*, because data is sorted. So if you do not find it, you can also find what is before it and what is after it.

So this kind of structure is actually used in one particular case in a very successful deployment of this hash tree technique in Estonia. So in Estonia, the government actually maintains this hash trees for all kind of government documents. And what happens is that when documents are created, they are basically put into this tree structures through the routers under which this network belongs.

And then to the next router to the next router all the way. And then every, let us say every 5 seconds, whatever is created, whatever information is created is put in a tree throughout the country. And then the trees route

is then put into a list. And so the list contains, let us say, every 5 seconds, whatever is created the Merkle tree root of that. Then next 5 second Merkle tree root will be that. And these things are also hash connected. Then the government publishes the latest hash. So you have a data structure which looks like merkle tree. But these are also hash connected. So this hash connected blocks is called a calendar blockchain.

So this calendar blockchain to convince people more into the government what they do is that every weekend they publish the latest hash value. The latest hash value that has been of the block that was created is published in a newspaper. So if somebody changes something here some internal government employee, then he has to change the next one then next one then next one and then eventually public data hash will change.

Now the public has this hash. Therefore, it will be not very easy if the public challenges that I believe that some data has been changed here. Then the government will be in trouble because public can challenge this information. So that is the idea of Merkle tree.

1.4.2.1 Membership Verification of Merkle Tree

This is to ensure whether the given data is member of the merkle tree or not. In any merkle tree, it can be identified through root and path hash. It has an advantage, verifier need not have complete merkle tree to ensure the membership of data only the root hash is sufficient. For example, a person $\mathcal{A}$ containining the root node of the Merkle tree for set $\mathcal{S}$, and person $\mathcal{B}$ wants to convince person $\mathcal{A}$ that element e is in $\mathcal{S}$. To do this, only person $\mathcal{B}$ has to provide respective element to person $\mathcal{A}$ with the siblings of all of the elements in the tree in the path from e to the root node. This is only $log(n)$ nodes. So $\mathcal{B}$ has to provide only $log(n)$ pieces of data, and person $\mathcal{A}$ has to store the root node. With this provided information, person $\mathcal{A}$ can recompute the root node of the tree, and check to make sure that it matches the one, it is correct and element is member of the tree. If the hash function is collision resistant, then $\mathcal{A}$ can be sure that e is in $\mathcal{S}$ [1]. For better understanding, we can discuss with a sample tree shown in figure 1.17.

If person $\mathcal{A}$ holds *H(abcdefghijklmnop)* and person $\mathcal{B}$ want to prove that the value k is part of the merkle tree. The person $\mathcal{A}$ can ensure the membership without knowing k that is with $H(k)$ it can be proved. To verify the membership along with $H(k)$, H(l), $H(ij)$, $H(hmnop)$ and $H(abcdefgh)$ are required. With the available $H(k)$ and $H(l)$, $H(kl)$ can be computed.

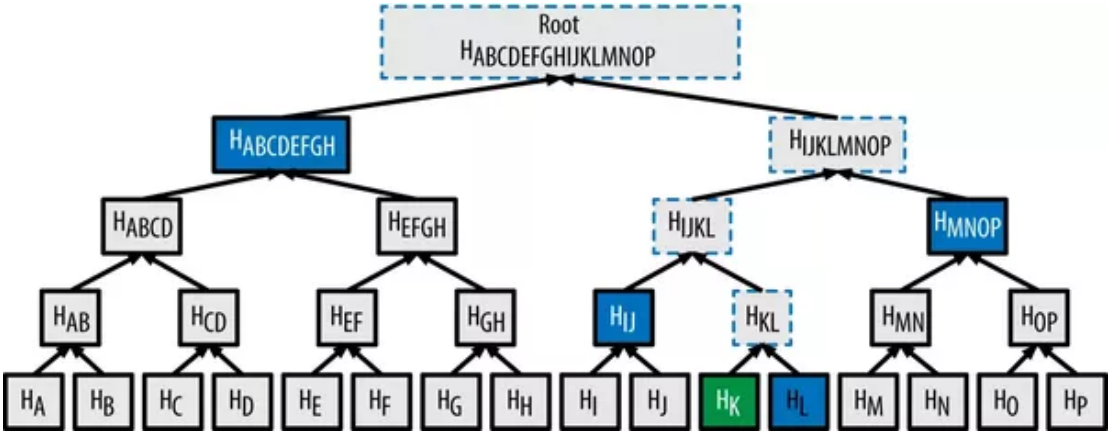

Figure 1.17: Merkle Tree

Likewise, with $H(kl)$ and $H(ij)$, $H(klij)$ can be computed and using $H(klij)$ and $H(hmnop)$, $H(klijhmnop)$ can be computed. Finally, with $H(abcdefgh)$ and $H(klijhmnop)$, $H(abcdefghijklmnop)$ can be computed. The person $\mathcal{A}$ can verify the membership of k in the merkle tree through equation 1.4 using the available root hash ($rh = H(abcdefghijklmnop)$) and the computed root hash ($ch = H(abcdefghijklmnop)$).

$$Descision = \begin{cases} Member & \text{If } rh == ch \\ Not\ a\ Member & otherwise \end{cases} \tag{1.4}$$

1.4.3 PATRICIA Tree

Practical Algorithm To Retrieve Information Coded In Alphanumeric (PATRICIA) is a type of trie tree and it was first described in 1968 by Donald R. Morrison [12]. This is similar to Radix Tree with radix equals to 2 and has an innovative concept to store n items in the n nodes. It is very compact that if a node is only one child for a parent then it get merged with parent. The way it is used in cryptocurrency especially in Ethereum [18] is with the merkle tree concept to ensure the integrity of the leaf node data. This trie uses every part (bit, character, ...) of the key, in turn, to determine which subtree to select. A PATRICIA tree instead nominates (by storing its position in the node) which element of the key will next be used to determine the branching [3]. PATRICIA tree uses the following components.

Longest Common Prefix (LCS): LCS of a set S of strings is a string σ such that

- σ is a prefix of every string in S.
- There is no string σ' such that σ' is a prefix of every string in S,and $|\sigma'| > |\sigma|$.

Leaf node: Every string in S corresponds to a leaf in its Patricia trie.

Extension node: Let S be a set of strings $(s_1, s_2..., s_n)$, and σ the LCS of S. The extension set of S is the set of characters c such that σc is a prefix of at least one string in S.

The Patricia trie T on S is a tree where each node u carries a positional index $PI(u)$, and a representative pointer $RP(u)$. T can be recursively defined using the following steps.

- If $f|S| = 1$, then T has only one node whose its PI is $|S|$, and its RP references $s \epsilon S$.
- otherwise, let σ be the LCS of S. The root of T is a node u with $PI(u) = |\sigma|$, and $RP(u)$ referencing s, where s is an arbitrary string in S.
- Let E be the extension set of S. Then, u has $|E|$ child nodes, one for each character c in E. Specifically, the child node v_c for c is the root of a Patricia trie on the set of strings in S with σc as a prefix.

Figure 1.18 [17] shows an example for the PATRICIA tree constructed using the steps given above.

1.4.4 Merkle Patricia Tree

A node in a Merkle Patricia trie is any one of the following:

- NULL (represented as the empty string)
- branch A 17-item node [v0 ... v15, vt]

Example: Let $S = \{$aaabb$\perp$, aabaa$\perp$, aabab$\perp$, abbb$\perp$, abbba$\perp$, abbbb$\perp\}$. The Patricia trie of S is:

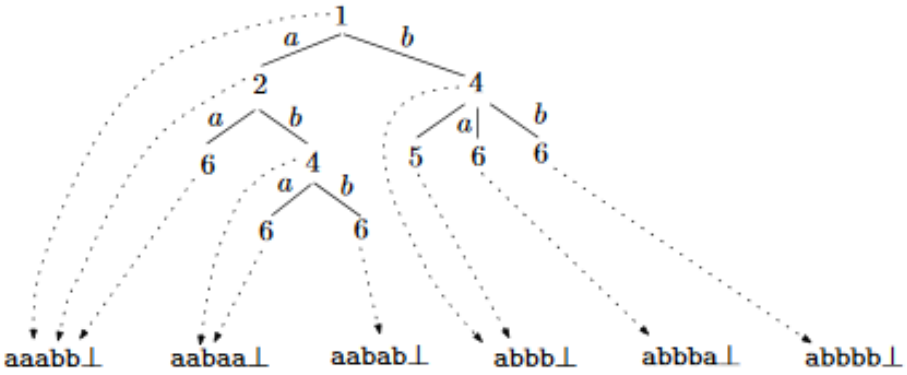

Figure 1.18: Patricia Trie Tree

- leaf A 2-item node [encodedPath, value]

- extension A 2-item node [encodedPath, key]

In this case ,the leaf is the node containing the element even branch node will also have the element at the end of the list. The branch node is the one having the link to further description of the trie tree that is from 0 to 15 (0 to f in hexadecimal). The last field in the branch node is the value belongs to the string end with that node. An extension node is an advantage in the Merkle Patricia Tree to have the link to the next level tree. It contains two field one for the shared substring and other is pointer to the next node.

Since it is an merkle tree, hash pointer is used to safegaurd the leaf node and branch node values. The leaf node and the extension node are differentiated using the prefix value. The root of the tree will have the hash of the complete tree and pointer to the sub nodes.

1.5 Exercises

1. In BFT, $3f + 1$ nodes are mandatory to achieve consensus. If we go for less than $3f + 1$ then we should have trusted nodes in the network? Justify the reason why cannot we ensure the consensus without trusted party if less than $3f + 1$ is the network size.
2. What is the difference between BFT and Raft consensus mechanism? Which one is best with respect to number of faulty nodes?
3. Is the Hash functions NP-hard?
4. What is the state machine in pBFT?
5. Using XOR, perform symmetric key encryption for a message "Blockchain". Take a random value and use as the key.
6. Prove the chance of hash collision using the Birthday problem?
7. What is the difference between SHA 512 and MD5 hash functions?
8. How can we ensure non-repudiation using Digital Signature?
9. The elliptic curve of the form $y^2 = x^3 + ax + b$ over finite field with $a = 0$, $b = 7$ and $q = 23$ has point $P = [1, 13]$. Find $3[P]$.
10. What is the advantage of private permissioned blockchain over conventional distributed database with a trusted third party and distributed consensus mechanism?
11. What is the advantage of Trie tree over binary search tree. What is the worst case search complexity of an element of size m in binary search tree and trie tree. Assume tree has n nodes.
12. Let us assume that, an adversary system takes t second for hash computation, r second for $integrify()$, u second for mod. Adversary stores only $V_1, V_5, V_10...V_n$ (computed using Algorithm 1.2.5.1) in RAM instead of complete V to minimize the memory utilization. If an attacker wish to compute B, how much time will be taken for $n = 15$ and $j = 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 11, 2, 1, 5, 4$. Also compute the time required when all values of V in RAM.
13. How do you prove Non-membership of Merkle tree ?

Bibliography

[1] Cryptography: How does a merkle proof actually work? *Chapter 13*. [Online; accessed 10-March-2019].

[2] Merkle tree. [Online; accessed 10-March-2019].

[3] Patricia. [Online; accessed 10-March-2019].

[4] Distributed databases. *Chapter 13* (1992).

[5] Back, A. Hashcash- a denial ofservice counter-measure. [Online; accessed 18-March-2019].

[6] Bayer, D., Haber, S., and Stornetta, W. S. Improving the efficiency and reliability of digital time-stamping. sequences. *Sequences II, pp. 329–334* (1992).

[7] Castro, M., and Liskov, B. Practical byzantine fault tolerance. *Proceedings of the Third Symposium on Operating Systems Design and Implementation* (1999).

[8] Dwork, C., and Naor, M. Pricing via processing or combatting junk mail. *Proceedings of the 12th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO'92), pp.139–147* (1992).

[9] Hooda, P. Raft consensus algorithm. [Online; accessed 21-May-2019].

[10] Jakobsson, M., and Juels, A. Secure information networks: Communications and multimedia security. *Kluwer Academic Publishers* (1999), 258–272.

[11] LAMPORT, L., SHOSTAK, R., AND PEASE, M. The byzantine generals problem. *ACM Transactionson Programming Languages and Systems, 4(3)* (1982).

[12] MORRISON, D. R. Patricia - practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM, 15(4) pp.514-534* (1968).

[13] ONGARO, D., AND OUSTERHOUT, J. In search of an understandable consensus algorithm. *Proceedings of USENIX ATC '14:*. [Online; accessed 21-May-2019].

[14] SAWANT, S. What is a fork in a blockchain? [Online; accessed 16-March-2019].

[15] SHERMAN, A. T., JAVANI, F., ZHANG, H., AND GOLASZEWSKI, E. On the origins and variations of blockchain technologies.

[16] STUART, H., AND STORNETTA, W. S. How to time-stamp a digital document. *Journal of Cryptology, 3 (2): 99–111* (1991).

[17] TAO, Y. Patricia tries. [Online; accessed 10-March-2019].

[18] WOOD, G. Ethereum: A secure decentralised generalised transaction ledger eip-150 revision. *Yellow paper* (2014).

[19] WUŠT, K., AND GERVAIS, A. Do you need a blockchain? [Online; accessed 16-March-2019].

Contents

Chapter 1

Cryptocurrency

Cryptocurrency is the digital currency implemented using the cryptographic concept and Blockchain. It supports, creation and transfer of currency without any centralized manager or bank. Bitcoin is the first popular cryptocurrency which paved the path for multiple cryptocurrencies. This chapter discusses implementations of cryptocurrencies. As can be seen in this chapter, there are significant differences between these. This chapter is divided into three sections

- Bitcoin- The first section introduces the first popular cryptocurrency Bitcoin and discusses its implementation.
- Ethereum- The second section discusses the implementation and mining principle of ethereum. Ethereum has additional features such as Decentralized Autonomous Organization (DAO) compared to Bitcoin.
- ZeroCash - The third section briefly discusses about an anonymized cryptocurrency ZeroCash .

1.1 Introduction

Nowadays, there are multiple cryptocurrency implementations available for the users however basic terminology are same for all. We first discuss the basic terminologies and later the implementation of popular cryptocurrencies.

Participants: Any individual/group in this world can participate in the bitcoin network on the following category.

- Full node - An user having enough infrastructure to store the growing blockchain and mine the blocks. User must have their key pairs and the derived address.
- Light node - An user store only the header of the blockchain and will have the key pairs and the derived address
- Account with key pair and without Blockchain data - An user have their own key pair and derived address but he/she will not store the blockchain information.
- Account with address and without key pair - An user get the address from an exchange or third party and use it. He/She will not have their own key pair and store the blockchain information. Security of the user is depend on the third party.

It is mandatory for all category of above users in the Bitcoin network to have atleast one address derived from the key pair to own the bitcoin.

Miner: The node that generate or mine the successful block after validating and including transactions and solving the hard Proof of Work (PoW) problem.

Money Generation: In physical currency, the currency notes and coins are printed or made by the organization or agency approved by respective government of the country. In case of cryptocurrency like Bitcoin, there is no centralized agency to generate or make the coins. In Bitcoin, coins are generated or created by miners as an incentive or reward. A miner creating an valid block will also create a new transaction to generate a defined bitcoins (BTC). In the beginning it was 50 BTC and it will reduce to half for every 210,000 block and reach to 0 since bitcoin has the limit of 21 million BTC. The transaction created by miner is called coinbase transaction or generation transaction, which do not have the input transaction but it has the output transaction having his/her address as receiver. This will always be the first transaction in the block.

Block: Block contains record of some or all recent valid transactions along with other attributes to ensure security.

Genesis Block: This is the first block in the bitcoin blockchain. It is almost hardcoded in the software of the applications that utilize its block chain. It does not reference to any previous block, and for Bitcoin or other cryptocurrency, it produces an unspendable incentive. All members of the network should have the same genesis block to take part in the respective bitcoin blockchain network.

Transaction: It is a transfer of Bitcoin from one user to another user and it is broadcast to the network and collected into blocks after validation. A transaction is successful only after it gets added into the block.

Double Spending: Spending the coin more than once is called double spending. It is possible only when the attacker mines two block for same number with and without the respective transaction. PoW mitigates mining two same number blocks in the network. Nevertheless there are chances that attacker mined block ($\mathcal{B}$) may not be largely accepted in the network and go as stale block. If a recipient react based on that transaction available in $\mathcal{B}$ then it will lead to double spend. Hence any bitcoin transaction cannot be accepted by the recipient before minimum six blocks build on top of the block on which respective transaction is included.

Key Pairs: The digital signature scheme used in cryptocurrency is the Elliptic Curve Digital Signature Algorithm (ECDSA) and it is used to sign the transactions. The digital signature elliptic curve cryptographic private and public key pair for Bitcoin or Ethereum is generated from the secp256k1 curve. The reason for using digital signature is to own the cryptocurrency. The Private key is to ensure only owner of the coin is spending it and the public key helps other users in the network to verify the ownership of the coin. For example, *Alice* generate signature for the coin using its private key and all other users in the network can verify the signature using *Alice* public key.

Distributed Ledger: Cryptocurrency uses a distributed ledger known as the blockchain to store transactions carried out between users. Because the block chain is massively replicated by mutually distrustful peers, the information it contains is public.

SECP-256k1 Elliptic Curve: The SECP256k1 curve is defined in Stan-

dards for Efficient Cryptography (SEC) and Bitcoin uses it for the digital signature

$$E : y^2 = x^3 + ax + b \; over \; F_p \tag{1.1}$$

The elliptic curve equation given in equation 1.1 and discussed in Introduction section takes the following values for its constants a, b and p along with base point G, order n of G and co-factor h for SECP256k1.

$$a = 0$$
$$b = 7$$
$$p = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$$
$$G = (x_1, y_1)$$
$$x_1 = 79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D9$$
$$59F2815B16F81798$$
$$y_1 = 483ADA7726A3C4655DA4FBFC0E1108A8FD17B448A6855419$$
$$9C47D08FFB10D4B8$$
$$n = FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6$$
$$AF48A03BBFD25E8CD0364141$$
$$h = 01$$

Elliptic curve points are derived using the above values and the point based on the random integer value (that is private key) is picked as the public key for bitcoin users.

Attacker: An individual or group members with malicious intention try to sabotage the bitcoin network by doing double spending the coin, isolating a genuine user, etc.

1.2 Bitcoin

Bitcoin is the first fully decentralized cryptocurrency and a popular application of a public distributed ledger called Blockchain. It was introduced in 2008 by unknown individual/group Satoshi Nakamoto and he maximum

limit of Bitcoin is 21 million BTC. It completely implements a peer to peer financial network without any financial organization. Bitcoin works based on transactions and it does not store the account balance that means user can spend the BTC available in one transaction through another transaction not by deducting from the account balance. Any digital currency has the problem of double spending because no physical form of currency available to control the duplication. Bitcoin also has the double spending problem but it mitigates through Proof of Work (PoW) distributed consensus.

1.2.1 How Bitcoin Works

Bitcoin network is a peer to peer network with many users to send and receive bitcoins (BTC) among themselves. Any one in this world can participate in the Bitcoin network however he/she should have the private and public key pair with derived public address, which is unique for every user and may be unique for every transaction to own it. Bitcoin transaction is a transfer of bitcoin from one user to another user through their public address. User wish to transfer the bitcoin makes the transaction and broadcast to the bitcoin network. A transaction has input and output, where input is references to the previous transaction to be spent and output is for transferring bitcoins to receiving address. An other users of the network receives the transaction and add in their transaction pool for validation. Miner, who is an user in the network with sufficient infrastructure, validate the transaction and include in the block if it is valid. A block can have many transactions however size of the block should not exceed 1 MB to enable fast propagation. After validating the transaction and constructing the candidate block, miner has to solve a Proof of Work (PoW) hard problem to create the valid block. A PoW is distributed consensus protocol to mitigate double spending of coin, reduce the growing speed of blockchain and mitigate the orphaned blocks. After successful mining of a valid block, miner broadcast it to the network and that will be validated by other users before add in to the blockchain and transaction included in the block will be transferred to Unspent transaction output (UTXO) list.

1.2.2 Bitcoin Script

Bitcoin uses a scripting system for transactions [16]. A script is list of instructions appended with each transaction that describes how the next person wanting to spend the bitcoin can gain access to the transaction. The script for a typical Bitcoin transfer to destination address $\mathcal{D}$ includes two following things for future spending.

- *scriptPubKey* - A public key or its hash embedded in the script to derive destination address $\mathcal{D}$.
  Ex. scriptPubKey: OP_DUP OP_HASH160 <public_key hash> OP_EQUALVERIFY OP_CHECKSIG
  - OP_DUP instructs to duplicate the top of the stack item.
  - OP_HASH160 instructs to compute the hash of stack top item.
  - OP_EQUALVERIFY verifies the equality of stack two top items.
  - OP_CHECKSIG verifies the signature available in the stack using the public key available in the top of the stack.

  A transaction is valid if nothing in the combined script triggers failure and the top stack item is True (non-zero) when the script exits. The stack top item can be true if equality is checked between the top two stack items. The <public_key hash> is the address, which is derived from the public key.

- *scriptSig* - A signature to prove ownership of the private key corresponding to the public key provided.
  Ex. scriptSig: <signature> <public key>
  The <signature> and <public key> are the signature of the transaction hash and public key of the sender.

Any user in the network can verify the ownership of the transaction using the script and the signature, public key and public key hash available in the transaction. The script execution for validation of transaction is shown in table 1.1 [16] for the following.

scriptPubKey: OP_DUP OP_HASH160 <public_key hash> OP_EQUALVERIFY OP_CHECKSIG
scriptSig: <signature> <public key>

Table 1.1: Script Execution

Stack	**Script**	**Description**
Empty	<sig> <pubKey> OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG	scriptSig and scriptPubKey are combined.
<sig> <pubKey>	OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG	Constants are added to the stack.
<sig> <pubKey> <pubKey>	OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG	Top stack item is duplicated.
<sig> <pubKey> <pubHashA>	<pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG	Top stack item is hashed.
<sig> <pubKey> <pubHashA> <pubKeyHash>	OP_EQUALVERIFY OP_CHECKSIG	Constant added.
<sig> <pubKey>	OP_CHECKSIG	Equality is checked between the top two stack items.
true	Empty.	Signature is checked for top two stack items.

Bitcoin has many script words such as constant, flow control, stack, splice, bitwise logic, arithmetic, crypto, locktime, pseudocodes and reserved words [16] however this book only introduces script with an example does not discuss all the script words.

1.2.3 Address

Every user of bitcoin network should have unique address to receive and spend bitcoins. A pay-to-pubkey based transaction was used before the bitcoin address is introduced. In pay-to-pubkey, public key is used like address but it has the size issue. Later, key hash is introduced as address, which is of less in size compared to key size. Bitcoin uses addresses, size varying between 26 and 35 characters to identify a user and a transaction. Bitcoin transaction does not need $from$ address, it should only have the recipient address. There are different address types namely Pay-to-Script-Hash(P2SH), Pay-to-PubKeyHash (P2PKH),and Bech32 segwit address. However, Pay-to-Script-Hash(P2SH) is by default used and Pay-to-PubKeyHash (P2PKH) is the predecessor and it can be used if required by user. In both, address are generated using the compressed or uncompressed public key of the user. In elliptic curve digital signature algorithm, private key is a scalar value and public key is a vector value (x,y). If we take x alone then it is called compressed public key otherwise it is called as uncompressed public key. A Key type can be identified using the prefix for example, $0x04$ is to represent uncompressed public key, $0x02$ is to represent compressed public key, which has even value for y and prefix $0x03$ is to represent compressed public key, which has odd value for y [48]. Algorithm 1 shows the process of generating the address using the uncompressed public key.

Algorithm 1 Bitcoin address generation using uncompressed public key

Require: *Elliptic curve Public key* $\mathcal{U}_{pub} = (\mathcal{X}_{co-ordinate}, \mathcal{Y}_{co-ordinate})$
Ensure: *Address* $\mathcal{A}$

1: $\mathcal{I} = ripemd160(sha256(0x04||\mathcal{U}_{pub}))$ #{ripemd160 is the hash function and $\|$ is the concatenation}
2: $\mathcal{F} = sha256(sha256(ID||\mathcal{I}))$ #{sha256 is the hash function and ID is the network identity}
3: $\mathcal{A} = base58(ID||\mathcal{I}||\mathcal{F}[1..4])$ #{$\mathcal{F}[1..4]$ is first 4 bytes of $\mathcal{F}$ and $base58$ is binary to text enconding}

Bitcoin supports different networks, some are given in table 1.2 along with its identity to be added at the time of generating address and the prefix (leading symbol) to be added with address to identify its type. An example address *1BvBMSEYstWetqTFn5Au4m4GFg7xJaNVN2* of main network P2PKH.

Table 1.2: Nework Identity

Name	**Identity**	**Leading Symbol**
Main Network (Pubkey hash (P2PKH address))	0x00	1
Test Network (Pubkey hash (P2PKH address))	0x6F	m or n
Main Network (Script hash (P2SH address))	0x05	3
Test Network (Script hash (P2SH address))	0xC4	2

1.2.3.1 Wallet Import Format (WIF)

It is a way of encoding a Elliptic curve private key to have shorter form and avoid copying error. It includes built-in error checking codes so that typos can be automatically detected and/or corrected. Wallet import format is the most common way to represent private keys in Bitcoin [11]. Algorithm 2 shows the process of converting the private key to the Wallet Import Format. In step 1 of the algorithm, 0x01 should be added at the end if the private key will correspond to a compressed public key otherwise no addition of hexadecimal required. We can derive the private key from the WIF if required by decoding and removing the header (network identity) and trailer (checksum) part. The checksum is to verify the integrity of the key that is it is not modified.

1.2.3.2 Collisions

Bitcoin addresses are basically random numbers and it is possible, although extremely unlikely, that two users independently generate the same address called as collision. In such case, both the original owner of the address and the colliding owner could spend money sent to that address. It would not be possible for the colliding person to spend the original owner's entire wallet if

Algorithm 2 Wallet Import Format

Require: *Elliptic curve Private key* $\mathcal{U}_{pri}$
Ensure: *WIF of Private key* $\mathcal{A}$
1: $\mathcal{I} = ID||\mathcal{U}_{pri}$ #{ID is the network identity}
2: $\mathcal{F} = sha256(sha256(\mathcal{I}))$ #{sha256 is the hash function}
3: $\mathcal{A} = base58(\mathcal{I}||\mathcal{F}[1..4])$ #{sha256 is the hash function and $F[1..4]$ are first four bytes of F}

different address is used for each transaction. If you were to intentionally try to make a collision, it would currently take 2^{107} times longer to generate a colliding Bitcoin address than to generate a block. As long as the signing and hashing algorithms remain cryptographically strong, it will likely always be more profitable to collect generations and transaction fees than try to create collisions [10].

1.2.4 Transaction

A transaction is transfer of a digital bitcoin for one user to another user. If *Alice* wish to transfer or spend a coin then she creates a new transaction by digitally signing the hash of the previous transaction called as input transaction, adding address of the next owner and her public key as shown in figure 1.1. A payee or any other member in the network can verify the signatures to confirm the chain of ownership of the coin [44]. In figure 1.1, payer $\mathcal{C}$ creates transaction III by computing signature of its own coin transaction II using his/her private key, adding its public key and payee $\mathcal{D}$ address. Any user of the network including $\mathcal{D}$ can access the public key of the transaction III and validate the transaction. User can derive $\mathcal{C}$'s address from the available public key and ensure the transaction is belong to $\mathcal{C}$. User can use the public key and verify the signature to ensure $\mathcal{C}$ is spending the coin.

There are three collection of transactions

- Transaction pool - An unordered collection of transactions that are not in blocks of the main chain, but for which we have input transactions. It is the pool of transactions to be spent.
- Orphaned transaction pool - Transactions that cannot go to the transaction pool due to one or more missing input transactions will be into the Orphaned transaction pool.

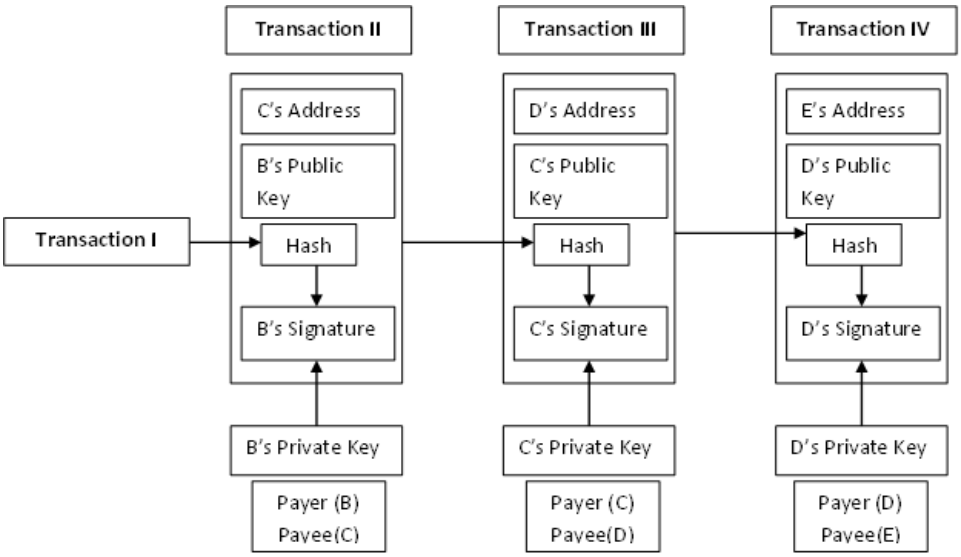

Figure 1.1: Bitcoin Transaction overview

- UTXO pool - Unspent Transaction Output (UTXO) pool is to store all the unspent and valid transaction. It is discussed in details in later section.

Transaction Signature: The signature of a transaction is computed using the cryptography hash and encryption function. It takes the hash of the transaction and encrypt it using the sender's private key. Any user knowing the transaction and public key of the signer can verify the transaction signature using the following steps.

- Step 1: Using public key, the signed that is encrypted hash will be decrypted.
- Step 2: Compute hash of the transaction.
- Step 3: Compare of step 1 and step 2 output. If both matches signature is valid otherwise invalid.

Multisignature Transaction: Multisignature transaction is a transaction that is signed by multiple users. It is generally used to divide up responsibility for possession of bitcoins. An organization may depute more

than one person to look after all bitcoin transaction. To spend the coin, all of them have to sign the transaction or M out of N signature is enough. Here, more than one person indicates more than one private keys are required, where N is greater than M. It is required that all of them have to agree to spend the coin or majority should agree on spending the coin.

1.2.4.1 Transaction Creation

Bitcoin supports 1 or n input and 1 or n output transactions. For example, *Alice* has two transactions one with 10 BTC and another with 5 BTC and she has to pay 13 BTC to *Bob*. Now she has to create a transaction including two transactions (10 BTC and 5 BTC) as input because no fraction of value in transaction is permitted in Bitcoin and two transactions as output in that one is for *Bob* with 13BTC and another for herself with 2 BTC. In case *Alice* has to pay only 10 BTC to *Bob* then she will create transaction with one input (10 BTC) and one output transaction for *Bob*. The general format of a bitcoin transaction is given in table 1.3 [14] with its size. The Input and output counters in a transaction (for eg. $\mathcal{A}$) are to define the number of input and output transactions involved in executing $\mathcal{A}$. The parameter Lock_time is to indicate the earliest block or time the transaction should be added in to the blockchain. In Banking system, we have fixed deposit to keep the money unspent for a period of time to get the interest and save the money. Similarly, lock time in bitcoin helps to lock the BTC for saving. If the Lock_time value is less than 500 million that means it represents the block height otherwise it is the unix timestamp (number of seconds since 1st January 1970). The parameter Witnesses will have the signatures of all input transactions for that transaction. In the typical bitcoin transaction, signature of a transactions is added at the end of each transaction but in the SegWit (Segregated Witness) transaction, witnesses are segregated from the list of input transactions.

SegWit (Segregated Witness): The witnesses are the signatures of the input transactions. Instead of placing the signature after every input transaction, all signatures are segregated from its transactions and placed altogether at the end of the transaction. In simple explanation, transaction has three parts: one is input transaction, second is output transaction and third is general information such as version number, flag, input counter, output counter, witnesses and lock time. The signature to be placed inside the input transactions will be separated and put in the witnesses part of the

Table 1.3: Transaction parameters

Field	**Description**	**Size (in Bytes)**
Version No.	This is to indicate the transaction version.	4
Flag	This is to indicate whether witness data present in it or not	2 (optional)
Input counter	Number of input transactions considered for this transaction	1- 9
List of Inputs	Input transactions. The format of each input transaction is given in table 1.4	Variable
Output counter	Number of output transactions in this transaction	1-9
List of outputs	Output transactions. The format of each input transaction is given in table 1.5	Variable
Witnesses	A list of witnesses that is 1 signature for each input transaction. This is omitted if flag in above is missing.	Variable
Lock_time	This indicates the earliest time or earliest block when that transaction may be added to the block chain. If non-zero and sequence numbers are $<$ 0xFFFFFFFF then it refers block height or timestamp when transaction is final otherwise it is transaction lock time.	4

general information. The signatures in the SegWit are not part of the bitcoin block size computation. Bitcoin blocks cannot exceed 1MB of size. Since the signatures are not part of the block size computation, more transactions can be added to mitigate the waiting time of transaction approval.

Input and Output Transaction:The format of an input transaction given in table 1.4. The Previous Txout-index refers to the previous transaction output index that is in one transaction there may be multiple output transactions. Index refers the i^{th} output transaction, which is used for creating new transaction. For example, in Figure 1.2 new transaction is created using the old (previous) transaction hash and output. Transaction index for the first old input and output transactions are assigned O and it increase by one for every next transaction. New transaction input uses the old transaction second output that is index 1.

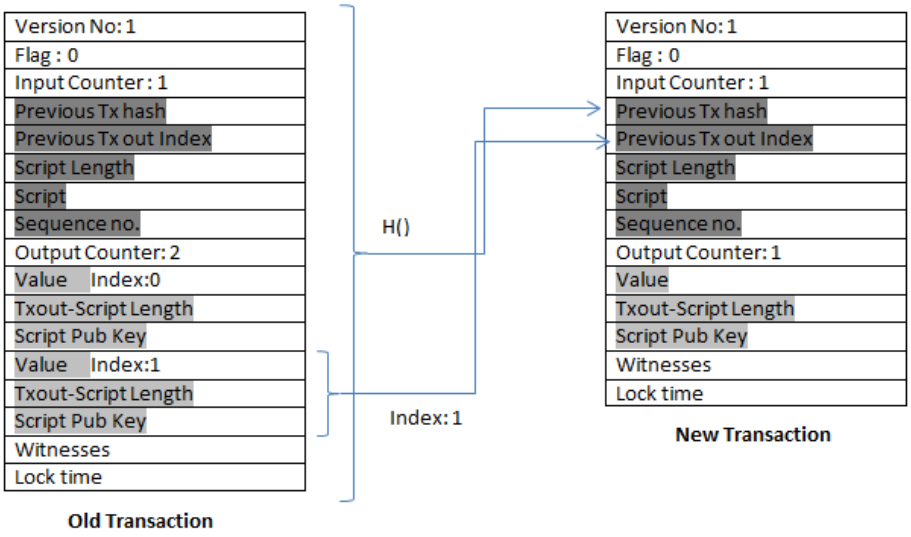

Figure 1.2: Bitcoin Transaction overview

The format of an output transaction is shown in table 1.5.

1.2.4.1.1 Coinbase Transaction : A transaction, which does not have any parent transaction is referred as coinbase. This is for the rewards and will have some arbitrary data as parent. For example, genesis block has *The Times 03/Jan/2009 Chancellor on brink of second bailout for banks*

Table 1.4: Input Transaction parameters

Field	**Description**	**Size (in Bytes)**
Previous Transaction hash	Doubled SHA256 hash of a input transaction	32
Previous Txout-index	The index number of the specific output to spend from the transaction	4
Txin-script length	Length of the script that is scriptSig	1 - 9
Txin-script	scriptSig: Signature and Public key	Variable
sequence_no	This is intended transaction replacement. Replacement is currently disabled and it is irrelevant unless transaction's lock_time is > 0. Before LockTime expires, you can replace the transaction with as many new versions as you want. Initial version is 0 and newer versions have higher sequence numbers	4

Table 1.5: Output Transaction parameters

Field	**Description**	**Size (in Bytes)**
Value	Bitcoin (BTC) to be transferred	32
Txout-script length	Size of the output transaction script	1- 9
Txout-script	Receiver Public key or Public key hash	Variable

as parent [13]. It is mandatory for all the blocks to include reward and transaction fee. Also this is the first transaction of every block.

1.2.4.2 Transaction Validation

Transactions constructed and broadcasted to the network gets validated to importantly ensure that the respective owner is spending the coin not randomized user. Following are the validation rules [25].

- Empty - Ensure neither input and output transaction is empty.
- Structure - The transaction is syntactically correct.
- Size - The size of the transaction is <= MAX_BLOCK_SIZE and greater than or equal to 100 bytes.
- Range - Transaction's each output value, as well as the total, must be in legal money range. Similarly for the input transaction referred output transactions value.
- Insufficient BTC - Reject if the sum of input values less than sum of output values.
- Low Transaction fee - Reject if transaction fee (defined as sum of input values minus sum of output values) would be too low to get into an empty block. Empty block will not have any transaction other than coinbase transaction.
- Public key - Verify the scriptPubKey accepts for each input otherwise reject.
- Locktime - LockTime is less than 31 bits.
- Script Standard - Reject "nonstandard" transactions: scriptSig doing anything other than pushing numbers on the stack, or scriptPubkey not matching the two usual forms.
- Double Spending - Reject if already have matching tx in the pool, or in a block in the main branch. For each input, if the referenced output exists in any other tx in the pool, reject this transaction.

- Orphan transaction - For each input, look in the main branch and the transaction pool to find the referenced output transaction. If the output transaction is missing for any input, this will be an orphan transaction.
- Coinbase transaction - For each input, if the referenced output transaction is coinbase (i.e. only 1 input, with hash=0, n=-1), it must have at least COINBASE_MATURITY (100) confirmations; else reject this transaction
- No output transaction - For each input, if the referenced output does not exist (e.g. never existed or has already been spent), reject this transaction.

1.2.4.3 Unspent Transaction Output

Unspent Transaction Output (UTXO) is an output transaction not yet spent by the owner. All UTXO will be in the UTXO list available with all full nodes of the network and that will be used as inputs for the later transactions. Once it is spent then the respective transaction will be removed from the UTXO. UTXO may have the dust and unprofitable transactions, which may not be taken for execution.

Unprofitable Transaction: A transaction that holds less value than the fee necessary to be spent, resulting in financial loss called as unprofitable transactions. Financial loss refers that the miner will not get any advantage of including the transaction in the block.

Dust Transaction: A transaction for which the fee to redeem is greater than 1/3 of its value is called as dust output transaction [34].

The difference between dust and unprofitable transactions are only with transaction fee quantity. All dust transactions are unprofitable transactions but not all unprofitable transactions are dust transactions.

1.2.4.4 Transaction Age

Transactions broadcasted for validation and inclusion in the block always points to unspent transaction output (UTXO). These block are collected by every full node and put in the transaction pool. Miner chooses transactions

from the transaction pool based on priority and include in the block after validation. Transactions are prioritized based on the age of UTXO and high value inputs [29]. The age of a UTXO is the number of blocks that have elapsed since the UTXO was recorded on the blockchain or number of blocks that are build on top of the UTXO included block in the blockchain. Prioritized transactions will be processed and included in the block without any transaction fee if space is available in the block. The priority of a transaction is calculated using equation 1.2.

$$Priority = \frac{\sum_{i=1}^{n}(Value\_of\_input_i * Input\_Age_i)}{Transaction\_Size} \tag{1.2}$$

Where n is the number of input transactions in the respective transaction. The $Input\_Age_i$ is i^{th} input UTXO age that is number of blocks that are build on top of the UTXO included block in the blockchain. The $Value\_of\_input_i$ in equation 1.2 is i^{th} input UTXO value (BTC) in bitcoin base unit that is Satoshi unit. The satoshi is a smallest unit of the bitcoin currency recorded on the blockchain and named in collective homage to the original creator of Bitcoin, Satoshi Nakamoto. It is a one hundred millionth of a single bitcoin (0.00000001 BTC). The size of a transaction is measured in bytes. The $Transaction\_Size$ is the size of the transaction in total including all input and output transactions. A transaction can be considered as high priority if the equation 1.2 result is greater than 57,600,000, which is possible when the value of input is atleast 1 BTC, aged one day (144 blocks) and size of 250 bytes.

The first 50 kilobytes of transaction space in a block are set aside for high-priority transactions. Miner will fill the first 50 kilobytes, prioritizing the highest priority transactions first, regardless of fee. This allows high-priority transactions to be processed even if they carry zero fees. This concept introduced with notion "bitcoin days destroyed".

Bitcoin days destroyed: It is introduced because it was realised that total transaction volume per day might be an inappropriate measure of the level of economic activity in Bitcoin. Just like that, someone could repeatedly sending the same money back and forth between their own addresses. If same 50 BTC sent back and forth 20 times, it would look like 1000 BTC worth of activity, while in fact it represents almost nothing in terms of real transaction

volume and creates spam in the network. The "bitcoin days destroyed" is an idea that to give more weight to coins which have not been spent in a while and prioritize.

1.2.4.5 Transaction ID (TxID)

A TxID is basically an identification number for a bitcoin transaction. It is the reversed byte order of transaction's two times SHA256 hash (SHA256(SHA256(transaction))) [27]. There is a possibility that two coinbase transactions can get same transaction id as shown in 1.3 because coinbase transaction usually have same data except the miner address. In case, miners use same address for more than one coinbase transaction then it produce same TxID for all those transactions. To solve this, Bitcoin Improvement Proposal (BIP) 30 introduced a rule that prevented blocks from containing a TxID that already exists. Later, BIP 34 suggested miner to include the height of the block in to their transaction data to get different coinbase transactions identity.

e3bf3d07d4b0375638d5f1db5255fe07ba2c4cb067cd81b84ee974b6585fb468:
block 91,722: 00000000000271a2dc26e7667f8419f2e15416dc6955e5a6c6cdf3f2574dd08e
block 91,880: 00000000000743f190a18c5577a3c2d2a1f610ae9601ac046a38084ccb7cd721
d5d27987d2a3dfc724e359870c6644b40e497bdc0589a033220fe15429d88599:
block 91,812: 00000000000af0aed4792b1acee3d966af36cf5def14935db8de83d6f9306f2f
block 91,842: 00000000000a4d0a398161ffc163c503763b1f4360639393e0e4c8e300e0caec

Figure 1.3: Transaction Id Collision

1.2.4.6 Simplified Payment Verification

It is introduced to support the light nodes who are not having enough resources to store complete blockchain. Light nodes get the headers of the longest blockchain from trusted neighbours and stores it. In case light node

want to verify the transaction, he/she will access the merkle bunch from the neighboring full node and validates the received transaction and merkle root hash with the available merkle root hash in the block header. Linking the transaction in the main chain demonstrates that a network node has accepted it, and blocks added after it further confirms.

Issues: Simplified Payment Verification (SPV) introduces security and privacy issues. For example, if a light node request for a transaction bunch from a full node then the respective full node will get inference that the transaction is connected to the light node. Also overwhelmed attacker could cheat the light-node with fabricated transaction.

1.2.4.7 Reclaiming Disk Space

Instead of keeping all old transactions on the merkle tree of the blockchain, it can be discarded after a period. This helps in reducing the storage space. However, this is not yet implemented in the real time bitcoin cryptocurrency because in financial system all transactions from the beginning is required to track the owners and their transactions.

1.2.5 Mining

Bitcoin Mining is the process of generating a new block by validating the transaction and solving the hard problem. Miner of the network validate the transaction and mine the blocks by including all valid transactions. A bitcoin user with sufficient infrastructure can mine the blocks and broadcast it to the network. An other nodes in the network validates the block and include in the chain if it is valid. Block mined and successfully linked with the main block chain is almost impossible to modify. Impossible only when 51% hash power is not with the attacker(s). It means, an attacker can make the block as orphaned block, which will not contribute to the main chain if have less hash power. In Bitcoin or any other blockchain application with Proof of Work, chain with the longest length that is having more blocks will be the main chain. For example, in figure 1.4, the block with grey border and pointer is not contributing to the main chain because this block is orphaned or forked. An effort of miner to mine such block will go useless and transactions included in that block become unrecorded. Block will be considered as successful one when it takes part in the main chain. A fork is

possible when more than one miner mines the same numbered block and it is quite obvious to happen because any user in the network is allowed to mine the block. For example, two or more miners mined the same numbered block (Block:2) and broadcast in the network. After receiving any one of those blocks, miner will start mining next numbered block. Similarly all miners will mine next block based on any one block he/she received. There will be a chance that only one miners block (Block:2) get more block on top hence part of the main chain. Miners ($\mathcal{A}$) block without getting more block on top will go as the fork or orphan. However, $\mathcal{A}$ will try to bring their block included chain as main chain but it may go unsuccessful if number of blocks linked to the main chain is more and other miners will not take those blocks for building next block because they do not get incentive or reward.

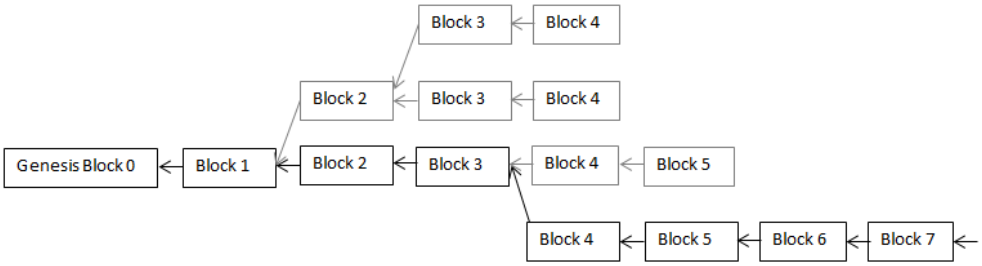

Figure 1.4: Blockchain with fork

1.2.5.1 Block

Block is a file where data pertaining to bitcoin is permanently recorded. Data in bitcoin is the bitcoin transactions. In addition to transactions, it contains meta-data to link with other blocks to ensure security. The parameters of the bitcoin block are as follows [1] [2]

- Magic number - magic value 0xD9B4BEF9 for main network, and 0xDAB5BFFA for testnet [12]. The magic number is to find the type of file just by looking at the first 4 bytes similar to other file system and data structure. Bitcoin uses magic numbers to identify its network. In Bitcoin, it is to find the type of block that is whether it is block of main network or test network or any other network.

- Blocksize - Size of the block represented in 4 bytes.
- Version - Block version number is represented in 4 bytes. This is to identify the software update it follows. Every protocol implementation has an version number to identify which protocol block it is. Bitcoin has different version, the first was 106 and as of 2017 it is 70015 for Bitcoin Core 0.13.2. This is to indicate what type of version this block belongs to. Before accepting the block, receiver ensures the block is of the version what he/she follows. If it is same then accept the blocks otherwise reject.
- hashPrevBlock - 256-bit hash of the previous block header.
- hashMerkleRoot - 256-bit hash of the root hash of merkle tree, which includes all validated transactions and coinbase transaction that are part of the block.
- Time - Linux timestamp of the miner which is represented using 4 bytes
- Target or nBits - This is the value to ensure effort of the miner to mine the block. The hash of a block should be less than or equal to this value. The target changes for every 2016 blocks approximately two weeks considering 10 minutes per block. The method of computing the target is discussed in later sections. The size of this is 4 bytes.
- Nonce - It is a 32-bit number (starts at 0) along with block header (except nonce and blockhash) to find the hash value less than or equal to the target. Nonce means number once. If we reuse the number then the hash value repeats which does not have any meaning.
- Number of Transactions - It refers the number of transactions included in the block. It size is between 1 and 9 bytes
- Transactions - Non empty list of transactions in the form of merkle tree of variable size.

The bitcoin block parameters are divided into header and data parameters. The 80 bytes header of the bitcoin block includes only Version, hashPrevBlock, hashMerkleRoot, Time, Target and Nonce. Transaction parameters is the data part of the block. If we say hash of the block that means hash of

the header. Transactions are not part of the block hash computation however the transactions merkle root hash is part of it. The parameters magic number and block size are the additional data to the block for interpretation. Light nodes, which does not have sufficient storage, stores only the 80 bytes of header and use simplified payment verification for validation of transaction. The simplified payment verification method is discussed in the later section.

There are following three types of blocks possible in bitcoin blockchain.

- Blocks in the main branch - The transactions in these blocks are considered as confirmed.

- Blocks on side branches off the main branch (fork) - These blocks have lost the race to be in the main branch and such blocks are called as stale block. In figure 1.4, the blocks with grey border are the stale blocks.

- Orphan blocks - these are blocks which does not link into the main branch, normally because of a missing predecessor or nth-level predecessor. However, in all discussion on forums, stale blocks are called as orphan block [8].

Blocks in the first two categories form a tree rooted at the genesis block, linked by the previous pointer, which points toward the root. The main branch is defined as the branch with highest total difficulty, summing the difficulties for each block in the branch.

The parameters Target and Nonce are used for the Proof of Work (PoW) distributed consensus to ensure sufficient amount of time is spent to mine the block. Bitcoin uses Proof of Work to ensure that all participating nodes agree on the same branch of the blockchain. To achieve PoW, miner solve a computationally expensive problem, mines the new block and broadcasts it to the other nodes in the network which can easily validate based on the values present in the block [45]. There could be thousands or more nodes of the network around the world simultaneously solving the same PoW problem. Intention of giving complex problem to miner is to reduce the chances of nodes solving the problem at the same time and thereby reduce the chance of forks.

1.2.5.1.1 Block Validation Users validate the block before adding the block into main chain. The first validation is, the block should follow the syntactic structure and the hash of the block should be less than the target otherwise it will be rejected. Following are some other important validations to be done before including the block in main chain. [25] [37].

- Duplicate - Received block should not be duplicate.
- Timestamp Validation - Block will be valid if its timestamp is greater than the median timestamp of previous 11 blocks and less than the *network-adjusted time* + 2 hours. The *Network-adjusted time* is the median of the timestamps returned by all connected peer nodes [3]
- Transaction - Transaction list must have atleast one transaction (coinbase transaction) or more transactions and it must be valid according to transaction validation process. First transaction should be coinbase transaction.
- Merkle Hash - Ensure that the merkle hash present in the block header and the hash computed for the merkle tree containing all transactions matches.
- Previous Block Hash - Verify the previous block hash is in main branch or side branches. If not add into orphaned blocks.
- Target - Check that nBits (target) value matches the difficulty rules.

1.2.5.2 Bitcoin Consensus

A consensus mechanism is to achieve the necessary agreement on a single data value or a single state of the network among distributed processes or multi-agent systems. It is useful in record-keeping [36]. The Bitcoin operate as decentralized and self-regulating system work on a global scale without any single authority. It involves contributions from more than thousands of participants who work on verification, authentication of transactions occurring on the bitcoin, and on the block mining activities. Hence, a fair, reliable, and secure consensus mechanism is required to ensure that all the transactions and block mining occurring on the network are genuine and all participants agree on the status of the shared blocks otherwise network will face the byzantine general problem. Proof of Work(PoW) consensus mechanism is the one in bitcoin ensures the above requirement with a set of rules.

1.2.5.3 Proof of Work

The Bitcoin Proof of Work(PoW) uses the concept from HashCash, which was invented in 1997 by Adam Back for anti-Denial of Service (DoS) [6]. The intention is to control DoS initiator by giving a task, which will consume sufficient effort of the attacker before performing the attack. Bitcoin used this for the distributed consensus, which is the collective decision. In Bitcoin Proof of Work, miner has to spend sufficient time to mine the block to convince the members of the network that he/she spent time and resources and the block will be successful only if majority of the members accept that block. Advantage of Proof related distributed consensus (PoW, PoS, PoB, etc.) over other distributed consensus is that no need to wait for the approval from other members of the network to mine the block. Miner has to mine the block and broadcast to the network. Block will be successful if it is accepted by the majority of the network members other not. In bitcoin, distributed consensus is required for the acceptance of transactions in the block. The process of transaction acceptance is as follows.

- An user, creates a transaction and broadcast to the network.
- Miner takes the transaction, validate and include in the block and broadcast the block to other members of the network, if transaction is genuine.
- Other users of the network validate the block and transaction, accept if it is valid.
- The transaction included block will achieve the consensus if it is accepted by majority of users and blocks are build on top of it.

To achieve PoW, Bitcoin miners have to solve hard problem but just in one attempt other members verify the correctness of the solution. At first, miner construct the candidate block that is block without nonce. To achieve PoW, SHA256 hash will be computed for the candidate block and nonce (start at 0 and increment) till the output is less than or equal to the target value. The target value will be computed for every 2016 blocks and it is discussed in the later section. Figure 1.5 shows the procedure of repeating the hash computation by concatenating the nonce value [9] with candidate block. Assume "Hello, World" is the candidate block data in the figure 1.5 and 0,1,...4250 are the nonces for understanding. Target assigned is 2^{240}.

The condition of PoW is satisfied when the nonce 4250 is concatenated with "Hello, World". In the block header, nonce size is of 4 bytes than the maximum value it can support is $2^{32}-1$. There may be a possibility that numbers from 0 to $2^{32}-1$ cannot not produce the hash less than the target and no point in again repeating the same nonce because it will produce the same result because the SHA256 hash algorithm is deterministic, which produces same output for an input even it is repeated infinite times. In such case, Proof of Work cannot be achieved. Hence the option of extraNonce and timestamp update is included in the mining. An extraNonce can be included and that will be placed in the left node of the merkle tree not on the block header. In such case, hashMerkleRoot in candidate block gets changed and as consequence candidate block is changed. Now repeating the same nonce with modified candidate block will give different hash values because input is different. Also, if required timestamp of the block can be updated to update the candidate block data to produce the new hash output with same nonce series. These two options should be preferred only after all possible nonces are tried with the first made candidate block.
How the timestamp updated? At the time of making candidate block, miner uses their system linux timestamp and start computing the hash with different nonce. There is a time change when the nonces are exhausted. Latest timestamp will be included in the candidate block.

Multiple Blocks: There is a possibility, multiple miners can successfully mine the same numbered block. For example, assume miners are working on top of block number 4 to mine 5^{th} block. There is a possibility that two miners $(\mathcal{A}, \mathcal{B})$ mined the 5^{th} block but only one block can take part in the main chain. Both miners broadcast blocks to their neighbors for propagation in the network. In this case, some of the miners will work on top of the $\mathcal{A}$s 5^{th} block and other set of miners will work on top of the $\mathcal{B}$s 5^{th} block. The 6^{th} block will decide which 5^{th} block will be considered for the main chain because the longest chain is the main chain. If the miners produce 6^{th} block from miner $\mathcal{A}$s 5^{th} block then $\mathcal{A}$s block will be part of the main chain and $\mathcal{B}$s block will go as stale block. However, there is a possibility that 6^{th} block for both $\mathcal{A}$ and $\mathcal{B}$s 5^{th} can be produced at same time or with valid delay. In such case, the main chain cannot be decided. The main chain can be decided based on the 7^{th} block. Likewise it will move forward to decide the main chain and side chain. According to the live bitcoin experience, the success of block will be decided only after next 7 blocks on top of it. This is the reason no transaction can be implemented before 7 blocks build on top of the block

Proof of Work computation:
"Hello, world!0" => 1312af178c253f84028d480a6adc1e25e81caa44c749ec81976192e2ec934c64 = 2$\hat{2}$52.253458683
"Hello, world!1" => e9afc424b79e4f6ab42d99c81156d3a17228d6e1eef4139be78e948a9332a7d8 = $2^{255.868431117}$
"Hello, world!2" => ae37343a357a8297591625e7134cbea22f5928be8ca2a32aa475cf05fd4266b7 = $2^{255.444730341}$
....
"Hello, world!4248" => 6e110d98b388e77e9c6f042ac6b497cec46660deef75a55ebc7cfdf65cc0b965 = $2^{254.782233115}$
"Hello, world!4249" => c004190b822f1669cac8dc37e761cb73652e7832fb814565702245cf26ebb9e6 = $2^{255.585082774}$
"Hello, world!4250" => 0000c3af42fc31103f1fdc0151fa747ff87349a4714df7cc52ea464e12dcd4e9 = $2^{239.61238653}$

Figure 1.5: Proof of Work [9]

which includes the respective transaction.

1.2.6 Difficulty and Target

This is a measure to know how difficult to mine a block in bitcoin network. This get adjusted for every 2016 blocks (roughly two weeks called as epoch) based on the expected and actual time of blocks generation. Difficulty is to derive the target value, which is to prove, miner puts enough computational effort and time to mine the blocks. As discussed before, Miner should compute a SHA256 hash output and that should be less than or equal to the target value. The Difficulty and Target for an epoch is calculated using the equations 1.3 to 1.6 [19]. The unexpected changes in difficulty is controlled by allowing at most adjustment of a factor of 4 that is a new difficulty cannot be more than four times of old difficulty and not less than $\frac{1}{4}$th of old difficulty ($\frac{old_Difficulty}{4} \geq Difficulty \leq old_Difficulty * 4$).

$$\text{Difficulty} = \frac{Max\_Target}{Current\_Target} \tag{1.3}$$

Max_Target = 0X00000000FFFF0000000000000000000000000000000000000000000000000000

Each block stores an encoded representation of target (called "nBits") to minimize the storage requirement. For example, the maximum target is encoded as 0x1d00ffff in 32 bits instead of 256 bits. The decoding mechanism of 32 bits target to the actual 256 target is shown below.

0x00ffff * $2^{8*(0X1d-3)}$ = 0x00000000FFFF0000000000000000000000000000000000000000000000000000

The $Current\_Target$ that is the target for next 2016 block will be calculated using equation 1.4 and 1.6. The $old\_Difficulty$ is the difficulty of the last 2016 blocks. Initial 2016 blocks target is are computed using the Deviation in equation 1.4 is ratio between $Expected\_Time$ to mine 2016 blocks and $Actual\_Time$ time taken to mine [47]. As defined, 10 minutes are required to mine a bitcoin but it varies. Considering 10 minutes per block, expected time is 20160 minutes for 2016 blocks.

$$\text{Deviation} = \frac{Expected\_Time}{Actual\_Time} \tag{1.4}$$

$$\text{Difficulty} = Deviation * old\_Difficulty \tag{1.5}$$

$$\text{Current\_Target} = \frac{Max\_Target}{Difficulty} \tag{1.6}$$

Example: If each block in an average takes 5 minutes then to mine 2016 blocks it will take 10080 minutes.

$\text{Deviation} = \frac{20160}{10080} = 2$

$\text{Difficulty} = 2 * 2^{200} = 2^{201}$

$\text{Current\_Target} = \frac{2^{224}}{2^{201}} = 8388608$

The miner should generate the SHA256 output less than 8388608 by taking input as the candidate block and nonce to successfully mine the block that is sha256(blockheader||nonce) $\leq$ 8388608. Initial target in the bitcoin genesis block is Max_Target.

1.2.7 Mining Incentive

Mining is the most important activity because it executes or approves the bitcoin transaction and generate blocks. Miner has to invest in infrastructure to achieve distributed consensus through Proof of Work based block mining. Miner cannot invest their own money for infrastructure and do voluntary job. Any blockchain application will be considered as dead once mining is stopped. Incentive for mining is the only option to keep the miners alive in the network. Incentives for a bitcoin miner are in two forms

- *Block Reward*: Miner will get reward for every successful block. It started with 50 BTC and it is halved every four years. Currently as of 15 February 2019, it is 12.5 BTC per block.
- *Transaction Fee*: Transaction fee is for the validation and execution of the transaction. Any miner can validate the transactions and add in the block but eventually only one miner will collect the fees whose block will be confirmed. The block will be considered as confirmed if it is part of the main or heavy weight chain. Transaction fee will be fixed by the sender of a coin. Transaction with high fee will confirm fast when compared to the transaction with less fee.

Miner should look into size of the transaction as well before taking transaction even fee is more. For example, assume an allowed block size is one million bytes, and there is a transaction ($\mathcal{A}$) with size 0.5 million bytes and transaction fee of 1BTC. Also, there are 100 transactions with size 1 KB and fee of 0.5BTC each. In this case, instead of taking $\mathcal{A}$, it is better to take other transactions for validation so that miner can get more transaction fee.

1.2.8 Bitcoin Wallet

Bitcoin Wallet is a digital file, where both Bitcoin address to receive bitcoins and the private key to send/spend bitcoins are stored [40].

$$Bitcoin\ Wallet = Private\ Key||Public\ Key/Address$$

We can categorize bitcoin users in two based on bitcoin wallet address

- Use the same address for all transaction - A user does not care about the privacy will go for same address for all transactions since it is easy to maintain the single key pairs and address.

- Use new address for every transactions - A user worries about financial privacy and does frequent transactions will generate new key pairs and use different address for each transaction.

Most of the bitcoin users are of second type uses new address for each transaction and back up the key pairs regularly. Since key pairs are used to derive the address, backup the key pair is sufficient. In the beginning, private/public key pair is generated in non-deterministic way that is key pairs are randomly generated and it should be backup each time you make a new pair of addresses. The process seems easy but it become more and more complicated and cumbersome to track/backup these days because of increase in number of private/public key pairs as the number of transaction increases for a bitcoin user.

Deterministic Wallet [4] is introduced to overcome the issues of handling multiple address. In the first version of deterministic wallet, key pairs to derive address are generated from a known starting *string* or *seed*. The private key is generated by computing the SHA256 hash of the *seed* added with an integer.

$$Private\ Key(PK_n) = \text{SHA256}(seed + n)$$

Where n is an ASCII-coded number that starts from 1 and increments as additional keys are needed. Since the addresses are generated in a known method rather than random method, users can conveniently create a single backup of the seed instead of backup all key pairs.

Gregory Maxwell [42] gave the idea of hierarchical deterministic wallets and many discussions about it. In Bitcoin Improvement Proposal (BIP) 32 [51], hierarchical deterministic wallet is introduced and implemented. Hierarchical Deterministic Wallet has the feature of master key derived securely from the *seed* using many rounds of SHA256. This allows the server to create as many public keys as is necessary for receiving funds, but a compromise of the master public key will not allow an attacker to spend from the wallet. However, compromise of master private key lead to loss.

1.2.8.0.1 Pseudo-Anonymity We can say Bitcoin is anonymous for the following reasons.

- Bitcoin components such as address, public and private key are the strings that do not have link with personal identity.

- An unique address for each transaction. Even, user identity for an address is disclosed, bitcoin account is anonymous because that address is valid only for the respective transaction.

However, Bitcoin is pseudo-anonymous because all transactions are plain and public, Internet Protocol Address analysis may disclose the identity, etc.

1.3 Ethereum

Ethereum is the implementation of cryptocurrency with an additional feature that is smart contract to run the decentralized applications. It takes less time to mine the blocks than Bitcoin even though it also uses Proof of Work (PoW). It uses Gas units to specify fee for every transactions and other operations. The size of a block is defined based on Gas limit. Like Bitcoin, Ethereum also does not preserve the privacy of the accounts that if addresses and amounts in the transactions are public. However, mapping of account holder and the account number (or) address is not public to achieve pseudoanonymity. Ethereum works based on the account balances and transaction unlike Bitcoin, which works only based on transaction.

1.3.1 How Ethereum works

Ethereum network is a peer to peer public network, which allows any individual or organization to take part or create account in it. All users of the Ethereum network need to possess unique network identity called address, which is derived from their public key. User wish to take part in the network has to generate their SECP-256k1 Elliptic Curve asymmetric key (private and public) pair. With key pair and ether (ETH, ethereum cryptocurrency), user can create a transaction, which is to send the ether from one account to another account. User created a transaction will broadcast to his/her peers, from the peer it go to the next peer eventually it will reach all users in the network. Similarly other users in the network broadcast their transactions. All transactions including their own transaction will be stored in the transaction pool of every user. Users fetch transactions from the pool, validate and include in the block if validation is successful. Users includes the transaction, solve the Proof of Work (PoW) problem and mine the block are called miners. The successfully mined block will be broadcast to the entire network and it will be validated by other users in the network and added to the blockchain if

the block is mined according to the ethereum requirement that is valid transaction, valid Proof of Work, valid timestamp, valid previous block hash, etc. The transaction will be considered as successful after inclusion in the block and take part in the main blockchain. The miner will get reward as well as transaction fee as a compensation for mining the block. Users wish to create their own decentralized application can create the account in Ethereum network and store their smart contract code, which will be validated and included in the blockchain by miners. The smart contract will automatically get executed when the condition in the contract satisfies. Ethereum uses gas to measures the amount of computational effort that it will take to execute certain operations. It restrict users from adding more transactions and performing infinite computations. The gas for each operation is predefined in the implementation for example

- 21000 for gas for a transaction
- 30 gases for SHA3 computation
- 32000 gases for all contract-creating transactions after the Homestead transition.

One gas is equal to 10^9 Wei or 10^{-9} Ether. Wei is the denomination like paisa for Indian Rupees. If user $\mathcal{A}$ makes transaction then equivalent gas value will be deducted from $\mathcal{A}$ account as transaction fee and deposited in the miner account. The transaction fee is calculated as $gaslimit * gasprice$. Users not having sufficient gas on his/her account cannot make the transaction in the ethereum network. However, for mining a block there is no need to have gas rather miner will get the transaction fee and mining reward on their account if the block is successful.

1.3.2 Address Construction

Ethereum provides 160 bits hexadecimal address for its members to uniquely identify them. Addresses are derived from the public key of a member using hash functions. Member interested to join the network should derive its key pair (Public and Private) by following the ECDSA of ethereum protocol. The key size should be 256 bits and 512 bits respectively for private and public key. The address of a member is derived using the 512 bits public key and Keccak256 hash algorithm [49]. In the following steps, construction of ethereum member address is defined.

1. $f_{Hash}(publickey) \rightarrow H$: Compute the Keccak256 hash of the public key of size 512 bits and get the output of size 256 bits.

2. $f_{trim}(H) \rightarrow address$: Take the rightmost 160 bits of the hash output H and convert to ASCII to get the address.

Even though there are many reasons, one of the reasons for choosing the last 40 characters instead of first 40 characters is that the chance for repetition of characters in first 40 is more compared to the last 40.

1.3.3 Transaction

Transaction is an important property for cryptocurrency. A transaction is a message that is sent from one account to another account. Message is of two types: one is message call and another is contract creation. Transaction orginator may differ from sender in the case when the message call or contract creation comes from the EVM code. Following are the parameters of a transaction.

- nonce: It is a scalar value represents number of transaction sent by the sender.

- gasPrice: A scalar value equal to Wei to be paid per unit of gas spent for this transaction. gasPrice will be set by the transaction owner. Its miners choice to choose the transaction for mining or not.

- gasLimit: A scalar value indicates the maximum amount of gas that should be used to execute the transaction. Gas value for each operation is pre-defined by the ethereum developer.

- to: The 160-bit address receiver.

- value: A scalar value equal to the number of Wei to be transferred to the recipient.

- v, r, s: Values of transaction signature to determine the sender.

- Additionally, transaction will have a contract creation attribute for contracts.

1.3.3.1 Transaction execution

Execution of a transaction will be done in two steps: Validity and Transfer or execution. Validity includes: (1) The transaction is well-formed RLP, with no additional trailing bytes; (2) the transaction signature is valid; (3) the transaction nonce is valid (equivalent to the sender accounts current nonce); (4) the gas provided is sufficient for the transaction; (5) the sender account has sufficient balance for transfer plus the gas cost. After the successful validation, the cost or amount (T_a) mentioned in the transaction will took place that is T_a will be deducted from the senders balance and added to the receiver balance. The gas price will be deducted from the senders balance and added to the miner balance in addition to reward. In case any one the validation from (1) to (5) fails then respective transactions will not be executed and included in the block. In such case, Transaction fee will not be deducted from the sender balance.

1.3.3.1.1 Transaction execution policy Transactions of an ethereum account holder will get executed sequentially not in random order and double spending is not possible because each transaction will have a nonce and the same nonce will be available in the senders account at stateRoot merkle tree node. Nonce is the transaction sequence number for an account. In case any malicious intend user sends more than one transaction with same nonce then it will be detected. For example, assume an account holder *Alice* have nonce A in her stateRoot node. She creates the transaction with nonce B and broadcast for mining. The process of transaction validation using A & B to avoid double spending and sequential execution of transaction is given in Algorithm 3.

There are peculiar situation for double spending such as

- In case, same transaction sent twice with different nonces it will be executed sequentially as genuine transaction accordingly balance of the sender will be deducted and both transaction will be in the blockchain.

- In case, same transaction sent twice with same nonce then any one will get executed and next will be discarded by the network.

Algorithm 3 Transaction Execution

Require: *A, B*
1: **if** $(B - A) == 1$ **then**
2: Transaction with nonce B will be executed
3: **else**
4: **if** $(B - A) > 1$ **then**
5: Transaction will be deferred till the prior transaction executed
6: **else**
7: Discard the transaction
8: **end if**
9: **end if**

1.3.4 Proof of Work

Ehtereum achieves distributed consensus through Proof-of-Work (PoW) [39]. The concept was invented by Cynthia Dwork and Moni Naor [35] to combat against the junk email. Later it is used for the distributed consensus. Ethash [20] latest version of Dagger-Hashimoto is the PoW algorithm use in Ethereum 1.0. This works based on seed value, 16MB cache for light node and 1GB data set for full node to successfully mine the block in the initial stage. Later size increases based on epoch that is cache and data set get updated for every 30000 blocks. Data set is a Directed Acyclic Graph to completely have unique value for each field of the data set.

1.3.4.1 Directed Acyclic Graph

Directed Acyclic Graph (DAG) in Graph theory is a finite directed graph without cycle. Figure 1.6 shows the representation of directed acyclic graph.

In ethereum implementation, DAG is represented in $n*16$ two dimensional array [21]. The value of n starts from 16777186 and keep on growing for every epoch that is for every 30000 blocks. The rows value are written sequentially into the file, with no delimiter between rows. For every epoch, completely new DAG will be generated with more nodes. Every ethereum full nodes need to spend time for DAG generation before mining the blocks. DAG can be pre-generated because it depends only on block height [28] [23]. Initially DAG was of 1GB size and grows approximately 8MB for every epoch because of its input parameters shown in 1.6 [20].

The process of generating DAG is through the following steps

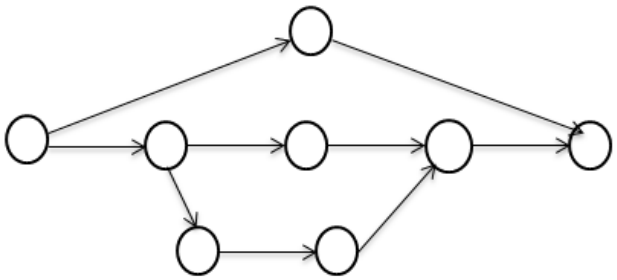

Figure 1.6: Directed Acyclic Graph

- Step 1: Cache and data size computation
- Step 2: Seed hash computation
- Step 3: Cache computation
- Step 4: Data set computation

1.3.4.1.1 Size Computation The size of the cache and data set DAG is decided using block number and other initial parameters listed in table 1.6. The algorithms 4 and 5 [20] shows the process of computing the size of the cache and DAG respectively. The size will vary based on the block height.

Algorithm 4 Cache Size Computation

Require: block_number
Ensure: size (sz)
1: sz = CACHE_BYTES_INIT + CACHE_BYTES_GROWTH * $\frac{block\_number}{EPOCH\_LENGTH}$
2: sz -= HASH_BYTES
3: while not isprime($\frac{sz}{HASH\_BYTES}$):
4: sz -= 2 * HASH_BYTES
5: return sz

1.3.4.1.2 Seed Computation The random seed is one of the parameters to generate the cache data. The seed hash is different for every epoch and it is depend on the block number. The seed hash for very first epoch is

Table 1.6: DAG generation parameters

Parameter with Value	***Description***
WORD_BYTES = 4	bytes in word
DATASET_BYTES_INIT = 2^{30}	bytes in dataset at genesis
DATASET_BYTES_GROWTH = 2^{23}	dataset growth per epoch
CACHE_BYTES_INIT = 2^{24}	bytes in cache at genesis
CACHE_BYTES_GROWTH = 2^{17}	cache growth per epoch
CACHE_MULTIPLIER=1024	Size of the DAG relative to the cache
EPOCH_LENGTH = 30000	blocks per epoch
MIX_BYTES = 128	width of mix
HASH_BYTES = 64	hash length in bytes
DATASET_PARENTS = 256	number of parents of each dataset element
CACHE_ROUNDS = 3	number of rounds in cache production
ACCESSES = 64	number of accesses in hashimoto loop

Algorithm 5 Data Size Computation

Require: block_number
Ensure: size (sz)
1: sz = DATASET_BYTES_INIT + DATASET_BYTES_GROWTH * $\frac{block\_number}{EPOCH\_LENGTH}$
2: sz -= MIX_BYTES
3: while not isprime($\frac{sz}{MIX\_BYTES}$):
4: sz -= 2 * MIX_BYTES
5: return sz

calculated by taking Keccak-256 hash for 32 bytes of zeros and for other epochs Keccak-256 hash for the previous epoch seed hash as in equation 1.7. SH_i denotes seed hash for the epoch number i.

$$SH_i = \begin{cases} KECCAK - 256(0_{32}) & \text{If Epoch Number}(E_n) == 1 \\ KECCAK - 256(SH_i - 1) & \text{otherwise} \end{cases} \tag{1.7}$$

Sha3_256 in the Algorithm 6 for seed hash computation is the Keccak-256 hash function.

Algorithm 6 Seed Hash Computation

Require: block_number
Ensure: size (sz)
1: s = '0x00' * 32
2: **for** i in range($\frac{block.number}{EPOCH\_LENGTH}$): **do**
3: s = sha3_256(s)
4: **end for**
5: return s

1.3.4.1.3 Cache computation Using the seed and cache size, the initial cache will be generated using equation 1.8 till $i == 0$ *to* n, where $n = \frac{Cache\_Size}{HASH\_BYTES}$. It takes seed as the initial value for intermediate cache C_0 computation and iterate with C_{i-1} to generate C_i till n. Final cache is computed using the equation 1.9 with input C and CACHE_ROUNDS . The function rmh in equation 1.10 is the RandMemoHash algorithm [41] which takes arbitrary size input and produces 64 bytes output.

$$C_i = \begin{cases} KECCAK - 512(s) & \text{If } i == 0 \\ KECCAK - 512(C_i - 1) & \text{otherwise} \end{cases} \tag{1.8}$$

$$R(x = C, y = CACHE\_ROUNDS) = \begin{cases} x & \text{If } y == 0 \\ E_{RMH}(x) & \text{If } y == 1 \\ R(E_{RMH}(x), y - 1) & \text{otherwise} \end{cases}$$

(1.9)

$$E_{RMH}(x) = (E_{rmh}(x,0), E_{rmh}(x,1), ..., E_{rmh}(x,n1)) \tag{1.10}$$

Figure 1.7 shows the golang based pseudo code to generate the cache data using equation 1.8 to 1.10.

```
def mkcache(cache_size, seed):
    n = cache_size / HASH_BYTES
    # Sequentially produce the initial dataset
    o = [sha3_512(seed)]
    for i in range(1, n):
        o.append(sha3_512(o[-1]))
    # Use a low-round version of randmemohash
    for _ in range(CACHE_ROUNDS):
        for i in range(n):
            v = o[i][0] % n
            o[i] = sha3_512(map(xor, o[(i-1+n) % n], o[v]))
    return o
```

Figure 1.7: Pseudocode for Cache computation

1.3.4.1.4 Data set computation Using every pieces of cache, data set will be computed to support the mining process. Equation 1.11 ot 1.13 shows the method of computing the data. It takes the input as cache (c) and the index (i) and computes the data by applying the KEC512 or Fowler/Noll/Vo (FNV) hash functions on the pseudorandomly selected cache nodes . The variable c_{size} is the size of cache and parameters $DATASET\_PARENTS$, $HASH\_BYTES$ and $WORD\_BYTES$ are pre-defined.

$$E_{datasetitem}(c,i) = E_{parents}(c,i,1,\varnothing) \tag{1.11}$$

$$E_{parents}(c,i,p,m) = \begin{cases} E_{parents}(c,i,p+1,E_{mix}(m,c,i,p+1)) & \text{if } p < DATASET\_PARENTS - 2 \\ E_{mix}(m,c,i,p+1) & otherwise \end{cases}$$

(1.12)

$$E_{mix}(c,i,p,m) = \begin{cases} KEC512(c[i \bmod c_{size}] \oplus i) & \text{if } p = 0 \\ E_{FNV}(m, c[E_{FNV}(i \oplus p, m[p \bmod \frac{HASH\_BYTES}{WORD\_BYTES}]) \bmod c_{size}] & otherwise \end{cases} \tag{1.13}$$

Pseudo code to compute the full data set using the cache and equations 1.11 ot 1.13 is shown in figure 1.8 [20].

```
def calc_dataset_item(cache, i):
        n = len(cache)
        r = HASH_BYTES / WORD_BYTES
        # initialize the mix
        mix = copy.copy(cache[i mod n])
        mix[0] ⊕ = i
        mix = sha3_512(mix)
        # fnv it with a lot of random cache nodes based on i
        for j in range(DATASET_PARENTS):
                cache_index = fnv(i⊕j, mix[j mod r])
                mix = map(fnv, mix, cache[cache_index mod n])
        return sha3_512(mix)

def calc_dataset(full_size, cache):
        return [calc_dataset_item(cache, i) for i in range full_size / HASH_BYTES
```

Figure 1.8: Pseudocode for Data computation

1.3.4.2 Mixhash Generation

Miner has to compute the mixhash to successfully mine the block and prove to the network that enough computational power is spent on mining the block. Miner uses full dataset (DAG) to produce mixhash for a particular header and nonce. If the result is less than the computed target then nonce is valid for the block otherwise generate new random nonce and compute the mixhash and result. It will go till the valid nonce is identified. Equations

1.14 to 1.17 are used to generate the mix hash. The parameter, d is the data set, m is the mix hash (initially it is empty), s is the seed hash that is sha3_512(header+revert(nonce)) and i is the iteration index.

$$E_{accesses}(d,m,s,i) = \begin{cases} E_{mixdataset}(d,m,s,i) & \text{if } i = ACCESSES - 2 \\ E_{accesses}(E_{mixdataset}(d,m,s,i),s,i+1) & otherwise \end{cases} \tag{1.14}$$

$$E_{mixdataset(d,m,s,i)} = E_{FNV}(m, E_{newdata}(d,m,s,i) \tag{1.15}$$

$$E_{newdata}(d,m,s,i)[j] = d[E_{FNV}(i \oplus s[0], m[imod\frac{MIX\_BYTES}{WORD\_BYTES})] \\ mod\frac{dsize/HASH_BYTES}{n_{mix}}.n_{mix} + j] \\ \forall j < n_{mix} \tag{1.16}$$

$$E_{compress}(d,m,s,i) = \begin{cases} m & \text{if } i > \| \ m \ \| \ 8 \\ E_{compress}(E_{FNV}(E_{FNV}(E_{FNV}(m[i+4], & \\ m[i+5]), m[i+6]), m[i+7]), i+8) & otherwise \end{cases} \tag{1.17}$$

Figure **??** and 1.9 has the pseudo code to compute the mixdigest for the full node and light node using equation 1.14 to 1.17. If the result less than the target, then it is considered as the valid nonce and it will be attached with the block otherwise redo the process. Non mining full node can use the same mix hash generation technique for the respective header and nonce then validates the proof. Light nodes can generate the respective part of data on the fly using available cache and validate the results.

1.3.4.3 Double Buffer of DAG

The constructed DAG will be stored in the main memory buffer for quick access while mining the block. After an epoch, new DAG need to be generated for mining the block. Having one buffer model will delay mining till

```
def hashimoto(header, nonce, full_size, dataset_lookup):
    n = full_size / HASH_BYTES
    w = MIX_BYTES // WORD_BYTES
    mixhashes = MIX_BYTES / HASH_BYTES
    # combine header+nonce into a 64 byte seed
    s = sha3_512(header + nonce[::-1])
    # start the mix with replicated s
    mix = []
    for _ in range(MIX_BYTES / HASH_BYTES):
        mix.extend(s)
    # mix in random dataset nodes
    for i in range(ACCESSES):
        p = fnv(i ŝ[0], mix[i % w]) % (n // mixhashes) * mixhashes
        newdata = []
        for j in range(MIX_BYTES / HASH_BYTES):
            newdata.extend(dataset_lookup(p + j))
        mix = map(fnv, mix, newdata)
    # compress mix
    cmix = []
    for i in range(0, len(mix), 4):
        cmix.append(fnv(fnv(fnv(mix[i], mix[i+1]),mix[i+2]),
mix[i+3]))
    return {
        "mix digest": serialize_hash(cmix),
        "result": serialize_hash(sha3_256(s+cmix))
    }

def hashimoto_light(full_size, cache, header, nonce):
    return hashimoto(header, nonce, full_size, lambda x:
calc_dataset_item(cache, x))

def hashimoto_full(full_size, dataset, header, nonce):
    return hashimoto(header, nonce, full_size, lambda x: dataset[x])
```

Figure 1.9: Pseudocode for Mix Hash Generation

new DAG created and replace the old in buffer. It will give periodic delay in the blockchain growth, which may bring the centralization thus 51% hash power attack rises. Double buffering is the solution to overcome the periodic delay. DAG used for the current epoch X_1 will be in buffer *B-1* and the pre-generated next epoch DAG X_2 in another buffer *B-2*. Once the epoch X_1 is over, the DAG in *B-2* for epoch X_2 can be used for mining without any delay. The DAG for next epoch X_3 will be pre-generated and stored in buffer *B-1*. Buffers will be used in cycle to avoid periodic delay in mining. [18]

1.3.5 Difficulty computation

Ehtereum uses Proof of Work (PoW) similar to bitcoin for the distributed consensus. Miner should compute hash value by taking mixhash value and partial block (without block hash, nonce and mixhash value). The hash output should be less than or equal to the target value, which is computed using the timestamp (parent timestamp *p_timestamp* and block timestamp *b_timestamp*), parent block difficulty (*parent_d*) and block number. Parent block is the block which is prior to the newly mining block. Equation 1 and 2 shows the computation of difficulty under frontier and homestead version respectively [49].

$$block\_d = parent\_d + \frac{parent\_d}{2048} * \begin{cases} 1 & b\_timestamp - p\_timestamp < 13 \\ -1 & \text{otherwise} \end{cases} \tag{1.18}$$

$$block\_d = parent\_d + \frac{parent\_d}{2048} * max(1 - \frac{b\_timestamp - p\_timestamp}{10}, -99) \underset{+int(2^{\frac{blocknumber}{100000}} - 2)}{} \tag{1.19}$$

The time difference between latest or presently mined block and its parent block are compared with 13 in equation 1.18. The difficulty value goes up if the time difference is smaller than 13 otherwise difficulty goes down. In equation 1.18, the difference how far off from 13 seconds was not taken into consideration. A block mined in a second after the previous one has the

same effect on the difficulty as one mined after 12 seconds. Also, 13 seconds and more will be considered as same in equation 1.18. To overcome this issue, homestead version of difficulty computation given in equation 1.19 is introduced on which for every 10 seconds difference up to a certain limit will be weighted separately. The genesis block and the lowest difficulty in ethereum is 131072 which is a power of 2 and magic number does not have sufficient proof for the usage.

In the Byzantium version of ethereum [50], uncle block headers contribute to the difficulty computation. Equation 1.20 shows Byzantium version of difficulty computation. The value of y is 1, if no uncle header in the latest block otherwise it is 2. The denominator 10 from 9 will ensures the same block time even, it should decrease by 3% given the current uncle rate of 7%. [33]

$$block\_d = parent\_d + \frac{parent\_d}{2048} * max(\underset{+int(2^{\frac{blocknumber}{100000}}-2)}{y} - \frac{b\_timestamp - p\_timestamp}{9}, -99) \tag{1.20}$$

The difficulty bomb is introduced in the second part of the equation, $int(2^{\frac{block.number}{100000}-2})$. This will happen after 100000 number of blocks. The difficult will go very high once this part of the equations contributes in computing the difficulty value will. It will lead to delay in mining. This is kept for fork the chain to another version of the implementation. The target of a block is computed using the equation 1.21.

$$target = \frac{2^{256}}{difficulty} \tag{1.21}$$

Combining 64-bit nonce and the 256 bit mix-hash compute a 256 bits hash value that is n. The value n should be less than or equal to $2^{\frac{256}{blockdiff}}$ otherwise choose another nonce and repeat the process. This value n shows to the network that the miner spent sufficient amount of effort to mine this block. It is called as Proof of Work (PoW) in ethereum.

1.3.6 Block

Block of the ethereum blockchain contains three parts (B_H, B_T, B_U). The B_H is the block header, B_T is the list of transactions and B_U is the list of

Uncle blocks headers.

1.3.6.1 Block Header

The header stores meta-information and data part stores actual data such as account details, transaction details, etc. The block header includes the following attributes

- parentHash: It is a 256-bit hash (output of SHA256) of the parent blocks header, including nonce and mix digest.
- ommersHash: It is a 256-bit hash of the ommers (uncle) list of the block.
- beneficiary: It is a 160-bit address of the block miner who will receive the reward and transaction fee.
- stateRoot: It is 256-bit hash of the root node of the stateRoot trie, after all transactions are executed and updated on the trie.
- transactionsRoot: It is 256-bit hash of the root node of the transactionRoot trie which includes all transactions of the block.
- receiptsRoot: It is 256-bit hash of the root node of the receiptsRoot trie which includes all transactions receipt of the block.
- logsBloom: It is 2048 bits Bloom filter composed from indexable information (logger address and log topics) contained in each log entry from the receipt of each transaction in the transactions list.
- difficulty: It is a scalar value corresponding to the difficulty level of this block.
- number: It is a scalar value equal to the number of ancestor blocks. The genesis block has a number zero.
- gasLimit: It is a scalar value equal to the current maximum limit of gas expenditure per block.
- gasUsed: It is a scalar value equal to the total gas spent in transactions in the block

- timestamp: Unixs timestamp of block inception.
- extraData: It is an arbitrary ≤ 32 byte array containing data relevant to this block.
- mixHash: It is a 256 bit hash which together with the nonce proves that a sufficient amount of computation has been carried out on this block.
- nonce: It is a 64 bit hash which together with the mix-hash proves that a sufficient amount of computation has been carried out on this block.
- hash: The Keccak 256-bit hash of its own header in entirety that is including nonce and mix hash.

1.3.6.2 Receipt structure

The transaction receipt which is prepared after the transaction execution contains four items $< R_u, R_l, R_b, R_z >$.

- R_u is the cumulative gas used by the block till the execution of the respective transaction. For example, if three transactions are included in the block then the second transaction receipt will have the gas up to execution of the second transaction not the third one but third transaction receipt will have the cumulative gas which includes gas of all transactions.
- R_l is the set of logs created when the execution of respective transaction. It is a sequence of log entries like $O_1, O_2, O_3, ...$ Evey O contains, O_a loggers address, O_t 32-byte log topics and O_d some bytes of data.
- R_b is the bloom filter created out of the logs R_l. Bloom filter function will be called for each logger address O_a and its topics O_t. This reduce a series of log to 2048 bit (256 byte) value using equation 1.30.

$$M(O) = \bigvee_{x \in \{O_a\} \cup O_t} (M_{3:2048}(x)) \tag{1.22}$$

$M_{3:2048}$ is a bloom filter that sets three bits of 2048. Three bits are chosen based on the lower order 11 bits of first three pairs (0,1; 2,3;

and 4,5) of KEC(x). We consider a bloom filter b sequence of 2048 bits and initially it is all zeros. $\bigvee$ is the greatest element of elements, means out of $O_a \cup O_t$, it will choose the greatest element and compute the bloom filter. However, in goethereum implementation all values are used for the bloom filter computation as shown in algorithm 8.

Example, consider $KEC(x)$ output is as follows, where x is an log topic or address

(**00101010**11100000**00000010**00000010**00101011**11100000.....).

The lower order 11 bits of first pair of bytes are (01011100000)

$p = (0 \times 2^{2042} + 1 \times 2^{2041} + 0 \times 2^{2040} + 1 \times 2^{2039} + ... + 0 \times 2^{2032})\, mod\, 2018$

The p^{th} position in the 256 byte sequence will be set. This will be repeated for the second and third pair of bytes. In yellow paper [49], instead of 11 bits all 16 bits of two bytes are considered to find the location. Algorithm 7 and 8 shows the computation of bloom filter for the logs.

Algorithm 7 Bloom Filter

Require: $O \rightarrow \{O_1, O_2, O_3, ...\}$
1: Set $Y \leftarrow 0_{2048}$
2: **for** $i = 1\ to\ \parallel O \parallel$ **do**
3: $Y = M(O_{ai}, Y)$ #{O_{ai} is logger address of i^{th} log}
4: **for** $j = 1\ to\ \parallel O_{ti} \parallel$ **do** {#O_{ti} is i^{th} log topics set}
5: $Y = M(O_{tj}, Y)$ #{O_{tj} is j^{th} topic of i^{th} log}
6: **end for**
7: **end for**

Algorithm 8 $M(X, Y)$

Require: X, Y
1: **for** $i = 0, 2, 4$ **do** {# only for 0,2,4 loop iterates}
2: $p = KEC[X][i, i+1]\ mod\ 2048$
3: $Y_p = 1$
4: **end for**
5: return Y

The topics for logs are created based on the input parameters of a transaction. The parameters required are $(Sender_{address}, Receiver_{address}, Value)$.

Topics (T) are created as follows.
$T_1 = sha256(Sender_{address}, Receiver_{address}, Value)$
$T_2 = sha256(Sender_{address})$
$T_3 = sha256(Receiver_{address})$

- R_z is the status code (success, failure, expired, etc.) of the transaction. It is an non negative integer.

1.3.7 Mining Incentive

Like Bitcoin Miner, ethereum miner also invest in their infrastructure for successful mining. Miner is the one who secure the network by creating, verifying, propagating blocks and validates the transactions. In physical currency, the organization formed by the Government of the country will print the currency. In case of cryptocurrency, currencies are originated by the miners. At the time of mining the block, miners themself will add defined currency in their account. Ethereum started it mining Ether from its Frontier release. On the initial version Olympics testnet, the Frontier pre-release, the ether mined have no value. The variants of ethereum [22] are given in table 1.7

Table 1.7: Ethereum Version

Version	**Code Name**	**Release Date**
0	Olympic	May, 2015
1	Frontier	30 July 2015
2	Homestead	14 March 2016
3	Metropolis (vByzantium)	16 October 2017
3.5	Metropolis (vConstantinople)	to be announced
4	Serenity	to be announced

As discussed in Bitcoin, miner should be encouraged through incentives to secure and run the ethereum application. The successful block miner will get the following incentives or rewards.

- Block Reward - A static block reward of defined ether. (at initial stage it was 5Ether, as on 14/11/2018 it is 3 Ether and it will change in future.)

- Transaction fee - It is calculated based on the gas consumed for the operations of the transactions. Gas price derived from consumed gas is credited to the miner's account as part of the consensus protocol. It is expected, in future it may stop the block reward.

- Uncle Block Reward - An extra reward for including Uncle blocks as part of the block.

The mined block will be considered as successful once it is part of the main chain irrespective of block mining. If a miner mines the block and it does not take part in the main chain then he/she get discouraged and stop mining at one stage. This may bring down network security that is centralization and node with more hash power will lead the application. The Greedy Heaviest Observed Subtree (GHOST) protocol [46] is introduced by Yonatan Sompolinsky and Aviv Zohar in December 2013 to overcome centralization issue. The intention of this protocol is to encourage the miners by giving rewards to unsuccessful block i.e, Uncle block. Uncle blocks are the stale blocks which will be part of the chain but not the main chain. These blocks are part of the blockchain to ensure security and called as *Uncle* block or *Ommers* Block. Uncle block miners and the miner includes Uncle block (B_U) are eligible for reward in ethereum. Real block miner (RB_M) will get the block reward and additionally $\frac{1}{32}$ of the block reward (i.e, 3.125% of the block reward) for inclusion of each Uncle block as in equation 1.23. Transaction fee is not included in the 1.23. However, number of Uncle block included in one real block should not be greater than two and same uncle block cannot be added more than once.

$$Block\,Reward\,for\,B_M = (1 + \frac{Number\,of\,Uncle\,Blocks}{32}) * Reward \quad (1.23)$$

The reward policy for the Uncle block miner UB_M is given in equation 1.24, where U_i and RB_i are block number of Uncle block and Real block respectively. The maximum difference can be six and same numbered Uncle block is not allowed to include in the Real Block.

$$Block\,Reward\,for\,B_U = (1 + \frac{1}{8}(U_i - RB_i)) * Reward \quad (1.24)$$

1.3.8 Merkle Patricia Trie

Ethereum uses modified Merkle Patricia trie [24] to store the data securely and efficiently. Patricia is practical algorithm to retrieve information coded in alpha numeric. A node in a Merkle Patricia trie of ethereum is one of the following:

- *Branch*: A 17-items node, where first 16 items corresponds to the Hexadecimal values (0..F) and the 17^{th} item is used as a terminator node and the respective search key being ended at this point in its traversal.
- *Leaf*: It is two items structure, where first is for the key suffix and second is for the value.
- *Extension*: It is two items structure, where first is for the key part of size greater than one and shared by atleast two distinct keys. Second item is the pointer to the branch node

Key refers the search index that is ethereum address. For an example, assume a User $\mathcal{A}$ address is $ca6d4fdde714fd979de3edf0f5aaa9716b898ec8$. This is the key on the trie for $\mathcal{A}$'s account node. Suffix may be $e8$ and key part may be $5aaa9716$.

1.3.8.1 Tries in Ethereum

Ethereum uses merkle patricia tries to store the account details, transaction details and receipts details.

- *stateRoot Trie*: This is to hold account information. The trie of one block may get link to the part of a trie in another block as shown in figure 1.10 for saving memory space. Indexing and path of the trie is sha3(ethereumAddress) and a value is in rlp(ethereumAccount). The ethereum account is a four item tuple $<$*nonce, Balance, storageRoot and CodeHash*$>$. Nonce is number of transaction done by an account, Balance is to keep the account balance, CodeHash is for the smart contract and storageRoot is the pointer to the another similar node in the patricia tree.
  - Storage trie is the embedded trie for the stateRoot trie, where all contract data lives. Every account will have a separate storage trie.

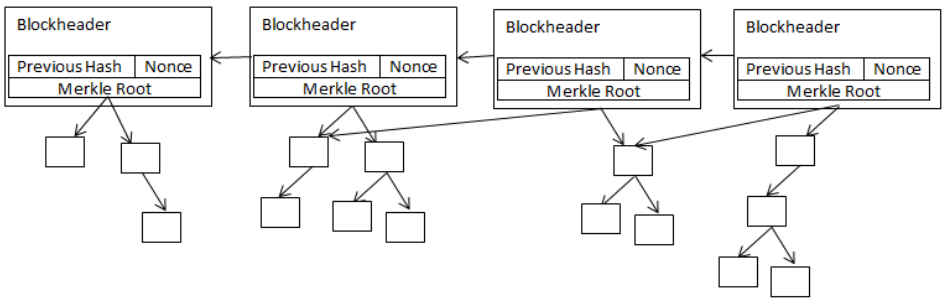

Figure 1.10: Abstract Ethereum Blockchain

- *transactionsRoot*: This is to store the transaction details. Since there is no relation between a transaction in one block and a transaction in another block, there is a separate transactions trie for every block. Indexing and path of the trie is rlp(transactionIndex). The transactionIndex is the index of a mined transaction. One block can have more than one transaction but limited to the gas. Transactions cannot be modified once it is mined.

- *receiptsRoot*: This is to store Receipts of a transaction. Indexing and path is rlp(transactionIndex). The trie ordering for the transaction and receipt will be same since it follows the same parameter for indexing. Receipts cannot be modified once the respective block is mined.

1.3.9 Recursive Length Prefix

1.3.9.1 Encoding

Ethereum uses the Recursive Length Prefix (RLP) for the purpose of serializing the arbitrary bytes [26] [49] [15]. The RLP encoding in ethereum is for the following two items.

- String item that is byte array: "cat"

- List of items: ["puppy","cow"]

Encoding on the items will be different for string and list. Equation 1.25

categorize the item and use 1.26 or 1.28 respectively.

$$RLP(x) = \begin{cases} R_b(x) & if\ x\ belongs\ to\ Byte\ array \\ R_l(x) & otherwise \end{cases} \tag{1.25}$$

$$R_b(x) = \begin{cases} x & if \parallel x \parallel = 1 \wedge x[0] < 128 \\ (128 + \parallel x \parallel).\, x & else\ if \parallel x \parallel < 56 \\ (183 + \parallel BE(\parallel x \parallel) \parallel).\, BE(\parallel x \parallel).x & otherwise \end{cases} \tag{1.26}$$

Equation 1.26 is for byte array encoding. In equation, x is the byte array and $x[0]$ is the first character. Dot operator performs sequence concatenation. The RLP encoding or serialization procedure for the byte array are as follows using equation 1.26

- If the input contains only a single byte of size less than 128, then the input is exactly the output. Example - The string "d" = ['d'], each character is 8 bits and it will be represented in hexadecimal as [0x64].
- If the input contains lesser than 56 bytes, then the output is input prefixed by the byte equal to the length of the input byte array plus 128. Example - hello world = [0x8b, 0x68, 0x65, 0x6c, 0x6c, 0x6f, 0x20, 0x77, 0x6f, 0x72, 0x6c, 0x64]
- Otherwise that is length upto 256^8, the output is equal to the input that is (1) prefixed by length of the input byte array in big endian format that is (2), which is itself prefixed by the length of (2) plus 183 that is (3). This has the limit that is size of (2) cannot be more than 8 and number of bytes on the byte array cannot be more than 256^8. Example - A string with 5000 'a' characters, so the encoding of aaa... = [0xb9, 0x13, 0x88, 0x61, 0x61,...]

BE in the equation 1.26 and 1.28 is to expand a non-negative integer value to a big-endian byte array using equation 1.27.

$$BE(x) = (b_0, b_1, ...) : b_0 \neq 0 \wedge \sum_{n<\|b\|}^{n=0} [b_n.256^{\|b\|-1-n}] \tag{1.27}$$

Big Endian Example: If we take the input of 5000 "a" which is 5000 bytes, the length of it is ($\| \ aaaaa... \ \|$) is 5000 and Big Endian $BE(5000) = 19136$ (19 and 136) that is $19 \times 256^1 + 136 \times 256^0$, where $b_0 = 19$ and $b_1 = 136$ the length $\| \ BE(\| \ x \ \|) \ \|$ is 2. The value 256 is the base for BE conversion.

The RLP encoding or serialization procedure for the list is similar to the serialization of byte-array using equation 1.28 and 1.29

$$R_l(x) = \begin{cases} (192+ \| \ s(x) \ \|) \, .s(x) & if \| \ s(x) \ \| < 56 \\ (247+ \| \ BE(\| \ s(x) \ \|) \ \|) \, .BE(\| \ s(x) \ \|) \, .s(x) & otherwise \end{cases} \tag{1.28}$$

$$s(x) = RLP(x_0) \, .RLP(x_1)... ==> R_b(x_0) \, .R_b(x_1)... \tag{1.29}$$

- If the input list is less than 56 bytes in length, then the output is equal to that list prefixed by the byte equal to the length of this byte array list plus 192. Example - [*hello*, *world*] = [0xcc, 0x85, 0x68, 0x65, 0x6c, 0x6c, 0x6f, 0x85, 0x77, 0x6f, 0x72, 0x6c, 0x64]. First '0xcc' refers base that is 192 plus length of upcoming bytes that is 204 $\rightarrow$ 0xc0 plus 0x0c (length of input that is 10 for *helloworld*) and its RLP (value:2 for mentioning two byte array size, 0x85 and 0x85 as per equation 1.26). Next for each byte array, RLP encoding is applied according to the rules in equation 1.26. The first representation '0x85' (128 $\rightarrow$ 0x80) plus 0x05 refers to number of bytes in that array that is five for *hello*. The second representation 0x85 for the next byte array *world*.

- Otherwise, the output is equal to the input list that is (1) prefixed by the length of the input byte array in big endian form that is (2), which is itself prefixed by the length of (2) plus 247 that is (3). Example - ["a", "a" upto 100 times]= [0xf8, 0x64, 0x61, 0x61, 0x61, ...]. The hex "0x64" is the length of the list that is 100 and no need to add RLP prefix because the byte array size is 1 for all 100 and value of the string is less than 128. The hex "0x61", the ASCII value of "a". The hex 0xf8 is derived from the base 247 $\rightarrow$ 0xf7 plus length of $BE(100)$ that is 1, where $b_0 = 100$ because 100×256^0 and no b_1 or above.

The scalar value will be specified as byte array in the form of Big Endian format. In case of empty string and list, it will be 128 and 192 respectively. However, there is no specific encoding format for signed or floating-point values.

1.3.9.2 Decoding

RLP decoding is the inverse of the encoding scheme. The string interpretation decision will be based on the first byte of RLP encoded output.

- If the first byte is less than 128 and that is the only byte then input is output.
- If the first byte is between 128 and 182 then output starts from second byte and length is less than 56 byte.
- If first byte(1) is more than 183, then the output starts after ((1) - 183) bytes. Similarly for the list item.
- If an expected fragment is decoded as a scalar and leading zeroes are found in the byte sequence, clients are required to consider it non-canonical and treat it in the same manner otherwise discard it as invalid RLP data.

1.3.10 Chain Replacement

Every blocks in the chain will have the total difficulty value to prevent the chain replacement. One or group of member of the network may create the false chain alternate to the real chain and try to replace it. As we discussed before, longest chain will be consider as main chain. For example, assume a user $\mathcal{A}$ has the chain with length 5000 and user $\mathcal{B}$ has the chain with length 5500. In this case, main chain will be the one with user $\mathcal{B}$. User $\mathcal{A}$ has to replace its chain with the chain of user $\mathcal{B}$. This is applicable only when the genesis block of the both chain is same. If user $\mathcal{B}$ is malicious and like to replace the real chain with false chain then it need to have more computational power than the network total users computational power otherwise it is not possible. Assume a malicious user creates a longest chain than the real chain with low difficulty value as shown in figure 1.11 and try to replace the real chain. This will not be successful because the network user will replace the

chain only when the total difficulty of the new chain block is greater than the total difficulty in the present chain last block. According to figure 1.11, user $\mathcal{A}$ will check the total difficulty of the 301 block of the false chain with real chain that is its own chain's last block that is 300. The total difficulty in block 300 is greater than the 301 block in false chain of user $\mathcal{B}$. Hence it will not be replaced and attack will not be successful.

$\mathcal{A}$: Real Chain $\Rightarrow 1:2000 \leftarrow 2:2500 \leftarrow 3:3000 \leftarrow .... \leftarrow 300:250000$
$\mathcal{B}$: False Chain $\Rightarrow 1:2000 \leftarrow 2:2000 \leftarrow 3:2000 \leftarrow .... \leftarrow 500:2000$

Figure 1.11: Sample Chains (Block Number: Total Difficulty Value)

1.3.11 Contract Creation

Smart contracts are account holding objects on the ethereum blockchain. It is a piece of software that contains rules and regulations to interact with other contracts, make decisions, store data, and send ether to others. It automatically verifies the contract and then executes the agreed upon terms. Coding and executing smart contracts on the Ethereum blockchain makes them immutable and independent from centralization. This is a decentralize application (DApp). The contract creation is to create a new account with its associated code referred as Ethereum Virtual Machine (EVM) code generally called as smart contract. To create a contract, the parameters required are sender (s), original transactor (o), available gas (g), gas price (p), endowment (v) together with an arbitrary length byte array, i, the initialisation EVM code and the present depth of the message call/contract-creation stack (e) since EVM is stack based architecture.

The contract creation function C_f take state σ and above listed parameters as inputs and produces the new state σ', remaining gas g' and transaction substate A.

$$(\sigma', g', A) = C_f(\sigma, s, o, g, p, v, i, e) \tag{1.30}$$

The address of new account will be created using equation 1.31 by passing two parameters: sender address (s_a) and its current nonce (s_n) in the WorldState that is stateRoot trie. The s_a and $s_n - 1$ will be concatenated, RLP encoded

and last 160 bits of KEC256 hash is the address of account. $s_n - 1$ instead of s_n is because the nonce get increased for this contract before the creation.

$$a = B_{96..255}(KEC(RLP((s, \sigma[s]n - 1)))) \tag{1.31}$$

The structure of the newly created state includes account address, nonce (initially it is zero), balance, storageRoot(will be empty till the next account created), and the code hash of the empty string. It will be updated once the code is generated. The sender's balance that is sender's main account (or) externally owned accounts will be reduced with the value v and the balance of the account created for contract will be increased with value v.

Also, there are information collected after the transaction execution called as transaction substate A and it is given in equation 1.32.

$$A = (A_s, A_l, A_t, A_r) \tag{1.32}$$

The first parameter of the tuple is self-destruct set A_s. Second A_l is the log series for tracking contract-calls. Third, A_t is the set of touched accounts during transaction, of which the empty ones are deleted at the end of a transaction. A_r is the refund balance including the remaining gas value and *self-destruct* reward.

1.3.11.1 Contract Deletion

The only possible way to remove the contract code from the blockchain is through *self-destruct* operation on which a contract on that address performs the deletion of contract account. The balance Ether or Wei at that address along with the reward for *self-destruct* will go to a designated target and then the storage and code is removed from the stateRoot trie [17]. There are chances, no reward will be provided for the service when no Ether or Wei.

1.3.12 Peer Identification

Ethereum is the global network and identifying the peer is complex. A node join to the network need to find its trusted peer for sharing the blocks and transactions. The goethereum implementation finds peers through the discovery protocol. In the discovery protocol, nodes are gossipping with each other to find out about other nodes on the network. In order to get the peers

at the initial stage, it uses a set of bootstrap nodes whose endpoints are hard coded in the source code. To change the default bootnodes, user can use the following command on startup of the node.

>geth --bootnodes enode://pubkey1@ip1:port1, enode://pubkey2@ip2:port2

1.3.13 Genesis Block

In ethereum, the genesis is the very first block, which is numbered as zero. The users have to use the same genesis block to connect into the respective network. Genesis block has the following items as per the block header
$(0_{256}; 0_{160}; stateRoot; 0; 0; 0_{2048}; 2^{17}; 0; 0; 3141592; time; 0; 0_{256}; Kec((42)); (); ())$

0_{256} is the parent hash, which is sequence of zeroes, 0_{160} is the beneficiary or miner address, which is sequence of zeroes of size 160 bits, stateRoot is the hash of the stateRoot trie root node (initially the stateRoot trie is depend on the genesis file), next two zeros for transaction root and receipt root, 0_{2048} is the logsbloom value, which is sequence of zeros, 2^{17} is the initial difficulty value of the blockchain, next two zeroes for block number and gas limit, 3141592 is the gas used for this block, time is the linux timestamp, next zero for extra data, 0_{256} is the sequence of zeroes for the mix hash, $KEC((42))$ is the Keccak hash of a one byte array of value 42, for nonce, next two empty () refers list of uncle block headers and series of transactions index. The genesis block of goethereum is shown in figure 1.12

1.3.13.1 goEthereum

The goethereum [5] is the implementation of ethereum protocol using the Go language [38]. This tool can be installed on your machine using Internet sources. It additionally needs packages like ldb (nosql database) required for the blockchain storage. Users can use this tool to run the private network to test.

Once downloaded and installed, try the commands in figure 1.13 after opening the folder in terminal to compile *geth*. Once it is successfully compiled, move to *build/bin* folder to start *geth*.

Figure 1.14 shows the command to initialize the genesis file. It includes *networkid*, which makes the *geth* node to connect to the respective network. It has *datadir*, which is to provide data directory to store all data related

Genesis Block:

{
difficulty: 17179869184,
extraData: ”0x11bbe8db4e347b4e8c937c1c8370e4b5ed33adb3db69cbdb7a38e1e
50b1b82fa”,
gasLimit: 5000,
gasUsed: 0,
hash: ”0x6b319f89979dc640f83445a0595ebf0db06abf0b8ceaeea62139e0f5a48a5f5a”,
logsBloom: ”0x00000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000”,
miner: ”0x0000000000000000000000000000000000000000”,
mixHash: ”0x000000000000000000000000000000000000000000000000000000000
0000000”,
nonce: ”0x0000000000000042”,
number: 0,
parentHash: ”0x0000000000000000000000000000000000000000000000000000000
000000000”,
receiptsRoot: ”0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e
363b421”,
sha3Uncles: ”0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd
40d49347”,
size: 573,
stateRoot: ”0xd7f8974fb5ac78d9ac099b9ad5018bedc2ce0a72dad1827a1709da305
80f0544”,
timestamp: 0,
totalDifficulty: 17179869184,
transactions: [],
transactionsRoot: ”0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc00162
2fb5e363b421”,
uncles: [],
}

Figure 1.12: Sample Genesis Block

Initializing genesis file:
goethereum: $ make geth
goethereum: $ cd build/bin
goethereum-bin: $

Figure 1.13: goethereum - Compilation Command

to the respective node. The *port* is to run the geth node on a particular port of the machine. Duos *datadir* and *port* are required when your run multiple nodes on the same machine for private test network. If you run only one node on a machine then no need to provide the *datadir* and *port*, automatically it will be taken as per the goethereum protocol. The parameter *init genesis.json* is to initialize the genesis file. It is required to have *genesis.json* file on the bin folder. It may be copied from internet source. Nodes like to connect to the network should have the same genesis file otherwise it connection cannot be done.

Initializing genesis file:
$./geth –networkid=”121” –datadir private22333 –port 22333 init genesis.json

Figure 1.14: goethereum - Genesis file initalization Command

Figure 1.15 shows the command to create an account. Command should have the parameter *account new* as the additional similar to the genesis file initialization command. This command will look for the passphrase, which will be the secret to encrypt the node private key. This passphrase is required to unlock the account before making transactions and whenever required by the ethereum protocol. After the execution of command, an externally owned account address will be created for the user. If we run *geth* command in Figure 1.16 without account creation then error will thrown at the time of mining as no *coinbase*. The *coinbase* is the address on which the benefits such as reward, transaction fee, etc. to be credited.

Figure 1.16 shows the command to run *geth* and open in console mode to execute the *geth* commands. Figure 1.17 shows the command to make the transaction. It includes sender address $eth.accounts[0]$ and recipient address $eth.accounts[1]$. In the example, address $eth.accounts[0]$ sends 40 ether to address $eth.accounts[1]$. The function $web3.toWei$, will convert the

Creating account:
$./geth –networkid="121" –datadir private22333 –port 22333 account new

Figure 1.15: goethereum - Account creation Command

Ether to Wei. The gas given for the transaction is 30000. The command *eth.accounts*[0] will give the hexadecimal address of the account which is in 0^{th} position similarly for the 1^{st} and others. Instead of giving *eth.accounts*[0], directly user hexadecimal address can be provided.

Start Ethereum with console:
$./geth –networkid="121" –datadir private22333 –port 22333 console

Figure 1.16: goethereum - Start node with console

sendTransaction:
>eth.sendTransaction({from:eth.accounts[0],to:eth.accounts[1], value:web3.toWei(40,"ether"), gas:30000})

Figure 1.17: goethereum - Transaction Command

Figure 1.18 shows the command to get the balance of an external account address *eth.accounts*[0] and unlock the account *eth.accounts*[0] by providing the passphrase. The *geth* command with *admin.addPeer* is to connect to another peer in the network. The parameters are public key, ip address and port of the remote peer node ($enode://pubkey@ip:port$).

1.4 ZeroCash

ZeroCash is used for the implementation of Zcash cryptocurrency [32]. ZeroCash [30] is the first popular privacy preserving model implemented with non interactive zero knowledge proof. In non zero knowledge proof based cryptocurrency, if user *Alice* want to prove to *Bob* that she owns 30 coin then *Alice* will simply sign a message using the key that controls the coin and send to *Bob*. This leaks the information regarding owner of the coin. In

Get Balance:
>eth.getBalance(eth.accounts[0])
Unlock Account:
>personal.unlockaccount(eth.accounts[0])
Connect to Peers:
> admin.addPeer("enode://f4642fa65af50cfdea8fa7414a5def7bb79
91478b768e296f5e4a54e8b995de102e0ceae2e826f293c481b5325f89be6d
207b003382e18a8ecba66fbaf6416c0@172.31.2.43:22331")

Figure 1.18: goethereum - Other Commands

zero-knowledge proof, same goal will be achieved without leaking knowledge of who is owning the coin that is *Bob* only knows some secret key is owning the coin but not who owns it. A cryptocurrency implementation to achieve decentralized anonymous payment should ensured the following properties.

- *No Information Reveal*: No information related to transaction should be revealed to the public.
- *Coin Spending*: Only owner of a coin has to spend the coin others should not.
- *Public Verifiability*: Any user in the network should validate the transaction.
- *No Double Spend*: No owner should double spend the coin. This can be ensured using the serial number *sn*.

Zerocash uses zero-knowledge Succinct Non-interactive ARguments of Knowledge (zk-SNARKs) [31] proof for privacy preserving transaction and overcomes the following issues of Zerocoin [43], which also use zero knowledge proof for privacy.

- Performance - Zerocoin uses double-discrete-logarithm proofs of knowledge which takes in average 450 ms to verify the proof if the key size is 128 bits. Proof will be verified by each and every node in the network before accepting the transaction.
- Anonymity - Zerocoin achieves anonymity by unlinking the transaction with sender's address but it does not hide the metadata (recipient address, etc) and values in the transaction.

- No Denomination of coins - It does not support to divide the coin and pay. The value for each coin is fixed.

ZeroCash uses lightweight zk-SNARKs to solve the above listed issues. zk-SNARKs is a non-interactive zero knowledge short proofs and it can be verifiable in short time. In addition to performance enhancement, zerocash achieves the following features.

- *User anonymity with fixed-value coins*: Similar to zerocoin, same value is considered for all coins. Owner of the coin generate a random number r, a serial number sn and compute $cm = COMM_r(sn)$. The new coin is made as $c = (r, sn, cm)$. A transaction tx_{mint} including cm is sent to the ledger. To spend the transaction, user has to prove that it knows r such that $cm = COMM_r(sn)$ can be generated.

- *Extending coins for direct anonymous payments*: In the above process, Serial Number sn will leak the transaction information such as sender get to know who is the recipient also there is a possibility sender may double spend if he does same coin transaction before the recipient. Zerocash uses pseudorandom function to generate the serial number and it vary for every further transfer of coins. It has the support for many to one transaction and one to many transaction.

- *Sending Coins*: ZeroCash uses random numbers and it is required to be transferred to the recipient for further transfer . To enable the secure random number sharing, zerocash introduces two address key pairs ($addr_{pk}$,$addr_{sk}$), where $addr_{pk} = (a_{pk}, pk_{enc}) and addr_{sk} = (a_{sk}, sk_{enc})$. Using pk_{enc}, random numbers and other secret parameters will be shared with the recipient along with transaction pour. Recipient having the respective sk_{enc} can decrypt and use the random numbers for future transaction. To verify the transaction, secret parameters are not required.

- *Public Output*: Zerocash has the facility to make the coin as public to spend in other cryptocurrency.

The mint transaction in Zerocash allows the user to convert a non-anonymous coins (may be bitcoin from some Bitcoin address) into the same number of zerocoins to a specified Zerocash address. The pour transactions allows user to

make the private payment [7] that is transacting a coin from one to another address. The decentralized anonymous payment scheme(DAP scheme) of ZeroCash uses the following functions to make the anonymous transactions.

- *Setup*: Trusted third party generates a list of public parameters by taking the security parameter. Trusted third party is used only once to generate the public parameters afterwards trusted party is not needed.
- *CreateAddress*: User generates the address key pair $(addr_{pk}, addr_{sk})$ using the public parameter computed from the previous function. The public key $addr_{pk}$ is published, while the secret key $addr_{sk}$ is used to redeem coins sent to $addr_{pk}$.
- *Minting Coins*: To mint a coin, user will use the public parameters, coin value $\{0, 1, ..., v_{max}\}$ and destination address public key $addr_{pk}$.
- *Pouring coins*: It transfers value from input coins into new output coins, marking the input coins as consumed. It uses public parameters, old coin owner secret address $addr_{sk}$, merkle tree root with its authentication path, new public address $addr_{sk}$ and transaction fee to generate the new coin.
- Verifying Transactions: Verifier uses the public parameter, minted or poured transaction and ledger to verify the transaction. The ledger is a sequence of transactions and is append-only. At any given time, a unique valid snapshot of the currencys ledger is available to all users.
- Receiving Coins:User scans the ledger and retrieves unspent coins paid to a particular user address using the address key pair.

1.5 Exercises

1. Perform the RLP decoding of [0x8b, 0x68, 0x65, 0x6c, 0x6c, 0x6f, 0x20, 0x77, 0x6f, 0x72, 0x6c, 0x64]
2. Perform the RLP encoding of [”my name”,[”ethereum”,”bitcoin”],”mining”]
3. What is the ethereum reward for the miner who includes two uncle blocks?

4. What is the ethereum reward for the miner of the uncle block no.1000 when it gets added in the real block no.1004?

5. Compute the frontier block difficulty, where block_timestamp is 61100014 and parent_timestamp is 61100004, parent_difficulty is 2222342.

6. Compute the Homestead block difficulty, where block_timestamp is 61100014 and parent_timestamp is 61100004, parent_difficulty is 2222342 and block number is 100002.

7. What is the transaction fee for a miner in ethereum, if she/he successfully mines the block.

8. What type of bitcoin address are the following

   - $1BvCDSEYstWetqLPn5Au4m4GFg7xJaRVS2$
   - $3J98t1EpFZ73LNmRvietrnpiPrnqRhMSTy$
   - $mF1tAaz6x1HUXrCNLbt6Dqcw6o5GNn6xqX$

9. Compute the bitcoin epoch difficulty, where actual time for 2016 blocks is 20000 minutes and old_difficulty is 2222342.

10. What is the bitcoin value after 2000 block considering 50 bitcoin reward?

11. What is the difference between Zerocash and Bitcoin?

12. How the balance of an address in bitcoin can be computed?

13. Consider a bitcoin transaction with value 3BTC and transaction fee as 4BTC. What type of transaction it is?

14. Compute the Cache size of ethereum for *block_number* is 15000. Can the miner user 2GB GPU to mine the 15001 block?

15. How much memory space is required for the light node in bitcoin when the number of blocks in the chain are 80000?

16. What is the problem with the frotier difficulty computation? Will it show any difference in difficulty if timestamp difference is 1 or 12.

17. Assuming that the total hash power of the network stays constant, what is the probability that a block will be found in the next 10 minutes?
18. Assuming that the total hash power of the network stays constant, what is the probability that a block will be found in the next 10 minutes?
19. Derive the construction cost of merkle tree.
20. What factors affect the rate of orphan blocks? Can you derive a formula for the rate based on these factors?
21. How do you prove Membership using Merkle tree?
22. What is the probability, the attacker can replace the chain, if he is z blocks behind the honest chain?

1.6 Lab Practices

1. Practice the Python program 1.19 in the lab for cache size computation.
2. Practice the Python program 1.20 in the lab for cache size computation.
3. Write the golang program to compute the difficulty of 500000 block in ethereum.
4. Write the golang program to implement the RLP encoding as discussed in 1.3.9.
5. Write a smart contract for simple voting system with GUI and run in goethereum private test network.

```
#!/usr/bin/env python
def is_prime_number(x):
    if x >= 2:
        for y in range(2,x):
            if not ( x % y ):
                return False
    else:
        return False
    return True
block_number = 1
epoch = 30000
C_I = 2**24
C_G = 2**17
c = block_number/epoch
C_sz = C_I + C_G * c
C_sz = C_sz - 64
while not(is_prime_number(C_sz/64)):
     C_sz = C_sz - 128
print ("C_sz=",C_sz)
```

Figure 1.19: Cache size Computation in python

```
#!/bin/env python
def is_prime_number(x):
    if x >= 2:
        for y in range(2,x):
            if not ( x % y ):
                return False
    else:
        return False
    return True
block_number = 1
epoch = 30000
D_I = 2**30
D_G = 2**23
c = block_number/epoch
D_sz = D_I + D_G * c
D_sz = D_sz - 128
while not(is_prime_number(D_sz/128)):
     D_sz = D_sz - 256
print ("D_sz=",D_sz)
```

Figure 1.20: Data size Computation in python

Bibliography

[1] Block. https://en.bitcoin.it/wiki/Block. [Online; accessed 30-November-2018].

[2] Block hashing algorithm. https://en.bitcoin.it/wiki/Block_hashing_algorithm. [Online; accessed 30-November-2018].

[3] Block timestamp. https://en.bitcoin.it/wiki/Block_timestamp/. [Online; accessed 06-December-2018].

[4] Deterministic wallet. https://en.bitcoin.it/wiki/Deterministic_wallet. [Online; accessed 16-January-2019].

[5] goethereum. https://github.com/ethereum/go-ethereum. [Online; accessed 02-December-2018].

[6] Hashcash. https://en.bitcoin.it/wiki/Hashcash. [Online; accessed 30-November-2018].

[7] How zerocash works. http://zerocash-project.org/how_zerocash_works.html. [Online; accessed 07-December-2018].

[8] Orphan block.

[9] Proof of work. https://en.bitcoin.it/wiki/Proof_of_work. [Online; accessed 30-November-2018].

[10] Technical background of version 1 bitcoin addresses.

[11] What is wif?

[12] How was the magic network id value chosen? https://bitcoin.stackexchange.com/questions/2337/how-was-the-magic-network-id-value-chosen/2355#2355, 2011. [Online; accessed 30-November-2018].

[13] Coinbase. https://en.bitcoin.it/wiki/Coinbase, 2015. [Online; accessed 01-December-2018].

[14] Transaction. https://en.bitcoin.it/wiki/Transaction, 2015. [Online; accessed 29-November-2018].

[15] Data structure in ethereum — episode 1: Recursive length prefix (rlp) encoding/decoding. https://medium.com/coinmonks/data-structure-in-ethereum-episode-1-recursive-length-prefix-rlp-encoding 2016. [Online; accessed 18-November-2018].

[16] Script. https://en.bitcoin.it/wiki/Script, 2016. [Online; accessed 24-December-2018].

[17] Introduction to smart contracts. https://solidity.readthedocs.io/en/v0.4.21/introduction-to-smart-contracts.html, 2017. [Online; accessed 22-November-2018].

[18] Dagger hashimoto. https://github.com/ethereum/wiki/wiki/Dagger-Hashimoto, 2018. [Online; accessed 08-November-2018].

[19] Difficulty. https://en.bitcoin.it/wiki/Difficulty, 2018. [Online; accessed 28-November-2018].

[20] Ethash. https://github.com/ethereum/wiki/wiki/Ethash, 2018. [Online; accessed 08-November-2018].

[21] Ethash DAG. https://github.com/ethereum/wiki/wiki/Ethash-DAG, 2018. [Online; accessed 08-November-2018].

[22] Ethereum. https://en.wikipedia.org/wiki/Ethereum, 2018. [Online; accessed 08-November-2018].

[23] Mining. https://github.com/ethereum/wiki/wiki/Mining#so-what-is-mining-anyway, 2018. [Online; accessed 08-November-2018].

[24] Patricia Tree. `https://github.com/ethereum/wiki/wiki/Patricia-Tree`, 2018. [Online; accessed 08-November-2018].

[25] Protocol rules. `https://en.bitcoin.it/wiki/Protocol_rules`, 2018. [Online; accessed 24-December-2018].

[26] Recursive Length Prefix (RLP). `https://github.com/ethereum/wiki/wiki/RLP`, 2018. [Online; accessed 08-November-2018].

[27] Txid: The hash of a transaction's data. `http://learnmeabitcoin.com/glossary/txid`, 2018. [Online; accessed 13-December-2018].

[28] What actually is a dag? `https://ethereum.stackexchange.com/questions/1993/what-actually-is-a-dag`, 2018. [Online; accessed 08-November-2018].

[29] Antonopoulos, A. M. Mining and consensus. `https://www.oreilly.com/library/view/mastering-bitcoin/9781491902639/ch08.html`, 2011. [Online; accessed 16-January-2019].

[30] Ben-Sasson, E., Chiesa, A., Garman, C., Green, M., Miers, I., Tromer, E., and Virza, M. Zerocash: Decentralized anonymous payments from bitcoin. `http://zerocash-project.org/media/pdf/zerocash-extended-20140518.pdf`, 2014. [Online; accessed 07-December-2018].

[31] Ben-Sasson, E., Chiesa, A., Tromer, E., and Virza, M. Succinct non-interactive zero knowledge for a von neumann architecture. `https://eprint.iacr.org/2013/879.pdf`, 2015. [].

[32] Bowe, S., Hopwood, D., Hornby, T., and Wilcox, N. Zcash protocol specification- version 2.0-alpha-1. `https://pdfs.semanticscholar.org/4363/fc884880a826aa64a65aa15de9f7c07f458e.pdf`, 2016. [Online; accessed 07-December-2018].

[33] Buterin, V. Eip-100: Change difficulty adjustment to target mean block time including uncles. *Yellow paper* (2016).

[34] Delgado-Segura, S., Pérez-Solà, C., Navarro-Arribas, G., and Herrera-Joancomartí, J. Analysis of the bitcoin utxo set.

https://eprint.iacr.org/2017/1095.pdf, 2015. [Online; accessed 01-December-2018].

[35] Dwork, C., and Naor, M. Pricing via processing, or, combatting junk mail, advances in cryptology. *CRYPTO92: Lecture Notes in Computer Science No. 740, Springer: 139147* (1993).

[36] Frankenfield, J. Consensus mechanism (cryptocurrency). https://www.investopedia.com/terms/c/consensus-mechanism-cryptocurrency.asp, 2018. [Online; accessed 25-January-2019].

[37] Giechaskiel, I., Cremers, C., and Rasmussen, K. B. On bitcoin security in the presence of broken crypto primitives. https://eprint.iacr.org/2016/167.pdf, 2016. [Online; accessed 24-December-2018].

[38] Griesemer, R., Pike, R., and Thompson, K. go - the go programming language. https://golang.org/. [Online; accessed 01-December-2018].

[39] Jakobsson, M., and Juels, A. Proofs of work and bread pudding protocols. *Communications and Multimedia Security, Kluwer Academic Publishers: 258272* (1999).

[40] Khatwani, S. What are hd wallets? (deterministic wallet). https://coinsutra.com/hd-wallets-deterministic-wallet/, 2018. [Online; accessed 16-January-2019].

[41] Lerner, S. D. Strict memory hard hashing functions. http://www.hashcash.org/papers/memohash.pdf, 2014. [Online; accessed 05-December-2018].

[42] Maxwell, G. Deterministic wallets. https://bitcointalk.org/index.php?topic=19137.msg239768#msg239768, 2011. [Online; accessed 16-January-2019].

[43] Miers, I., Garman, C., Green, M., and Rubin, A. D. Zerocoin: Anonymous distributed e-cash from bitcoin. http://spar.isi.jhu.edu/~mgreen/ZerocoinOakland.pdf, 2015. [].

[44] Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf, 2008. [Online; accessed 29-November-2018].

[45] Porat, A., Pratap, A., Shah, P., and Adkar, V. Blockchain consensus: An analysis of proof-of-work and its applications. http://www.scs.stanford.edu/17au-cs244b/labs/projects/porat_pratap_shah_adkar.pdf. [Online; accessed 21-December-2018].

[46] Sompolinsky, Y., and Zohar, A. Secure high-rate transaction processing in bitcoin. https://eprint.iacr.org/2013/881.pdf, 2013.

[47] Walker, G. Difficulty a mechanism for regulating the time it takes to mine a block. http://learnmeabitcoin.com/guide/difficulty, 2015. [Online; accessed 28-November-2018].

[48] Walker, G. Public key: A unique number mathematically generated from a public key. http://learnmeabitcoin.com/glossary/public-key, 2018. [Online; accessed 26-November-2018].

[49] Wood, G. Ethereum: A secure decentralised generalised transaction ledger eip-150 revision. *Yellow paper* (2014).

[50] Wood, G. Ethereum: A secure decentralised generalised transaction ledger- byzantium version 12779ac. *Yellow paper* (2018).

[51] Wuille, P. Hierarchical deterministic wallets. https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki#Specification_Key_derivation, 2012. [Online; accessed 16-January-2019].

Contents

Chapter 1

Introduction

There are applications other than cryptocurrency look for the properties of blockchain such as immutable data storage, decentralization, permitting non-trusted users, etc. to ensure the security of the data and environment. However, difficulties of blockchain implementation such as computational power and storage are the hurdle for the users to implement in the application especially in lightweight environment. Considering the problem, there are different implementation proposed following the blockchain properties. This section discuss about some of such implementations. This chapter is divided into two sections

- IOTA - The first section of the chapter discusses about the IOTA implementation and the possible attacks with solutions
- Algorand - The second section introduces the Algorand concept
- Corda - The third section discuss about the concept and its implementation.

1.1 IOTA

The popular blockchain implementations such as Bitcoin and Ethereum has the drawback of transaction fee for transactions of any value. The micropayments in future will be more because of the growth of IoT industry, and paying a transaction fee that is higher than the transaction amount seems illogical. Also, in Blockchain implementations, it is not easy to get rid of

transaction fees because that is the incentive for miners of the blocks. IOTA is the cryptocurrency that does not incorporate Blockchain Technology and transaction fee. It is specifically designed for the Internet of Things (IoT) industry [26]. The IOTA cryptocurrency is referred as mIOTA. In IOTA, Directed Acyclic Graph (DAG) called as *tangle* is used instead of the global blockchain. The transactions (called as site) issued by nodes (or users) make a site set of the tangle graph, which is the ledger for storing transactions and other messages. Following is the process to form the tangle.

- When a new transaction arrives to the network, it must approve k previous transactions. In the implementation k is 2.
- Approvals are represented by directed edges in the graph.

No transaction fee or incentive to approve the transactions because of two reasons.

- Computational cost of validating and approving the transaction is low.
- Our transactions will be approved only when we approve the existing transactions.

Hence, every node will approve the existing transaction without any transaction fee or incentive. Also nodes can propagate the transactions without looking for any transaction fee and even if it does not issue transactions because lazy nodes will be dropped by its neighbor. Every node in the network computes a statistic that is how many new transactions are received from a neighbor. In case a particular node is 'too lazy' that is not sharing transactions then it will be dropped by its neighbors. Therefore, even if a node does not issue transactions, and hence has no direct incentive to share new transactions with others, it still has incentive to participate in the network.

For example, assume *Alice* is in the IOTA and not making any transactions. According to the IOTA protocol, all nodes in the network has to share the new transactions of others and their own with neighbors like gossip protocol. The following two questions can rise since there is no incentive.

(a) Why should *Alice* approve others transaction?

(b) Why should *Alice* propagate others transactions to neighbors?

Answers are as follows for the above questions.

(a) Answer: *Alice* transaction will be approved only when she approves the existing transactions otherwise her transaction will be pending without confirmation

(b) Answer: *Alice* should propagate the transactions regularly otherwise she will not be part of the network that is her neighbors will drop her from theirs neighbor list. Whether she make the transactions or not but to be active member of the network she has to propagate the transactions.

Now the next question arise that is what is the issue if neighbors drop her?*Alice* will not get the growing tangle and when she want to make, approve and propagate the transaction then she has to look for active node of the network to help in finding the existing transactions to approve and propagate her transactions to the network.

1.1.1 Priliminaries

IOTA is a peer to peer network and it was first conceptualized in 2014 and later founded in 2015 by David Snsteb, Sergey Ivancheglo, Dominik Schiener, and Dr. Serguei Popov [20]. The cryptocurrency was initially released on 13 June 2017 and the maximum IOTA token is 2779530283277761 [4]. IOTA allows any user or node to join the network like Bitcoin and Ethereum. It has the genesis transaction instead of genesis block in blockchain implementation. The genesis transaction is the first transaction of the tangle created by one of the founders and has an address with a balance that contained all of the tokens. The genesis transaction sent these tokens to several other 'founder' addresses. All of the tokens were created in the genesis transaction and no tokens can be created in the future. A user wish to join the IOTA network has to run the IOTA software and connect to the neighbors through the IP address. User can create and issue the transaction after creating the seed, private key and public address through a pre-defined protocol discussed in the later sections. If any user wish to issue a transaction then he/she has to approve or validate other transactions available in the tangle and his transactions will be approved by other users. Transactions issued to the tangle are called as transaction bundle, which contains set of transactions. Transactions are validated in two directions: one is transaction data validation that means Proof of Work (PoW), signature, etc. and second is the transaction value and user account balance availability validation. Transactions will

be considered as confirmed when majority of the future transactions or coordinator transactions approved it directly or indirectly. In regular interval, coordinator of IOTA issue a transaction to show the valid tangle to all users of the network. IOTA users are also allowed to send message transactions that is a transaction without IOTA coin but having messages. Following are the important notations with its description used for the IOTA.

- *Transaction Bundle*: It is a set of transactions related to each other.
- *Site*: It is a transaction bundle represented on the tangle graph.
- *Node*: The network is composed of nodes that is a device or user that issue and validate transactions.
- *Edge*: The edge in the DAG (tangle) from a site $\mathcal{A}$ to site $\mathcal{B}$ means the transaction bundle (site) $\mathcal{A}$ approves the transaction bundle (site) $\mathcal{B}$.
- *Weight*: The weight assigned to each transaction. In the implementation, each transaction or transaction bundle has the equal weight that is one. In figure 1.1, bottom right values are the weight. For example, the site X has weight 3.
- *Cumulative Weight*: It is the weight of a particular transaction plus the sum of weights of all transactions that directly or indirectly approve this transaction. In figure 1.1, the top left value is the cumulative weight which is its own weight plus the directly or indirectly approved transaction weight. For example, the two sites approved by X has the cumulative weight 4 that is its own weight 1 plus X weight 3.
- *Tip*: Transactions that are not yet approved by any other transactions in the tangle. In figure 1.1, the site X is the tip which does not have any approver.
- *Height*: The length of the longest path to the genesis transaction. For example, in figure 1.2, the height of a site C is 4, F is 2, D is 3 and G is 1.
- *Depth*: The length of the longest reverse-oriented path to some tip. For example, in figure 1.2, the depth of a site E is 1, D is 2 and F is 3.

- *Score*: The score of a transaction is the sum of weights of all transactions approved by this transaction plus the own weight of the transaction itself. In figure 1.2, the score of a site C is 7 because of its own weight 1 plus the approved sites weight *D(1), F(3), G(1) and E(1).*
- *Hashing*: IOTA uses Keccak-384 hash function for computing the hash. The Keccak-384 hash function used for generating address, key pairs and signing transactions. The Keccak-384 hash function in IoTA is called as *Kerl*. In the beginning, IOTA used its own hash function called Curl but in July 2017, the vulnerability was reported by researchers from Boston University and Massachusetts Institute of Technology. Hence moved to Keccak-384 hash.

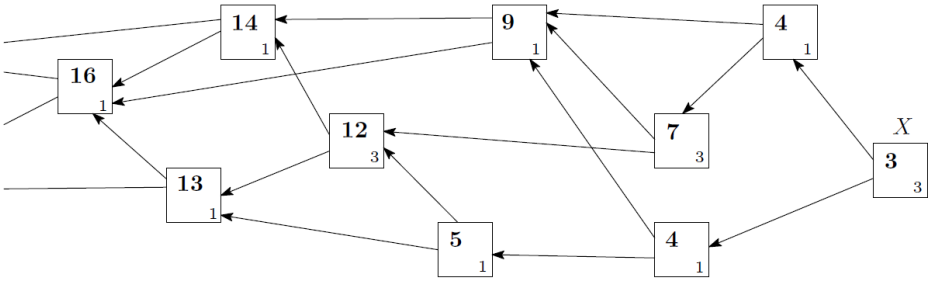

Figure 1.1: A sample Tangle

1.1.1.1 Encoding

The IOTA uses balanced trinary (trits) encoding scheme. Trinary is the three bit system containing $0, 1, 2$ (unbalanced) and $-1, 0, 1$ (balanced). IOTA uses the balanced trinary encoding representation named as Trit (**Tr**inary Dig**it**) and Tryte refers **Tr**inary B**tye**. One tryte is 3 trits that is $3^3 = 27$ values possible for a tryte. Equation 1.1 computes the decimal value for the trits $(1, 0, -1)$.

$$-8 = 1 \times 3^0 + 0 \times 3^1 + (-1) \times 3^2 \tag{1.1}$$

The possible values in balanced trytes are between -13 and 13 and total combination it can support is 27. To make the trytes human understandable,

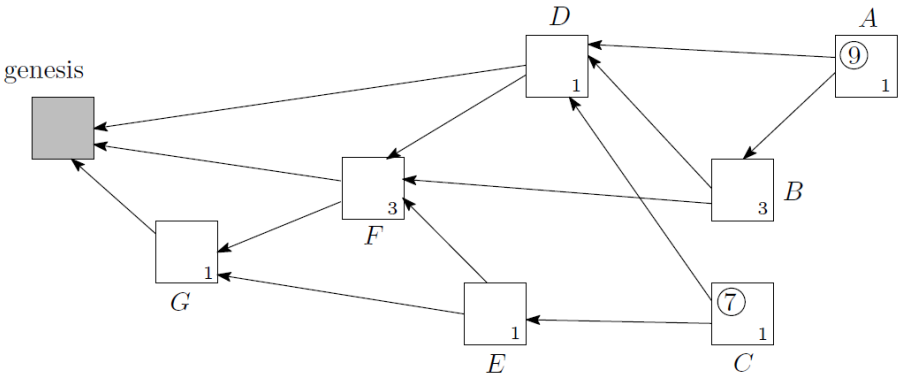

Figure 1.2: A sample Tangle

IoTA team created the 27 tryte characters those are alphabets A to Z and integer 9. Table 1.1 shows the trits and equivalent tryte alphabet and decimal value.

1.1.1.2 Winternitz signature

It is a hash based one time signature. In literature this proposal appears first in Merkle's thesis [18]. Merkle writes that the method was suggested to him by Winternitz in 1979 as a generalization of the Merkle OTS (One Time Signature) also described in [18]. It uses a one way function $f : \{0,1\}^n \rightarrow \{0,1\}^n$ and cryptographic hash function $g : \{0,1\}^* \rightarrow \{0,1\}^n$.

A Winternitz parameter $w \geq 2$ is the number of bits to be signed simultaneously. The process of computing key pair, signature generation and verification is respectively shown in algorithm 1, 2 and 3. To compute the signature σ, secret key X is hashed k times, where k is computed from the input message. To verify the signature, compute k from the message and derive r $(r = n - k)$, where n is the number of times X is hashed to bring the verification key Y. The signature σ will be hashed remaining r times to derive Y, which is the public key. If the receiver able to derive the public key from signature then it is valid otherwise it is invalid. IOTA uses the Winternitz algorithm for transaction signature.

Table 1.1: Conversion Table (Trytes to Decimal)

Tryte-encoded character	**Trits**	**Decimal number**
9	0, 0, 0	0
A	1, 0, 0	1
B	-1,1, 0	2
C	0, 1, 0	3
D	1, 1, 0	4
E	-1, -1,1	5
F	0, -1, 1	6
G	1, -1, 1	7
H	-1,0, 1	8
I	0, 0, 1	9
J	1, 0, 1	10
K	-1,1, 1	11
L	0, 1, 1	12
M	1, 1, 1	13
N	-1, -1, -1	-13
O	0, -1, -1	-12
P	1, -1, -1	-11
Q	-1,0, -1	-10
R	0, 0, -1	-9
S	1, 0, -1	-8
T	-1,1, -1	-7
U	0, 1, -1	-6
V	1, 1, -1	-5
W	-1, -1, 0	-4
X	0, -1, 0	-3
Y	1, -1, 0	-2
Z	-1, 0, 0	-1

Algorithm 1 Key Generation

Require: *w, n*
Ensure: *Signature Key X, Verification key Y*
1: $t_1 = \lceil \frac{n}{w} \rceil$
2: $t_2 = \lceil \frac{\lfloor log_2 t1 \rfloor + 1 + w}{w} \rceil$
3: $t = t_1 + t_2$
4: $X = (x_{t-1}, x_{t-2}, ...x_1, x_0) \in_R \{0,1\}^{(n,t)}$
5: $Y = (y_{t-1}, y_{t-2}, ...y_1, y_0) \in \{0,1\}^{(n,t)}$, where $y_i = f^{2^w-1}(x_i), 0 \leq i \leq t-1$

Algorithm 2 Signature Generation

Require: *Message M, Digest D = g(M), w, Signature key X,* t_1*,* t_2*, t, n*
Ensure: *Signature* σ *with size t.n*
1: Pad bits to make D divisible by w
2: Split D of $len(w)$ $D = b_{t-1} \parallel b_{t-2} \parallel ... \parallel b_{t-t_1}$ #{$\parallel$ is concatenation}
3: Bit strings b_i are identified with integers in $\{0, 1, ..., 2_{w-1}\}$
4: Checksum $C = \sum_{i=t-t_1}^{t-1}(2^w - b_i)$
5: Extended string C is split in to t_2 blocks of length w, $C = b_{t_2-1} \parallel b_{t_2-2} \parallel ... \parallel b_0$
6: $\sigma = f^{b_{t-1}}(x_{t-1}), ..., f^{b_1}(x_1), f^{b_0}(x_0))$

Algorithm 3 Signature Verification

Require: *Message M, Digest D = g(M), w, Verification key Y,* t_1*,* t_2*, t,* σ*, n*
Ensure: *Decision - Valid or Invalid*
1: Bit strings $b_{t-1}, ..., b_0$ are calculated as in steps 1 to 5 of algorithm 2
2: $(V_{n-1}, ..., V_0) = f^{2^w-1-b_{t-1}}(\sigma_{n1}), ..., f^{2^w-1-b_0}(\sigma_0)$
3: **if** $V == Y$ **then**
4: Signature is valid
5: **else**
6: Signature is invalid
7: **end if**

1.1.1.3 Type of Nodes

Nodes of IOTA network can be any of the following types based on its computational and storage infrastructure.

- Full Node: It is a standard node on the peer to peer network running the IOTA Reference Implementation (IRI) that is the IOTA open source software. This node is required to connect to neighbors via a static Internet Protocol (IP) address or any other static address and propagate transactions, otherwise its neighbors remove it from their list. The static address is required to propagate and received transactions to and from the neighbors. It stores the sub tangle or snapshot plus all subsequent transactions of the tangle. It can issue transactions and do the necessary validation by itself such as balance verification, Proof of Work verification of other transactions, Signature validation, etc.

- Light Node: It is also called light-client, or light-wallet on the network but relies on a full-node to act as a server. Light node does not store the tangle because of its less infrastructure and it must request the full node to provide transactions but it can validate the Proof of Work (PoW).

- Perma Node: It is a full-node with the facility to store the entire tangle. In IOTA, after every time interval, the sub tangle will be deleted after taking the snapshot (account balance, address, etc) of the tangle. Full node only store the snapshot and future transactions of the tangle but the permanent node that is perma node stores entire history of the Tangle that is from the genesis transaction to upto date transaction.

1.1.1.4 Snapshot

In blockchain, every full node is required to keep all growing blocks but IOTA made the provision to store only the subtangle along with the snapshot. The IOTA coordinator performs a signed snapshot once in a while. A new snapshot deletes all transactions or sites from the Tangle and only stores the balance of each address. A full node has to store the snapshot and all subsequent transactions to validate the transactions. [17]

1.1.1.5 Peer Discovery

The IOTA network is a mesh structured network, which requires every full node to have multiple neighbors to exchange the ledger updates[1]. The IOTA Foundation tested automatic peer discovery in 2016, but realized that it slows down the network noticeably since it consumes more bandwidth on sharing. Hence, in the current IOTA implementation, the users on their own need to find the recommended number of seven neighbors via community platforms, like Slack. This reduce the chance of Sybil-attack, because the attacker would need to convince hundreds of human users.

1.1.2 Address Construction

An user can join the IOTA network without any specific address but to make transaction, address is mandatory. An user can generate the transaction address using the random seed and index. The seed can be generated by computing a hash on the user secret password or using any random algorithm. The index is a number starting from 0 to 9007199254740991 for each address. The maximum size of the seed is 81 characters that is 81 trytes having possible values from A to Z and integer 9. Every address have a sequence number, which is the index of the address. In the construction, key pairs (public and private key) will be generated and public key after after adding checksum is the address and length of the private key is depend upon the security level such as 1, 2 and 3. According to the requirement, security level will be applied as follows.

- Level 1 - It has relatively low security but very high efficiency. It is applicable for tiny IoT devices that only transact or store small amounts of value.
- Level 2 - It has standard security and medium performance. It is applicable for people's wallets.
- Level 3 - It has quantum proof security for sensitive data.

Table 1.2 shows the key size in the form of trits and trytes for different security levels.

[1]IOTA mesh net: Each full node only sees one tiny part of the Tangle through their handful of neighbors. No one has a list of all IPs of all nodes.

Table 1.2: Key Size

Security Level	Key Size (in trits)	Key Size (in trytes)
1	6561	2187
2	13122	4374
3	19683	6561

The process of computing private key, public key and address is shown in algorithm 4. The private key is generated after repetitive hash computation of the seed. In simple, let us assume the private key has 27 fragments in security level 1 and the first fragment of the private key is hash of the sub-seed, second fragment is hash of the first fragment, third fragment is hash of the second fragment and so on upto the computation of 27 parts. The public key is constructed by computing hash of each private key fragment 26 times and computing the hash of all fragments output hash. The public key concatenated with the checksum (last 9 characters of hash of the public key) forms the address. The size of address is 90 trytes including the 9 trytes checksum. The reason for repeating 26 times instead of 27 is explained in signature generation and validation.

The size of seed is 81 trytes and there is a possibility that two node can generate the same seed. However, the chance of getting the same seed is very low. For example, if *Alice* and *Bob* both have exactly last 69 or more trytes equal then one can see the balance of one or more addresses of others. The chance that *Alice* and *Bob* will have the same last 69 trytes of seed is very small because huge combinations of trytes possible as indicated in equation 1.2.

$$Possible\_combinations = 27^{69} = 5.80 \times 10^{98} \tag{1.2}$$

Figure **??** shows the process of computing the address or public key based on Algorithm 4.

Address Re-use: IOTA strongly suggests to use different address for every transaction that is address once used cannot be repeated. Following are the reasons not to re-use the address.

Algorithm 4 Address Computation

Require: *index, seed and level*
Ensure: *address, public key, private key*
1: $subseed_1 = seed + index$ #{size of the seed is 243 trits}
2: $subseed_2 = hash(subseed_1)$ #{size of the $subseed_2$ is 243 trits}
3: $Size = level * 27$
4: $k = 0$
5: **for** $i = 0$ *to* $Size - 1$ **do**
6: $buffer = hash(subseed_2)$
7: **for** $j = k$ *to* $k + 242$ **do**
8: $key[j] = buffer[j]$
9: **end for**
10: $subseed_2 = buffer$
11: $k = k + 242 + 1$
12: **end for**
13: $private\_key = key$
14: $k = 0$
15: **for** $i = 0$ *to* $Size - 1$ **do**
16: $buffer = hash(key[i * 243 : 242 + i * 243])$ #{0:20 is bits from 0 to 20}
17: **for** $i = 1$ *to* 26 **do**
18: $buffer = hash(buffer)$
19: **end for**
20: **for** $j = k$ *to* $k + 242$ **do**
21: $key\_fragment[j] = buffer[j]$
22: **end for**
23: $k = k + 242 + 1$
24: **end for**
25: **for** $i = 1$ *to* $Level$ **do**
26: $digest[i] = hash(key\_fragment[(i - 1) * (27 * 243) : i * (27 * 243) - i])$
27: **end for**
28: $public\_key = hash(digest)$
29: $checkcum = hash(public\_key)$
30: $address = public\_key||checksum[73 : 81]$ #{public key concatenated (||) with last 9 trytes of checksum to derive the address}

- Anonymity - Like Bitcoin, user anonymity can be achieved by using different address for every transaction. If same address is used for all transactions, then the network nodes can track the fund transfer of a user.

- Compromise of account - Attacker may compromise the private key of the account and spend the balances. If we use different address for each transaction then attacker cannot spend the coin even the private key is compromised.

- Key disclose in digital Signature - IOTA uses the Lamport based Winternitz digital signature scheme. In Lamport digital signature scheme, for every signing process part of the key will be disclosed. In case we use the same key to sign two transactions then the key may be compromised. If the private key size is 512 bits then the chance of getting key in first signature is $1/2^{256}$, and in second signature with same key, it will be $1/2^{128}$ and so on.

1.1.2.1 Attach address

Once a user prepare and propagate the transaction, automatically the respective address is attached with the tangle. There are possibilities, user will make the zero value transaction to attach the newly generated address to the tangle. However, following questions arises against attaching the address using zero transaction to the tangle.

- What is the need to attach the address separately when address get attached automatically when we make non-zero value transaction?

- What is the need of attaching address through non-zero value transaction when the addresses attached to the tangle having zero balance will be removed while taking snapshot?

Following are the reasons to attach user address to the tangle even the above questions arise.

- Through attaching, address can be passed to somebody to pay for the address account. For example, *Alice* can create the address and attach to tangle. Later *Bob* use that address and pay to *Alice*.

- To compute the account balance of a user. For example, If a wallet (user or node) looks for all associated addresses and balance in the tangle then it will start by looking for the address at index 0. If it finds a match, it will continue to look for an address at index 1 and then for index 2, 3, etc. till it find a matching address to the index number. Let us assume address 0, 1, 2 and 3 are attached, address 4 is not and address 5 is attached again with some balance on it. The wallet would stop searching for associated addresses once it does not find address 4 and as such would not see the balance of address 5. [1]

1.1.3 Transaction

Transactions are to transfer the IOTA coin (mIOTA) from one account to another account also supports transfer of message between nodes with or without coin value. IOTA supports the following transaction. [24].

- Input Transaction - This is for the deduction of coin. The coin value of this transaction will be deducted from the address provided. The coin value in the transaction will be negative if it is spent for other user otherwise it is greater than zero for transfer of coin among the owner addresses.

- Output Transaction - This is for the addition of coin. The coin value of this transaction will be added to the address provided.

- Meta Transaction - This is a zero value transaction to hold a message or signature fragment. For example, if an input transaction is not sufficient to hold the complete transaction signature then the meta-transaction will be created to store the remaining fragment of the transaction signature. Like in security level 2, signature will be 4374 teraytes, but a transaction can store only 2187 trytes signature. The complete signature will be accommodated by fragment into two transactions. The input transaction will store first 2187 trytes of signature fragment and meta-transaction will store the next 2187 trytes of signature fragement.

- Message Transaction - It is used for messaging along with or without coin value.

- Message Transaction without value: In place of signature fragment, the required message in trytes will be added in the transaction. In case the message size is more than 2187 trytes then another transaction will be added to include the remaining message. The value in the transaction will be zero.
- Message Transaction with value: The output transaction need not to have signature so in signature fragment, the message will be added and the coin value greater than zero will also be there for transfer. In case the message size goes greater than the size of signature fragment then another transaction will be created with the coin value zero.

The transaction structure [6] is shown in table 1.3. In addition to the attributes give in table 1.3, persistence is another attribute added in the implementation to indicate whether the transaction is confirmed or pending. If it is true, the transaction is confirmed otherwise it is pending.The transaction hash will be computed after filling all the attributes that is from serial number 2 to 15 of the table 1.3. For example, if we have 5 transactions, the fifth transaction hash will be computed first then that will be used for the fourth transaction trunk hash and so on.

1.1.3.1 Bundle

Bundle is a unit of transactions has the atomicity property, where all transactions in the bundle will get executre or none of the transactions will get executed. Figure 1.3 shows the structure of the IoTA bundle. Tips shown in the figure 1.3 are bundles not yet confirmed by any other bundles of transactions.

The *trunkTransaction* of the first index (currentIndex 0) transaction points to the second transaction hash, second transaction *trunkTransaction* is pointing to the third transaction hash and so on. But the last transactions *trunkTransaction* points to one of the tip transactions, which is the branch transaction for all other transactions.

1.1.3.2 Bundle Hash

It is the hash of the transaction bundle and it is generated using the transaction attributes in the bundle. The transaction object attributes used for

Table 1.3: Transaction Structure

Sl.No.	Attribute	Description	Size (in trytes)
1	Transaction hash	Hash output of the transaction	81
2	Signature Message Fragment	Signature of the transaction or message. The value 9 appears continously represents the empty	2187
3	Address	receipt address (if value > 0) or withdrawal address (if value < 0)	81
4	Value	Amount of IoTA to be transferred	27
5	Obsolete Tag	Arbitrary user-defined value to generate proper bundle hash	27
6	Timestamp	Linux timestamp at the time of transaction issued	9
7	Current Index	Transaction index in the bundle	9
8	Last Index	Last transaction index in the bundle. If we have four transaction in a bundle then the first transaction, current index is 0 and the last index is 3	9
9	Current Index	Transaction index in the bundle	9
10	Bundle Hash	It is the hash of the bundle. All transactions in the bundle will have same hash to identify the transactions of a bundle.	81
11	Trunk Hash	Every transaction has to approve two existing tip (or) transaction bundle. Last transaction of the bundle refers to the tip and remaining transaction of the bundle will refer to the next indexed transaction of the bundle	81
12	Tag	Arbitrary user-defined value to search the transactions having the similar tag	27
13	Attachment Time	Linux timestamp of PoW completion	9
14 & 15	Attachment Time (lower bound & upper bound)	Linux timestamp but not yet used (none)	9
16	Nonce	This is for the Proof of Work (PoW)	27
17	Weight Magnitude (WM)	Proof of Work (PoW) target that is atleast WM trail zeros must be there in the PoW hash. It is minimum WM and if we get more zeros then the WM then it is also accepted	27

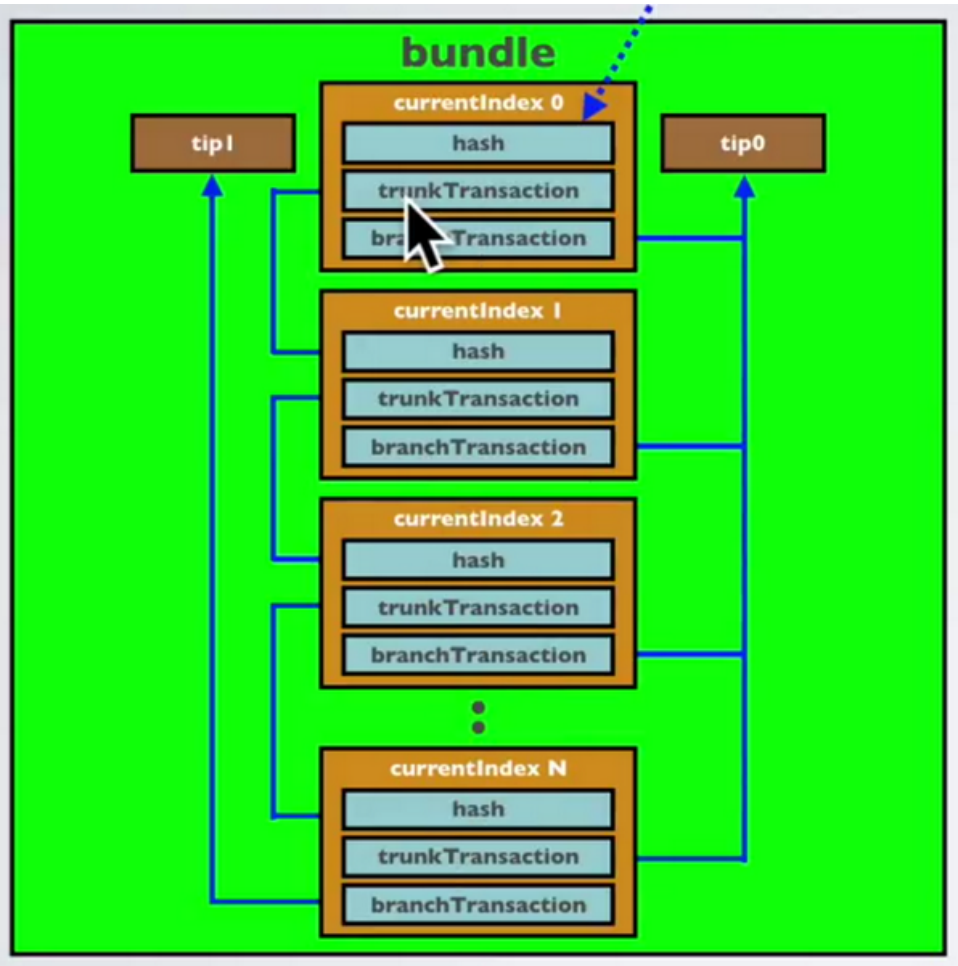

Figure 1.3: IoTA Bundle Structure

computing the bundle hash are as follows

- Address
- Value
- Obsolete tag
- Timestamp
- Current index
- Last index

Using the Keccak 384 hash function, the bundle hash will be computed as shown in equation 1.3

$$Bundle\_hash = h(transaction\ object\ 0 \parallel transaction\ object\ 1 \parallel ....) \quad (1.3)$$

The computed bundle hash will be added into all transactions to confirm that the transaction belongs to the respective bundle.

1.1.3.3 Normalized Bundle Hash

The computed bundle hash may be biased that is sum of decimal value of hash character may either be on positive side (that is higher side) or negative side (that is lower side). The bundle hash is used to create and validate the bundle signature. To compute the normalized bundle hash, at first divide the 81 trytes bundle hash equally into 3 fragments that is 27 trytes each. Later convert each trytes of fragment to decimal and compute the sum. Any one of the following will be applied to normalize the hash based on the computed sum.

- In case, sum is less than ($<$) 0 then it is biased on the lower side. To balance it, take the first tryte of the fragment and increase the value by one, compute the sum and check whether it is greater than or equal 0. If not repeat the process till first tryte equivalent decimal become 13. If the tryte decimal equivalent is +13 and sum not greater than or equal to zero then move to the second tryte, third tryte and so on till sum becomes greater than or equal to zero.

- In case, the sum is greater than or equal to zero then take the first tryte and reduce the value by one upto -13 or till the fragment sum less than 0. In case, the first tryte is -13 then take next tryte and proceed.

The process of computing the normalized bundle hash is shown in algorithm 5. In case the fragments normalized bundle hash has the trytes M then new bundle hash needs to be generated by modifying the obsolete tag in any one of the transactions. In signature validation, we can discuss in detail the reason for not allowing M in the fragment hash.

1.1.3.4 Signature Computation

The transaction signature is computed using the normalized bundle hash and the private key of the address. The process of computing signature is shown in algorithm 7. The normalized bundle hash trytes will be taken one by one, converted to decimal and it will be subtracted with the maximum tryte value 13. The equivalent indexed private key trytes will be hashed those many number of times. The tryte M, will produce 0, if we subtract the decimal equivalent M (13) from 13. In this case, private key tryte will not be hashed and key will appear in the signature and leak the fragment of private key. Because of this reason, M is not allowed in the normalized bundle hash.

1.1.3.5 Signature Validation

The signature is verified using the normalized bundle hash derived from the publicly available bundle hash in the transaction bundle. The signature validation process is shown in the algorithm 8. In validation, each normalized bundle hash trytes decimal equivalent is added with highest tryte value 13 and equivalent number of times signature fragment is hashed. The hashed signature fragment will produce the address, which spends the coin. If the address produced by the validation algorithm and the address available in the transaction is same then the transaction signature is valid otherwise it is invalid. If it is valid, the owner is spending the coin otherwise someone else attempt to spend the coin.

Algorithm 5 Normalized bundle hash computation

Require: *bundle_hash* (b_h)
Ensure: *normalized bundle_hash*

1: Divide b_h into three equal parts $b_h(1), b_h(2), b_h(3)$ #{each part is of 27 trytes}
2: **for** $i = 1$ *to* 3 **do**
3: $sum = 0$
4: **for** $j = 0$ *to* 26 **do**
5: $dec[j] = convdec(b_h(i)(j))$ #{convdec() is trytes to decimal conversion function}
6: $sum = sum + dec$
7: **end for**
8: $index = 0$
9: **while** $sum < 0$ **do**
10: **while** $dec[index] > 13$ **do**
11: $index++$
12: $dec[index]++$
13: **end while**
14: $sum(dec)$
15: **end while**
16: goto 24
17: **while** $sum >= 0$ **do**
18: **while** $dec[index] < -13$ **do**
19: $index++$
20: $dec[index]--$
21: **end while**
22: $sum(dec)$ #{call the function sum() in algorithm 6}
23: **end while**
24: **for** $j = 0$ *to* 26 **do**
25: $bh[j] = convdec(dec(j))$ #{convtry() is decimal to trytes conversion function}
26: **if** $bh[] ==' M'$ **then**
27: Re-compute bundle hash #{Re-compute the bundle hash by chainging the obsolete tag in transactions}
28: break; #{Break the process and recompute the normalized bundle hash after computing the new bundle hash}
29: **end if**
30: **end for**
31: **end for**

Algorithm 6 Sum the trytes ($sum(dec)$)

Require: *decimal array (dec)*
Ensure: *sum*
1: $sum = 0$
2: **for** $j = 0$ *to* 26 **do**
3: $sum = sum + dec[j]$
4: **end for**

Algorithm 7 Signature Computation

Require: *Address private_key (pr_k), normalized bundle hash (nb_h)*
Ensure: *Signature (sig)*
1: **for** $j = 0$ *to* 80 **do**
2: $t[j] = convdec(nb_h[j])$
3: $t[j] = 13 - t[j]$
4: $sig[j] = pr_k[j]$
5: **for** $i = 0$ *to* $t[j]$ **do**
6: $sig[j] = h(sig[j])$
7: **end for**
8: **end for**
9: return(sig)

Algorithm 8 Signature Verification

Require: *normalized bundle hash (nb_h), signature (sig), transaction address (t_{addr})*
Ensure: *Signature (sig)*

for $j = 0$ *to* 80 **do**
  $t[j] = convdec(nb_h[j])$
  $t[j] = 13 + t[j]$
  $fragment[j] = sig[j]$
  for $i = 0$ *to* $t[j]$ **do**
    $fragment[j] = h(fragment[j])$
  end for
end for
$digest = hash(fragment)$
$address = hash(digest)$
if $t_{addr} == address$ **then**
  Valid signature
else
  Invalid signature
end if

1.1.3.6 Transaction Validation

The IOTA nodes validates the transactions in the following two stages

- On receipt of new transactions or bundle: When a node receives a new transaction bundle, it verifies the following

  The Proof of Work (PoW) is done or not. It can be verified using the *nonce* and other attributes of the transactions. Node validates whether the hash of *nonce* along with other attributes of transaction is according to the target that is Weight Magnitude (WM).

  - The value of any transaction in the bundle does not exceed the total global supply.
  - The transaction is not older than the last snapshot and not newer than two hours ahead of the node's current time.
  - The last trit of an address is 0 for value transactions. This is to differentiate the message and value transactions.
  - Signatures are valid in value transactions.

- During the tip selection process: This is the approval of the transaction bundle. Each bundles on tip selection path are checked by the bundle and ledger validator.

  - Bundle Validator: The bundle validator checks the transactions for the following [3]:
    * The value of any transaction in the bundle does not exceed the total global supply.
    * The total value of all transactions in the bundle is 0 (inputs and outputs are balanced).
    * Signatures are valid in value transactions.
  - Ledger validator: The ledger validator makes sure that double-spends are never confirmed
    * A full node has a local database of current IOTA balances (which gets initialized from the latest Snapshot and then updated by confirmed transactions). When the full node performs its validation of a transaction bundle, it will verify that

the balances of the spending transactions do not exceed the amounts stored in its database. This is done sequentially so if there are double spends, only one can be validated. [5]

1.1.3.7 Transaction Confirmation

The confirmation can be decided by running the tip selection algorithm $\mathcal{N}$ times. The probability of your transaction confirmation or being accepted is therefore $\mathcal{M}$ of $\mathcal{N}$ where $\mathcal{M}$ being number of times you land on a tip that has a path to your transaction. For example, if we run the tip selection algorithm 100 times and 60 tips have path to your transaction that means it is 60% confirmed. It is up to the merchant to accept the transaction or not. Similar to bitcoin, which advice to wait for 6 blocks, the merchant of IOTA can wait for certain percentage(%) of confirmation. However, there are chances transaction not get confirmation in such case the transaction should be broadcasted, reattached or promoted.

Rebroadcast: Every node broadcast its transaction bundle to its neighbors. In case, transactions does not reach the network then there is a need to rebroadcast the transaction bundle.

Reattach: Transaction which is part of the tangle but not yet approved in 30 minutes then the chance of respective transaction is very less because tip selection always prefers new transactions instead of older ones. Reattaching the transaction will increase the chance of confirmation. Reattaching the transaction means create a new transaction with the same signed bundle as the original transaction but with new trunk and branch transaction. Two random tips (trunk and branch) are picked again and added in to the new transaction, also compute the Proof of Work for the new transaction and replace the nonce. The following attributes of the new transaction will be modified remaining will as it was in the old transaction.

- Transaction Hash
- Trunk hash
- Branch hash
- Attach timestamp

- Nonce

Reattach transaction can pave the path to double spend however only one transaction will be confirmed. The other transaction(s) will be pending.

Promote: In case, any transaction is pending for more than 30 minutes or so then make it to confirm, we can create a zero transaction to promote the pending transaction. In zero transaction, make a transaction with 0 value and choose the two tip one as the mile stone and another one as the pending transaction. This promotes the transaction that means the new zero transaction becomes the tip and chance of acceptance is more since it has the tip of milestone transaction. The promote transaction gets more chance to confirm the pending transaction compare to the reattach transaction. The promote transaction will iterated until the transaction is confirmed.

1.1.4 Tip Selection

A user wish to make a transaction has to approve at least k tips in the tangle. Usually tangle contains multiple tips and user has to choose any k tips randomly by traversing from a random site in the tangle. User has to select the tips randomly by running the weighted random walk algorithm

When the walkers have reached a tip, the node can decide among them. To make selection faster, the node could simply select the walker which found a tip rst. But then old tips might be favored, which are naturally closer to where the walkers started their walks. Consequently, it might be best to either wait, for example, for the third walker or to ignore any walker who nished "too fast"

Nodes are not required to follow one Tip Selection Strategy, but they benet from aligning their strategy to some non-deterministic "reference" rule for the following reasons: Nodes want to maximize the velocity by which, in turn, their own issued txs are approved. If the probability distribution of another strategy deviates very much from the default one, then these txs are in general less likely to be selected by subsequent txs. In other words, this keeps the probability of selecting a "bad" tip small. If all nodes were able to follow a deterministic strategy, all of them would end up choosing the same tip and so there would be much competition for subsequent approvals. A superior strategy might include nding out the "best" tips, but this is hardly possible, because plenty of walkers would have to be calculated which is time

consuming. Then, once a result is found, the Tangle has already changed.

1.1.4.1 Weighted Random Walk

The walk always starts with the latest milestone transaction and proceed the path until get the tip. In the implementation of IOTA cryptocurrency, every transaction need to approve two tips hence at least two entry points to be decided to start the algorithm. Nodes always choose latest milestone transaction of the tangle as one entry point and for the second entry point, it may choose any transaction in the path of milestone to tip transaction or the previous milestone transaction. For example, if there are 10 milestone transactions on the tangle, node will always choose the latest milestone (i.e, 10) as one entry point and another entry point as 9^{th} milestone or the transaction available in the path of 10^{th} milestone transaction and tip.

In general, there will be more than 10 random walk threads created to choose the tip because there is a possibility with only two threads we may not get the tip. IOTA uses Monte Carlo Markov Chain (MCMC) based random walk algorithm to pick the tip. The algorithm puts random walker on sites of the tangle to walk randomly towards the tip. The MCMC random walk process is as follows:

- Consider all sites on the interval $[W, 2W]$, where W is chosen reasonably large.
- Independently place N walkers on any sites in that interval
- Walkers perform independent discrete-time random walks towards tip by traversing multiple sites on the tangle. Transition from site x to y is possible if and only if y approves x.
- The two tips identified at first will be approved. However, it may be wise to modify this rule in the following way: first discard those random walkers that reached the tips too fast because they may have ended on one of the 'lazy tips', which are not approved for long time.
- The transition probabilities of the walkers are like if y approves $x (y \rightsquigarrow x)$, then the transition probability P_{xy} computed in equation 1.4 is proportional to $e^{-\alpha(H_x - H_y)}$.

$$P_{xy} = \frac{e^{-\alpha(H_x - H_y)}}{\sum_{z:z \rightsquigarrow x} e^{-\alpha(H_x - H_z)}} \tag{1.4}$$

Where P_{xy} is the probability to walk from x to y, H_y is the cumulative weight of transaction y, $z \rightsquigarrow x$ means z *directly approves* x and $\alpha > 0$ is a parameter to emphasize the significance of weight in computation.

1.1.4.2 Why Random?

The tip selection method provide the following two features

- Once a transaction accumulates large number of approvers, it is very unlikely to get abandoned from the main tangle.
- Honest transactions should get approved quickly.

To achieve the first goal, we can perform a deterministic walk from the genesis to the tips, always going towards the approver with the largest cumulative weight. However, this would discourage the second goal because only a single central chain of sites would get approvals and most transactions would be left behind. Hence, the transition is made through the random walk.

Example A transition function, computes the probability to move from approvee to approver during the random walk. We would like this probability to be large for approvers with a high cumulative weight, and small for lighter approvers with a less cumulative weight [21]. Equation 1.5 is the transition function used :

$$P_{xy} = \frac{e^{\alpha H_y}}{\sum_{z:z \rightsquigarrow x} e^{\alpha H_z}} \tag{1.5}$$

The probability of walking from x to y increases exponentially with the cumulative weight of y multiplied with α. The sum in denominator is a normalization factor, which sets the total possible transition probabilities sum to one.

In Figure 1.4, lets assume the random walker reached transaction X, which has three approvers or transactions $(A, B, andC)$. The cumulative weights of A is 2 because of one approver and its own weight, B has two approvers, so it has weight 3 and C has no approvers, so its weight is 1. The

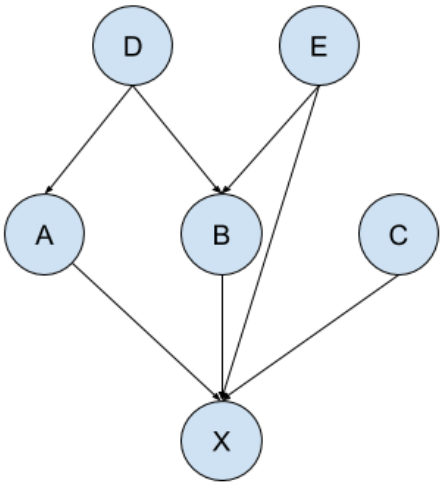

Figure 1.4: A sample graph

probability of choosing the next transaction is computed in the equation 1.6 to 1.8 with value 1 to α.

$$P_{XA} = \frac{e^{\alpha H_A}}{e^{\alpha H_A} + e^{\alpha H_B} + e^{\alpha H_C}} = \frac{e^2}{e^2 + e^3 + e} \approx 0.24 \tag{1.6}$$

$$P_{XB} = \frac{e^{\alpha H_A}}{e^{\alpha H_A} + e^{\alpha H_B} + e^{\alpha H_C}} = \frac{e^3}{e^2 + e^3 + e} \approx 0.67 \tag{1.7}$$

$$P_{XC} = \frac{e^{\alpha H_A}}{e^{\alpha H_A} + e^{\alpha H_B} + e^{\alpha H_C}} = \frac{e}{e^2 + e^3 + e} \approx 0.09 \tag{1.8}$$

The transaction C has low probability to get chosen than A or B because of less weight. If we set smaller value to α, we decrease the bias against C. For example, if we take $\alpha = 0.1$, we get the following probabilities:

$$P_{XA} \approx 0.33 \qquad P_{XB} \approx 0.37 \qquad P_{XC} \approx 0.30$$

There is still a slight bias against transaction C but it is drastically increased. If we examine the two extreme cases of α.

1. With $\alpha = 0$, all approvers are completely equivalent, and have a probability of 1/3. In this case, the weights stop contributing, and the walk is completely random.

2. With large α, the probability to walk towards B is 1, and A and C have transition probability zero. This case is similar to blockchain, where we approve only the single most likely block, and never merge different branches together.

Hence, the value for α should be chosen without any bias.

1.1.5 Masked Authenticate Messaging (MAM)

The Masked Authenticated Messaging (MAM) is a data communication protocol which adds functionality to emit and access data stream over the Tangle regardless of the size or cost of device. In IOTA, any user can publish a message at any time but only need to compute PoW to allow the data to propagate through the network. The PoW computation is required to control the spam attack. The message transaction need not to be confirmed and it will also be deleted after snapshot. IOTA messages are masked and authenticated that is messages are encrypted if required and receiver can confirm through the signature that the message is from the respective device. IOTA messages can be sent in the following three modes.

- Public Mode - Any one in the network can read the message
- Private Mode - A node in the network knows the merkle root of the message can read it.
- Restricted Mode - Node with the side key and merkle root can read the message.

User wish to send message to the receiver, will mask the message and attach in the tangle. In order to mask the message, following components are required.

- Side Key - The message is encrypted using the side key and the same is used for decryption. It is needed to be shared in the restricted mode.

- Mode - It can be public (anyone can be allowed to access, where address is the root), private (only the publisher has the access to the message, where hash of address is the root) and restricted mode (any one having the side key can access the message, where hash of address is the root)
- Security Level - The level as discussed in the key generation.
- Start - It is an an index of the merkle tree first leaf. For example, in figure 1.6 merkle tree ABC has the start index 0, merkle tree RST has the start index 1 and merkle tree CDE has start index 3. Start index of a next tree is the start index of the current tree plus the count of the leaves.
- Count - It is the number of leaves in the merkle tree.
- nextCount -It next merkle tree leaf count. When the message is created two merkle trees are created: one is the current merkle tree and another is the next message merkle tree. The nextcount is number of leaves in the next merkle tree.
- Index (i) - It is an index for the leaves of a merkle tree. For example, in figure 1.6, merkle tree CDE address 3 has the index 1, address 4 has the index 2 and so on.
- nextRoot - This is the root of the next merkle tree to access the next message in the message chain.
- Seed - It uses the same seed used for key generation
- Payload - It is the actual message.
- Root - It is the root of the merkle tree. For example, in figure 1.6, first merkle tree root is ABC. Merkle Root is computed using the hash value of the child nodes as discussed in the Introduction chapter.
- Address - Address of the message to attach in a tangle.

Figure 1.6 shows the masked message payload structure with the following parameters.

- Index - This is equal to index i discussed in the components

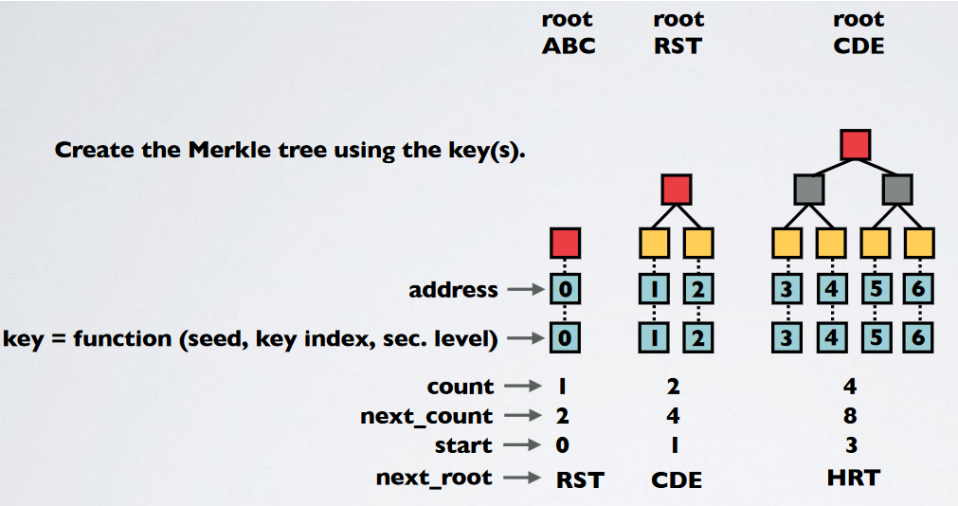

Figure 1.5: IoTA Merkle Tree [2]

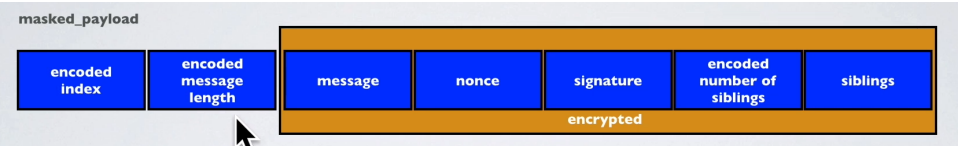

Figure 1.6: IoTA message payload [2]

- Length - This is length of the message to be transferred.
- Message - Actual message and next root discussed in the components
- Nonce - This is similar to checksum computed through the hash function with input parameters side_key, root, message index, message length, message and next root.
- Signature - It is the signature of the message
- Number of siblings - Total count of siblings in the merkle tree. For example, in figure 1.7, the siblings are given in blue color. Number of siblings for the node marked **X** is 4.
- Siblings - All the siblings of the leaf node to verify and compute the root. For example, in figure 1.7, the siblings are given in blue color and its value will be provided in this parameter .

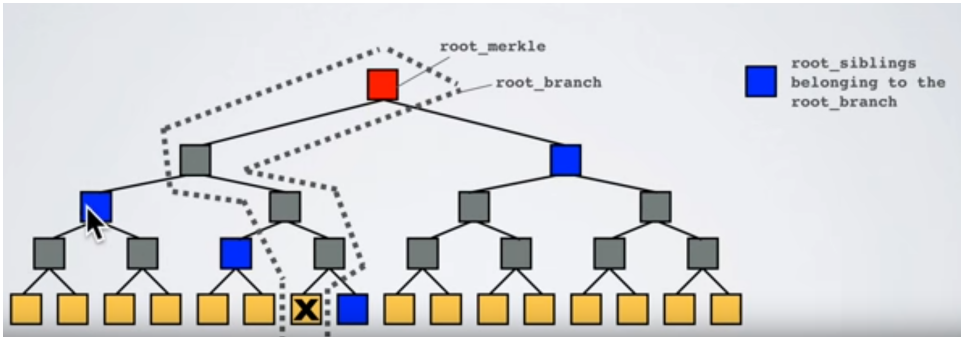

Figure 1.7: IoTA Merkle Tree Path

The IOTA message bundles are created as per the following steps using the components discussed before.

- *Step 1*: Construct the side key (S_k) if it is restricted mode of transfer
- *Step 2*: Construct the messages (M) and index it.
- *Step 3*: Compute the key (k_i) for each message using the seed, key index and security level like private key computation as discussed in the address construction.
- *Step 4*: Generate the address A_i for each message using the hash function - $h(k_i)$
- *Step 5*: Generate the merkle tree using the available message addresses. Merkle root is R.
- *Step 6*: Repeat steps 2 to 5 to generate the next merkle tree. Next merkle root is nR.
- *Step 7*: Prepare the message $m_i =$ by concatenating the i^{th} message from M and next merkle tree root.
- *Step 8*: Compute Nonce N_i using the hash function with the input $S_k, R, i + length(m_i)$ and m_i. First 81 trits of the hash output is the nonce.
- *Step 9*: Compute bundle hash Bh_i using the hash function with the input $S_k, R, i + length(M_i), m_i$ and N.

- *Step 10*: Compute the signature sig_i for message m_i using the key k_i and the computed bundle hash Bh_i.
- *Step 11*: Compute the number of siblings ns_i for each address A_i.
- *Step 12*: Store all siblings of a message m_i in a list $Sib_i[]$.
- *Step 13*: Encrypt the message bundle MB_i using side key S_k and message root $R => eB_i = e(MB_i, S_k, S_k)$, where e is any encryption function.
- *Step 14*: Compute the message mask mm_i, where $mm_i = (i \parallel length(m_i) \parallel eB_i)$

The computed message mask mm_i will be placed in the transaction as a signature parameter. In case the message mask length is greater than the transaction signature parameter length then it will be divided and placed in multiple transactions. Other attribute computations of the transaction is similar to the non-zero value transaction discussed before.

1.1.6 Possible Attacks

IOTA is a permissionless network, which allows any one with defined requirement to take part in the network. This would pave the path for multiple attacks and few are discussed here.

1.1.6.1 Double Spending

Double spending is a situation where an dishonest node spends the coin more than once. The dishonest node or attacker adds two conflicting transactions w and y in different areas of the tangle as shown in figure 1.8. The conflicting transaction means, the same coin is spent in both w and y transactions. Future transactions of the tangle might only have one of these conflicting transactions in their validation or tip selection path. For example, the transactions 1 and 2 will not see the conflict and confirm their chosen tips since it has not seen both w and y. Hence, the double spend attempt got its first confirmations from transactions 1 and 2. This may proceed to certain length and get the desired confirmation for succulence double spending. However, before the desired confirmation to double spend transactions, there is a high possibility that both conflicting transactions are in the path of validation

of one transaction. For example, transaction 5 in figure 1.8 would see the conflict and not attach to the elected tips. Instead it would reselect tips until it found to not conflicting ones in order to be sure itself turns into a valid transaction. In case transaction 5 approves transaction w and y then future transactions will not approve it.

It might happen that many transactions after w or y confirms those before the conflict becomes clear. However, at some point, either w or y will get confirm, while the other gets abandoned. All subsequent transactions approved or attached to the abandoned one will also be abandoned. In such case, even the honest users transactions will be abandoned. To overcome such issue, those transactions can be reattached to the tangle for a new chance of confirmation.

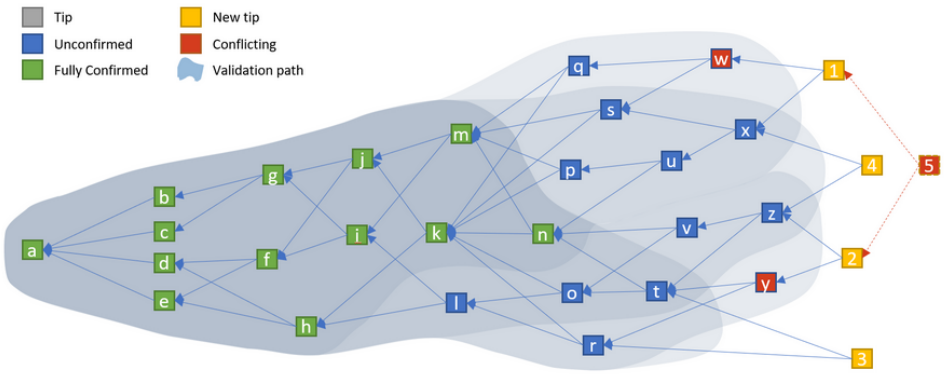

Figure 1.8: Sample Double Spend [10]

In figure 1.8, the transaction 5 finds the conflict and decided to attach to tips 1 and 4 as shown in figure 1.9. The transaction 7 attaches to tips 2 and 3. In this case, branches are arisen but only one can survive, due to the double spend in w and y. Based on the random selection of tips, one of the two branches will receive more child transactions for example users transaction could just continue attaching to transactions 5, 6 and 8 but not to transaction 7 anymore. Hence, transactions y, 2, 3 and 7 will never make it into a fully confirmed state. Hence attacker could spend only once that is transaction w and y will be abandoned. Transactions y, 2, 3 and 7 could be reattached to the tangle again for fresh chance of confirmation however transactions 2, 3 and 7 might become confirmed but y will stay invalid.

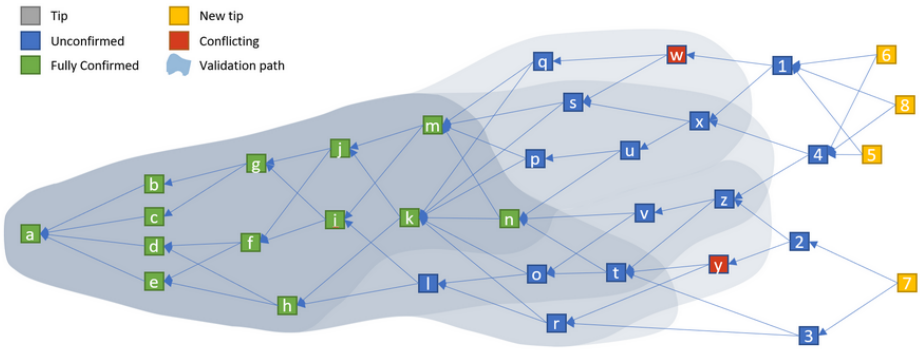

Figure 1.9: Sample Double Spend Resolution [10]

1.1.6.2 Parasite Chain Attack

An attacker secretly builds a subtangle called as parasite subtangle or parasite chain, which is invisible to the public that occasionally references the main tangle to gain a higher score. The score of honest tips is the sum of all own weights in the main tangle and score of the attacker's tips is the sum of all own weights in the parasite chain. An attacker who builds a subtangle alone might be able to give more height to the parasite tips he/she has the sufficient computing power. Also attacker can increase the number of tips at the moment of the attack by broadcasting many new transactions that approve transactions that they issued earlier on the parasite chain. This will give the attacker an advantage in the case where the honest nodes use some selection strategy that involves a simple choice between available tips. This attack makes attacker to double spend the coin.

Lets see an example using the figure 1.10. The attacker at first broadcast the green transaction into the Tangle and simultaneously, in secret, creates a second spend of the same coin with the yellow transaction followed by a chain of transactions, which approves it. The attacker waits for the green transaction to be confirmed, and the goods to be delivered, and then immediately broadcasts the secret or parasite chain to the network and continues publishing transactions which validate it [19].

This attack can be defended with the fact that the main tangle is supposed to have more active hashing power than the attacker. Hence, the main tangle is able to produce larger increases in cumulative weight for more transactions than the attacker. Also, random walkers on the tangle walk towards the tips

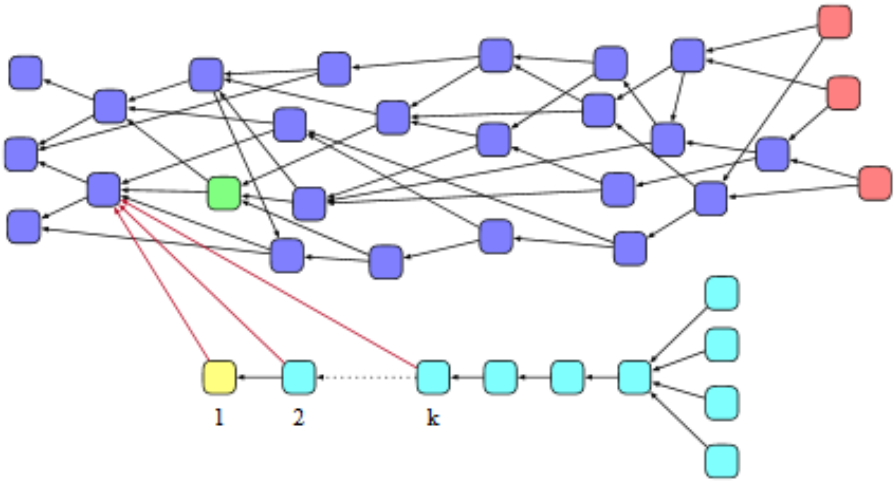

Figure 1.10: Parasite Chain

in a random way.

1.1.6.3 Splitting Attack

An attacker can try to split the tangle into two branches and maintain the balance between them and make both branches grow continuously. Split is possible only if attacker posts at least two conflicting transactions which prevent any honest node from effectively joining the branches by referencing both of them simultaneously. In later stage, it may be possible that roughly half of the network would contribute to each branch. In this case, the attacker would be able to spend the same funds on the two branches.

This attack can be defended if it will be hard to maintain the balance between two branches. Let us assume that the first and second branch has total weight 537 and 528 respectively. If an honest node selects the first branch with probability very close to 0.5, then the attacker would probably be able to maintain the balance but if the first branch is selected with probability much larger than 1/2, then the attacker would probably be unable to maintain the balance because it will be biased on one branch in a period of time.

It may not be possible for the attacker to balance between the two branches in the latter case is due to the fact that after an inevitable random

fluctuation, the network will quickly choose one of the branches and abandon the other. In order to make the Markov Chain Monte Carlo algorithm behave this way, initiate the random walk at a node with large depth using a decaying function so that it is highly probable that the walk starts from before the branch bifurcation in the tangle. In such case, the random walk would choose the heavier branch with high probability, even if the difference in cumulative weight between the competing branches is small.

There are other difficulty in performing this attack as discussed below

- *Network synchronization issues*: Attacker may not aware of a large number of recently issued transactions
- *Powerful node Balance*: An entity having more power can instantly publish a large number of transactions on one branch, thus rapidly changing the power balance and making it difficult for the attacker to deal with this change.
- *Less confirmation*: The most recent transactions will only have around 50% confirmation confidence, and the branches will not grow. In such case, the honest nodes may decide to start selectively giving their approval to the transactions that occurred before the split of tangle.
- *Own weight*: Nodes may choose the subtangle, which has larger sum of own weights. After choosing the subtange, the MCMC tip selection algorithm be executed.

1.2 Corda

Corda is a platform, which implements the distributed ledger and data such as transaction, state, etc. of an user is shared with other users only on a need-to-know basis and no global broadcasts in the network [23]. This platform is permissioned and specially designed for the financial sector. Any user can join the Corda network after registering with the network operator. The network operator is a trusted entity to provide specific service to the network participants. The network operator issue a digital certificate containing the user identity and other credentials to the participating user. Using the identity and available credential, nodes can make transaction with other nodes in the network through the notary. The notary is the set of nodes to store

the sub-network global ledger and validate the transaction. The nodes need to store only the transaction relevant to it not required to store all the transactions sent in the network however nodes may request the notary and keep all the transactions of the network. The notaries in the network will validate all transactions to identify the double spend and maintain the decentralized ledger. Nodes including notary can store the transaction details in the relational database for efficient access. Unlike other blockchain implementation, in Corda transaction will be sent only to the respective node through the notary using the IP address mapped with identity provided by Corda mapping service. The Corda supports smart contract transaction along with asset transaction. Also it ensures partial privacy by hiding the part of the transaction whenever required. The important entities for the Corda implementation are as follows.

User or Node: A party in the network communicates with other parties and maintain its own decentralized storage and relational storage. A relation storage is a conventional RDBMS storage, data from the decentralized ledger will be moved to the RDBMS for efficient access.

Identity: A node wish to take part in Corda should have single well-known identity and it is used to represent the node in transactions, such as transfer of value and creation of smart contract. Using network map service, the node identity maps to an Internet Protocol (IP) address, which is used for messaging between nodes.

Confidential Identity: Nodes can generate confidential identities for individual transactions and that will be used only when required. The digital certificate chain links a confidential identity to a well-known node identity or real-world legal identity. Even an attacker gets access to a plain transaction cannot identify the transaction's sender and receiver without additional information such as certification link.

Joining the Network: It is semi-private network similar to the permissioned public blockchain. Node wish to join a network must obtain a certificate from the network operator by providing the required information that is Know-Your-Customer (KYC) data. This certificate maps a well-known node identity to real-world legal identity, public key and confidential identity.

1.2.0.1 Services

Corda has following different services to support the decentralized storage and validation of transactions.

- *Network Map*: Every Corda network has a network map service, which is composed of multiple cooperating nodes. The network map publishes the IP addresses of every node to the network, along with the identity certificates of those nodes and the services they provide. An IP address is used to connect the nodes and each node will publish one or more IP addresses to the network through the network map service. The domain name equivalent of IP address may be helpful for debugging but are not required.

- *Reliable Message Delivery*: In the public network, there is no assumption of constant connectivity and continuous liveness of a node. It means nodes may depart temporarily due to crashes, connectivity interruptions or maintenance. Corda considered such situation and proposed a solution with retransmission. A messages to be delivered are written to disk and delivery is retried until the remote node has acknowledged a message.

- *Serialization*: All messages are encoded using a compact binary format to serialize the data while transmission and storage.

- *Deduplication*: Corda ensures the deduplication of message delivery using the Universal Unique Identity (UUID). Every message contains an unique identity, which is set in an Advanced Message Queuing Protocol (AMQP) header. This Identity is used as a deduplication key, thus ensures that accidentally redelivered messages are ignored.

- *Session*: Every Corda messages may also have an associated 64-bit session identity (ID). The session can be long lived and persist across node restart and network outage.

- *Receipt*: A node successfully validates the message will send a signed receipt to the sender. The purpose of the receipt is to give a node undeniable evidence that a counter party received a notification that would be used in a dispute mediation process.

- *State*: A state is an immutable object representing a fact known by one or more Corda nodes at a specific point in time. States will be used as the input and output for the transaction. A state can be updated by creating a new state.

- *Permissioning Service*: This service provides digital certificates for nodes for the secure communication.

- *Notary Service*: This is to validate transaction and identify the double spend. This service is provided by the notary nodes and it may be distributed over multiple nodes.

1.2.1 Network Permissioning

In a permissioned network, nodes can take part after the approval of an administrative node or network operator. In Corda, a node needs signed X.509 certificate from the network operator to join the network or zone. The certificate is to prove that the node is authenticated by the network operator. This certificate may be issued by different entities as discussed in the following.

1.2.1.1 Certificate Hierarchy

The Certification Authority (CA) create, sign, and issue public key certificates to subscribers. The Corda network has three types of Certificate Authorities (CAs) to issue the certificate and hierarchy of CA is as follows:

- Root network CA: Root CA's are implicitly trusted and the operator of a network.

- Doorman CA: It is used instead of the root network CA for day-to-day key signing and to reduce the risk of the root network CAs private key being compromised. This is equivalent to an intermediate certificate in the web Public Key Infrastructure (PKI).

- Own CA: Each node also serves as CA for itself to issue the child certificates that it uses to sign its identity keys and TLS certificates.

In the hierarchy, the certificate to validate the Root Network CA is issued by itself. The certificate to validate the Doorman CA will be issued by Root

Network CA and the certificate to validate the Own CA will be issued by the Doorman CA. The certificate type can be identified using the integer identity present in the certificate. The CAs must follow one of the following public key cryptographic algorithm and curve to generate the certificate.

- ECDSA using the NIST P-256 curve (secp256r1)
- ECDSA using the Koblitz k1 curve (secp256k1)
- RSA with 3072-bit key size or higher.

1.2.2 Ledger

The Corda decentralized ledger is different from other decentralized ledgers such as Ethereum and Bitcoin. Each node maintains a separate database containing its known facts or states not the complete network states or facts. As a result, each node can see a subset of facts on the ledger, and no node has the ledger with complete facts of the network [11]. For example, let us assume a network with five nodes (*Alice, Bob, Carl, Demi and Ed*) as in figure 1.11, where each coloured circle represents a shared fact. The fact 1 is available only with *Alice* and *Bob*, the fact 3 is only with *Carl, Demi* and *Ed*, similarly other facts.

1.2.2.1 State

A fact is called as state, which is an immutable object known by one or more Corda nodes at a specific point in time. A states can contain arbitrary data, allowing them to represent facts of any kind (e.g. stocks, bonds, loans, KYC data, identity information, etc.) [12]. For example, the state in figure 1.12 represents an I Owe You (IOU) - an agreement that *Alice* owes *Bob* an amount 10, due date is 01/03/2017 otherwise penalty of 20% and already paid 5 euro.

1.2.2.1.1 State Update A state is an immutable object or fact known by one or more Corda nodes at a specific point in time. In Corda, the life cycle of a shared fact is represented by a state sequence. In case a state needs to be updated, node can create a new version of the state representing the new state of the world, and mark the existing state as historic. For example, in figure 1.13, the state represents an agreement that *Alice* owes *Bob* an

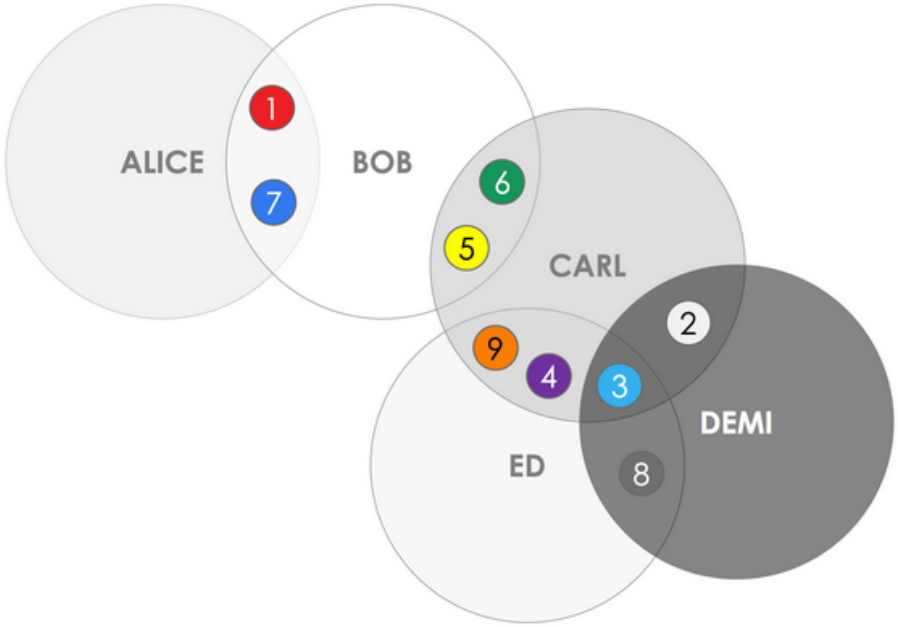

Figure 1.11: Corda Ledger [11]

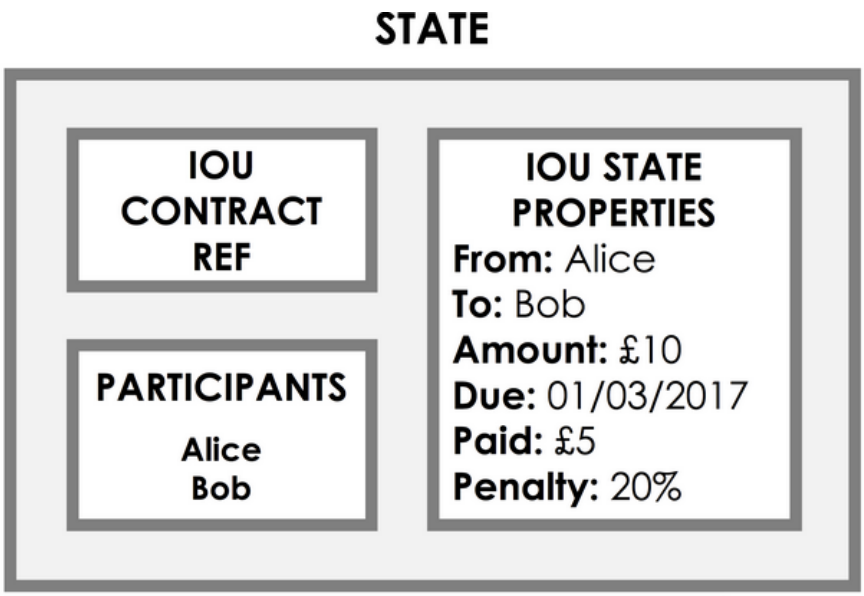

Figure 1.12: Corda State [12]

amount 10, due date is 01/03/2017 otherwise penalty of 20% interest daily is made as historic once the agreement is updated. The update occurs because *Alice* have paid 5 euros to *Bob*. Now, *Bob* cannot claim with the previous agreement.

1.2.2.2 Vault

In blockchain, nodes have a wallet, or as we call it, a vault. The vault contains data such as private key or key derivation material of the owner and stored in a form that can be easily queried [15]. Using vault, a Corda node can track all the current and historic states that it is aware of, and which it considers to be relevant to itself. The vault keeps track of both unconsumed and consumed states:

- *Unconsumed (or unspent) state*: It represents a fungible state available for spending or transfer to another party. *Linear state* - A state that evolves by superseding itself, all of which share the common *linearId*.
- *Consumed (or spent) state*: It represent ledger immutable state for the purpose of transaction reporting, audit and archival.

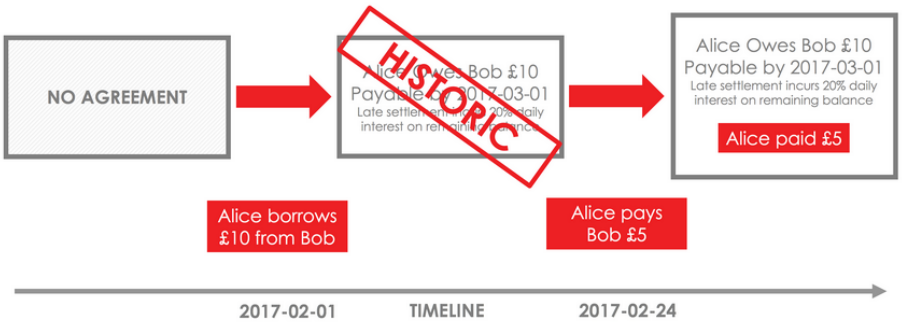

Figure 1.13: Corda State Sequence [12]

Using the Corda vault, node can create transactions that send value to someone else by combining fungible states and possibly adding a change output that makes the values balance called as *coin selection*.

Soft Locking: A feature provides the ability to automatically prevent multiple transactions within the same node trying to use the same output simultaneously. Even though, such transaction will be detected by a notary, soft locking provides a feature of early detection.

Reference: The Corda provides a facility for attaching descriptive textual notes against any transaction stored in the vault.

Ledger types: *On-ledger* - Data refers to distributed ledger state (cash, deals, trades) to which a firm is participant. *Off-ledger* - Data refers to a node's internal reference, static and systems data.

Fungible It means that an asset is interchangeable for any other identical instance, and that they can be split/merged. For example a Rs.5 note can reasonably be exchanged for any other Rs.5 note, and a Rs.10 note can be exchanged for two Rs.5 notes, or vice-versa.

1.2.2.3 Transactions

The Corda transactions consume zero or more input states and create zero or more new output states. Transactions are atomic, where all input and output transactions will be executed or none of the transactions will be executed. The attributes of the Corda transaction are listed in table 1.4.

Figure 1.14 shows a sample Corda cash issue transaction, which contains zero input and one output, a newly issued cash state. The output cash state

Table 1.4: Corda Transaction Attributes

Name	**Identity**
Input	It is a hash and output index pair. Hash is the input transaction hash and index point to the input transaction, which will be spent on this transaction.
Output states	It is an output transactions, which includes the contract and data.
Attachments	Transactions has an attachment in the form of zip file. Each zip file may contain code, data, certificates or supporting documentation for the transaction. Contract code has access to the contents of the attachments when checking the transaction for validity. In this field hash of the attachment will be available as pointer to the zip files.
Commands	It is a parameter to the contract that specifies more information about the transaction. Each command has an associated list of public keys.
Signatures	This is for the transaction owner's signature and the number of signature will be equal to the union of commands public keys.
Type	It refers the type of transaction. Transaction may be of general or notary-change. The validation rules are different for each type.
Timestamp	This is to indicate the time on which the transaction is generated.
Summaries	This is to contain the information about what the transaction does, checked by the involved smart contracts, etc..

contains following information.

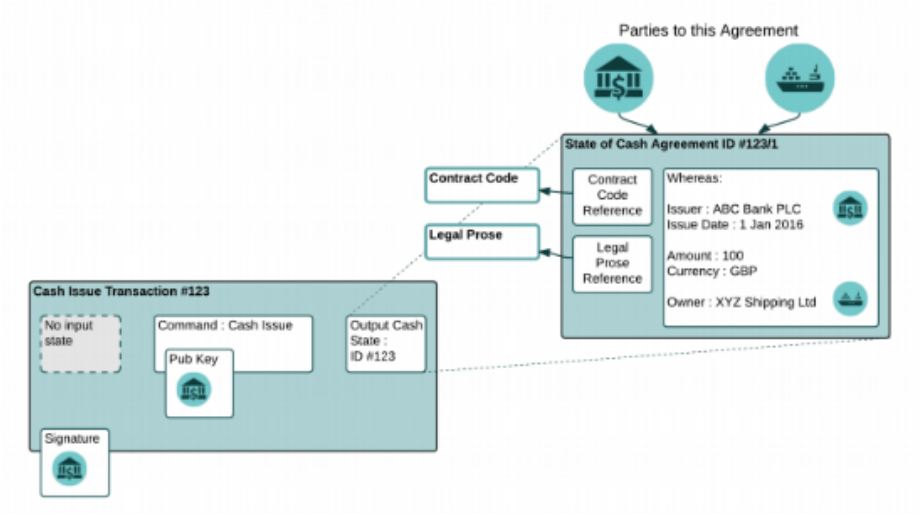

Figure 1.14: Corda Transaction [23]

- 1) Details about the cash that has been issued. The details are amount, currency, issuer, owner and issue date.
- 2) The contract code, for example *verify()* function, which is responsible for verifying this issuance transaction.
- 3) A hash of a document which may contain legal prose. Figure 1.15 shows an example legal prose and its hash in the state.

The transaction contains a command, which specifies that the intent of this transaction is to issue cash and the command specifies a public key. The *verify()* function is responsible for checking that the public key specified on the command is of the party whose signatures would be required to make this transaction valid. In this case, it means that the *verify()* function must check that the command has specified a key corresponding to the identity of the issuer of the cash state. The Corda framework ensures that the transaction has been signed by all keys listed by all commands in the transaction in case of multiple commands and parties. It means, a *verify()* function needs to ensure that all parties who need to sign the transaction are specified in commands and the framework responsible for ensuring that the transaction has been signed by all parties listed in all commands.

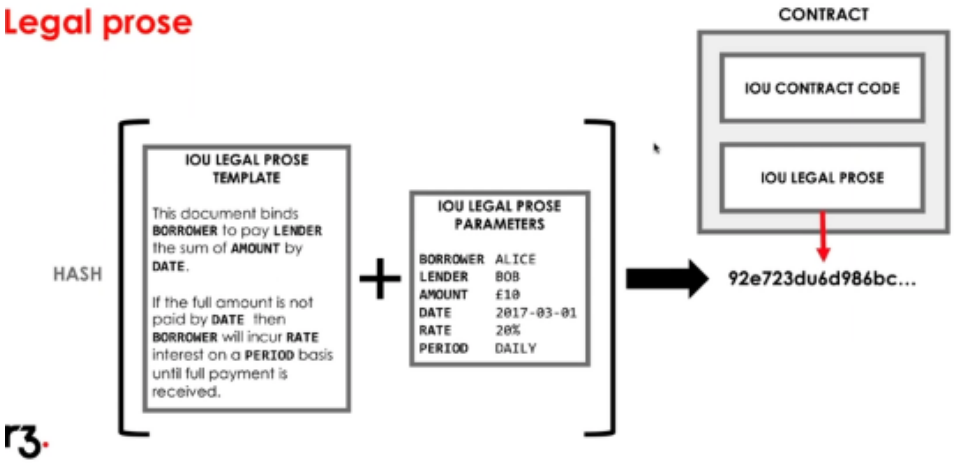

Figure 1.15: Corda Legal Prose [8]

1.2.2.4 Transaction chains

A transaction will contain input and output transactions. The output transactions is not exist and it is created by the proposer (s) of the transaction. However, the input transactions are already exist as the outputs of previous transactions. These input transactions are referred in the new transactions while it spent. These input states references are a pair of the following.

- *Hash* - The hash of the transaction that created the input
- *Index* - The inputs index in the outputs of the previous transaction

Figure 1.16 shows an example of the transaction chain. The transaction with hash *1ba0..* has the reference to two input transactions *95fd* and *0dbf*. The actual references are the New_State_2 of the *95fd* and New_State_1 of the *0dbf*. The input states pair are (*Hash:1ba0..,Index:1*) and (*Hash:0dbf..,Index:0*). According to implementation, the output states or transactions will be indexed from 1 to n but when it is referenced in spent transaction as input then it will be indexed from 0 to $n-1$ because of this pairs are having index as 1 and 0 instead of 2 and 1.

1.2.2.5 Committing transactions

Initially, a transaction is just a proposal to update the ledger. It represents the future state of the ledger that is desired by the transaction builder(s):

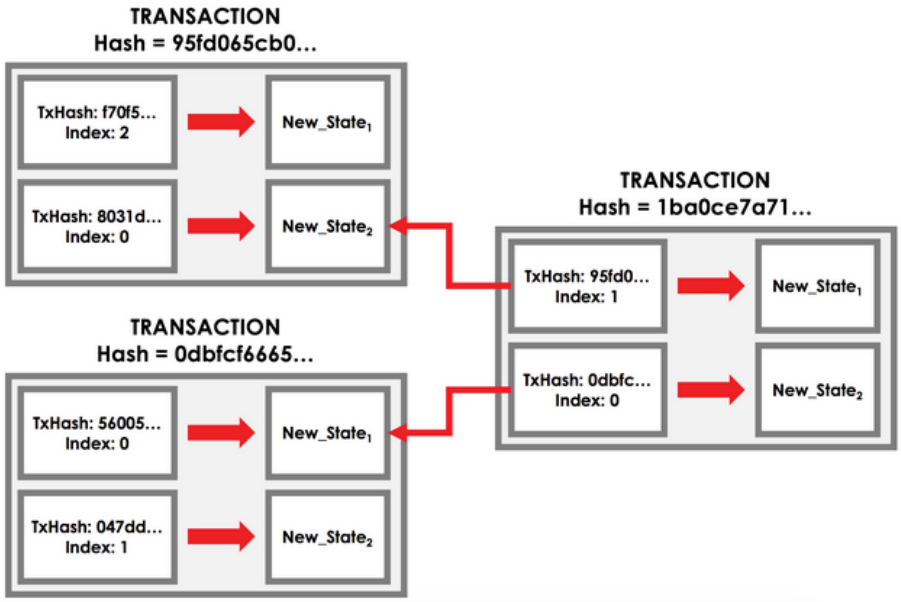

Figure 1.16: Corda Transaction Chain [14]

To become reality, the transaction must receive signatures from all of the required signers (see Commands, below). Each required signer appends their signature to the transaction to indicate that they approve the proposal:If all of the required signatures are gathered, the transaction becomes committed: This means that:

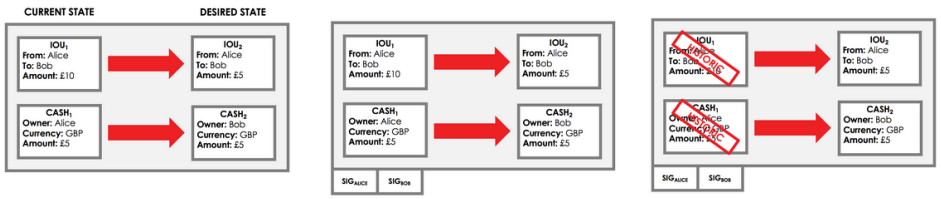

Figure 1.17: Corda Transaction [14]

- The transactions inputs are marked as historic, and cannot be used in any future transactions
- The transactions outputs become part of the current state of the ledger

1.2.2.6 Transaction Validity

Every transaction of the Corda will be validated based on its data and signature. Following validations will be done on the transaction before considering it for updating the ledger.

- The transaction is digitally signed by all the required parties.
- The transaction is contractually valid
  - Each transaction state specifies a contract type such as Cash Contract, Bond Contract, etc.
  - A contract takes a transaction as input, and states whether the transaction is considered valid based on the contracts rules.
  - Validate the number of inputs, outputs, commands, time-window, and/or attachments
  - The contract of every input state and every output state are valid.
- Transaction uniqueness: No other committed transaction that has consumed any inputs of the proposed transaction.

1.2.2.7 Reference states

There are states referred by the contracts of other input or output states but not updated/consumed called as reference state. A state added to the reference list of a transaction, instead of the inputs or outputs list, then it is treated as a reference state not an regular state. There are two important differences between regular states and reference states

- The specified notary for the transaction check whether the reference states are current. However, reference states are not consumed when the transaction containing them is committed to the ledger.
- The contracts for reference states are not executed for the transaction containing them.

Why is this type of reference data required? In blockchain systems everyone is sure they see the same as their counterpart - and for this to work in situations where accurate processing depends on reference data requires everybody to be operating on the same reference data.

Use Case: The KYC data can be distributed as reference data states and only updatable by the data owner. However, it is usable by any party and Notary ensures the data is current.

Regular contract states

Currently, the transaction model is too cumbersome to support reference data as unconsumed states for the following reasons:

Contract verification is required for the ContractStates used as reference data. This limits the use of states, such as Cash as reference data (unless a special reference command is added which allows a NOOP state transaction to assert no that changes were made.) As such, whenever an input state reference is added to a transaction as reference data, an output state must be added, otherwise the state will be extinguished. This results in long chains of unnecessarily duplicated data. Long chains of provenance result in confidentiality breaches as down-stream users of the reference data state see all the prior uses of the reference data in the chain of provenance. This is an important point: it means that two parties, who have no business relationship and care little about each others transactions nevertheless find themselves intimately bound: should one of them rely on a piece of common reference data in a transaction, the other one will not only need to be informed but will need to be furnished with a copy of the transaction. Reference data states will likely be used by many parties so they will be come highly contended. Parties will race to use the reference data. The latest copy must be continually distributed to all that require it.

Attachments

Of course, attachments can be used to store and share reference data. This approach does solve the contention issue around reference data as regular contract states. However, attachments dont allow users to ascertain whether they are working on the most recent copy of the data. Given that its crucial to know whether reference data is current, attachments cannot provide a workable solution here.

The other issue with attachments is that they do not give an intrinsic format to data, like state objects do. This makes working with attachments much harder as their contents are effectively bespoke. Whilst a data format tool could be written, its more convenient to work with state objects.

Oracles

Whilst Oracles could provide a solution for periodically changing reference data, they introduce unnecessary centralisation and are onerous to implement for each class of reference data. Oracles dont feel like an optimal solution

here.

Keeping reference data off-ledger

It makes sense to push as much verification as possible into the contract code, otherwise why bother having it? Performing verification inside flows is generally not a good idea as the flows can be re-written by malicious developers. In almost all cases, it is much more difficult to change the contract code. If transaction verification can be conditional on reference data included in a transaction, as a state, then the result is a more robust and secure ledger (and audit trail).

Versioning

This can be done in a backwards compatible way. However, a minimum platform version must be mandated. Nodes running on an older version of Corda will not be able to verify transactions which include references. Indeed, contracts which refer to references will fail at run-time for older nodes.

Privacy

Reference states will be visible to all that possess a chain of provenance including them. There are potential implications from a data protection perspective here. Creators of reference data must be careful not to include sensitive personal data.

1.2.2.8 Composite Key

Using a tree data structure, Corda represents composite public keys, which are used to represent the signing requirements for multi-signature scenarios. A composite key is a list of leaf keys and their contributing weight, and each leaf can be a conventional single key or a composite key. The nodes of a tree has weights of each child and a threshold weight that must be met. For example in figure 1.18 (b), *Alice* and *Bob* have weight 1 and *Charlie* has weight 2. The threshold required is 2, so *Alice* and *Bob* both have to sign or *Charlie* alone can sign. The validity of a set of signatures can be determined by walking the tree bottom-up, summing the weights of the keys that have a valid signature and comparing against the threshold. Using weights and thresholds, a variety of conditions can be encoded, including boolean with AND and OR. For example, figure 1.18(a) shows that the root node needs signature with threshold of 1 and it may be from *Charlie* or *Alice* and *Bob*. A key can contribute its weight to the total if it is matched by the signature. The figure 1.18(a) and (b) are having same threshold but with different representation.

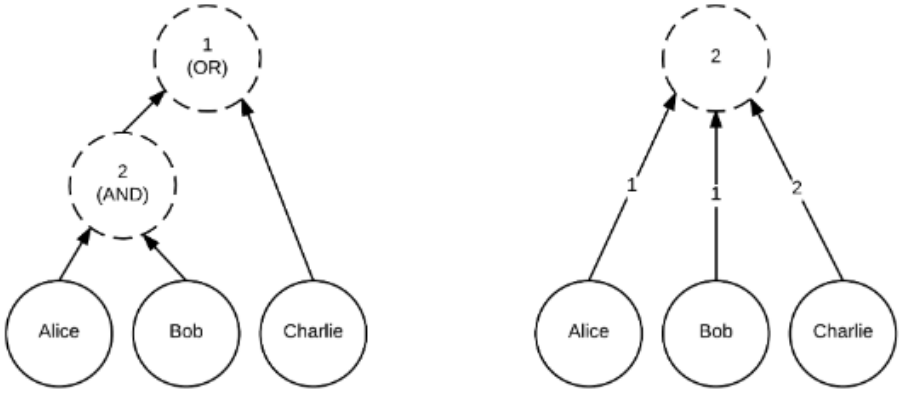

Figure 1.18: Corda Composite Key [14]

Use of Composite keys: For example, an assets can be placed under the control of a 2-of-2 composite key where one leaf key is owned by a user, and the other by an independent risk analysis system. The risk analysis system refuses to sign if the transaction made by a user seems suspicious, like if too much value has been transferred in short span.

Time-Window: It is not possible to have clock synchronisation between the party creating the transaction due to issues of physics (manufacturing components), network latency and other reasons such as notary approval, etc. The time at which the transaction is sent for notarization may be quite different from the time at which the transaction was created. Hence, times in transactions are specified as time windows in Corda. In some cases, user want a transaction proposed to be approved during a certain time-window. For example, an option can only be exercised after a certain date, a bond may only be redeemed before its expiry date, etc.In such case, we can add a time-window to the transaction. A Transaction timestamp specifies a [start, end] time window within which the transaction is asserted to have occurred. A notary can act as the timestamping authority that is before notarising a transaction, it verifies that a transaction occurred during a specific time-window or not [16].

1.2.2.9 Hard Fork

Unlike blockchain implementations, Corda does not have *hard fork*, so the only solution to discard fraudulent transaction chains would be to mutually agree out of band to discard an entire transaction subgraph. Since Corda nodes does not have global visibility, this mutual agreement would not need to encompass all network participants hence only those who have received and processed such transactions has to agree discard. However, the problem of global visibility is that there is no single point that records who exactly has seen which transactions. Determining the set of entities that would agree to discard a subgraph is through the correlating node activity logs. Corda nodes log sufficient information to ensure this correlation can take place.

1.2.2.10 Identity lookups

In blockchain system, a node making transaction want to know the counter party but not others to know about the transaction. To ensure it, a standard technique is to use randomised public keys in the shared data, and keep the knowledge of the identity that key maps to private. Also it is advised to you fresh key for every transactions to ensure the privacy. Corda takes this concept by encapsulating an identity and a public key in the Party field of state. For example, when a state is deserialised from a transaction in its raw form, the identity field of the Party object is null and only the public key is present. If a transaction is deserialised in conjunction with X.509 certificate then the transient public key and identity field is set. In this way a single data representation can be used for both the anonymised case and the identified case.

1.2.3 Oracle

There will be requirement sometimes to reveal a small part of a transaction to a counter-party to allow them to check the signatures and sign it. A typical use case for such case is an oracle, which is a network service that is trusted to sign transactions containing statements only if the statements are true. The oracle might check statements like as follows.

- The price of a stock at a particular moment was X.
- An agreed upon interest rate at a particular moment was Y.

- If a specific organisation has declared bankruptcy.
- Weather conditions in a particular place at a particular time.

This can be done by the smart contract also instead of doing through an Oracle service. The reason is that all calculations on the ledger must be deterministic. In the network, everyone must be able to check the validity of a transaction and arrive at exactly the same answer, at any time, on any kind of computer. If a smart contract could do things like read the system clock or fetch arbitrary transaction then it would be possible for some computers to conclude a transaction was valid, however others concluded it was not (e.g. if the remote server had gone off line). The solution to solve this problem is that all the data needed to validate the transaction must be in the ledger so that there cannot be disagreement among the nodes on a transaction.

An oracles would check the transaction and sign a small data structure which is then embedded somewhere in a transaction. An oracle uses the filtered transaction approach to *turn off* or *tear off* the unrelated parts of the transaction before the transaction is sent. It means, the transaction part which should not be seen by a node will be hided to ensure the privacy. This is achieved by structuring the transaction as a Merkle hash tree and the root hash will be used for signing operation. The counterparty can sign the entire transaction however only being able to see some of it. Additionally, if the counterparty needs to be convinced that some third party has already signed the transaction, that can be verified by supplying the necessary path or branch values to compute the root hash. An oracle with the counterparty will get the Merkle branches of the command or state that contains the data, and the timestamp field, and nothing else to validate the transaction.

1.2.3.1 Transaction Merkle Tree

A transaction merkle tree is constructed by splitting the transaction into leaves, where each leaf contains either an input, an output, a command, or an attachment [13]. It will also contains the other fields of the transaction, such as the time-window, the notary and the required signers as shown in figure 1.19.

The transaction in figure 1.19 has three input states, two output states, two commands, one attachment, a notary and a time-window. Notice that if a tree is not a full binary tree, leaves are padded to the nearest power of 2 with zero hash for example a node mentioned as *zero*. The hash of the root is

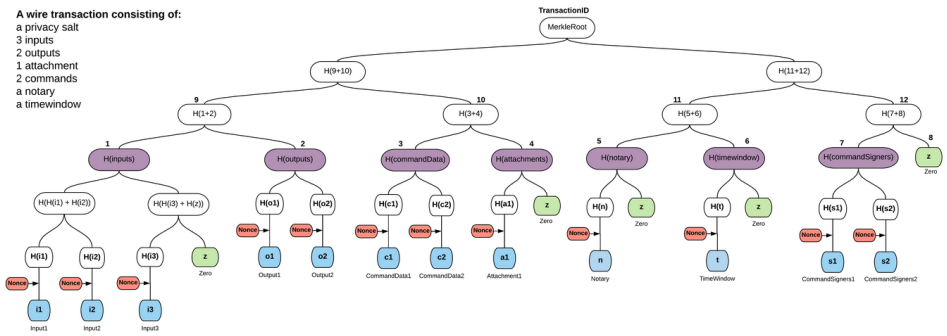

Figure 1.19: Corda Transaction Merkle Tree [13]

the identifier of the transaction and it will be used for signing and verification of data integrity. With merkle property, if any change in a transaction leaf level then it will change the transaction identifier.

Using merkle tree, hide data and provide the proof by constructing partial Merkle trees that is Merkle branches. A Merkle branch is a set of hashes computed from the leaves data, is used to calculate the root hash. The calculated hash is compared with the hash of a whole transaction and if they match it means that data we obtained belongs to that particular transaction otherwise not. Let us see an example of hiding data at the same time verifying the identity of the transaction. We take the case that only the first command of the transaction and all signers should be visible to an Oracle from the figure 1.19. It should provide guarantees that all of the commands requiring a signature from this oracle should be visible to the oracle entity, but not the rest. The filtered transaction for the oracle is shown in figure 1.19.

The node $c1$ shown in blue needs the signature of the Oracle. Hence blue node and $H(c2)$ are provided to the Oracle service, while the black ones are omitted. The $H(c2)$ is required for the Oracle to compute $H(commandData)$ without being to able to see the second command, but at the same time ensuring $CommandData1$ is part of the transaction. Also all signers ($CommandSigners1$ and $CommandSigners2$) are visible, so as to have a proof that no related command has been maliciously filtered out for the node Oracle. Additionally, hashes of sub-trees that is in violet colored nodes are also provided. With these data, Oracle can calculate the root of the top tree and compare it with original transaction identifier. If it matches then this command $CommandData1$ and time-window belong to this transaction.

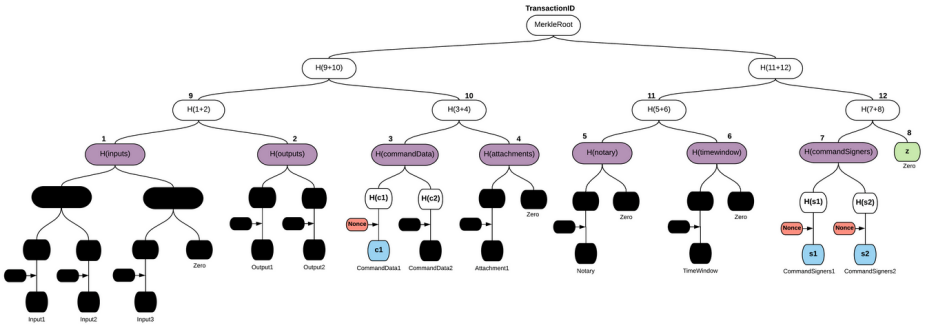

Figure 1.20: Corda Transaction Merkle Tree - Data Hiding [13]

1.2.3.2 Flow

In Corda, transaction data is not globally broadcast instead it is transmitted only to the relevant parties according to the requirement. For example, a coin transaction may involve counter parties and the third party such as a notary. Unlike traditional blockchain systems in which the primary form of communication is global broadcast, in Corda all communication takes the form of small multi-party sub-protocols called flows. The Corda flow provides the following service to distribute the transactions to update the ledger.

- *Identity to IP address mapping*: The transactions are constructed using the identities. The Corda flow framework takes care of routing messages (transactions) to the right IP address for a given identity.
- *Progress reporting*: The Corda flow can provide a progress tracker that indicates which step they are up to. Steps can have human-meaningful labels, along with other tagged data like a progress bar. Progress trackers are hierarchical and steps can have sub-trackers for invoked sub-flows.
- *Flow hospital*: Transactions can be paused by Corda flow, if transaction validation throw exceptions or explicitly request human assistance. A flow that has paused will be available in the flow hospital storage, where the node's administrator may decide to kill the flow or provide it with a solution based on the transaction correctness.

The automated process of Corda flow is shown in figure 1.21. The user *Alice* sends a transaction to her counter party *Bob*. The transaction is sent

from *Alice* after adding the signature. On the receipt of a signed transaction, *Bob* verify the transaction, sign, commit and send it to *Alice*. *Alice* verifies the received signed transaction and commit it. The commit in the flow indicates the update of ledger. The figure 1.21 does not include the notary for transaction verification and commit.

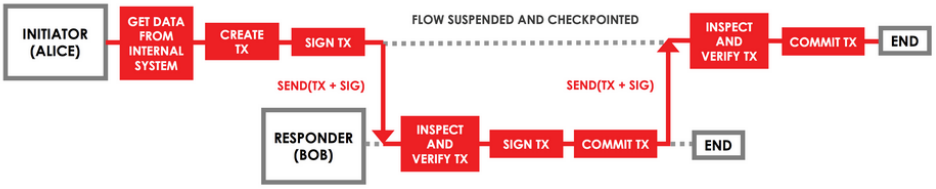

Figure 1.21: Corda Flow [9]

Inter-node communication In Corda, every node has zero or more flow classes that are registered to respond to messages from a single other flow. For example, let us assume a node *Alice* on the network, wishes to agree a ledger update with an another node *Bob*. To communicate with *Bob*, *Alice* must start a flow that *Bob* is registered to respond to and send a message within the context of that flow. On the receipt of message Bob will start its registered counter party flow. Finally, the connection will be established between *Alice* and *Bob* for passing a series of messages back and forth to update the ledger.

Subflows: A Flow can be composed by starting a flow as a subprocess in the context of another flow. The flow that is started as a subprocess is called as sub-flow and a flow that is started the subprocess is called as parent-flow. The parent flow will wait until the sub-flow returns.

Dependency Resolution: A node in Corda be able to present the entire dependency graph for a transaction to another node while asking to accept the transaction. In such case, there will never be any confusion about where to find transaction data. As we discussed, transactions are always communicated inside a flow, and flows embed the resolution flow, the necessary dependencies are fetched and checked automatically from the correct peer. Transactions propagate around the network lazily and there is no need for distributed hash tables.

1.2.3.3 Consensus

Determining whether a proposed transaction is a valid ledger update involves reaching two types of consensus:

- Validity consensus - this is checked by each required signer before they sign the transaction
- Uniqueness consensus - this is only checked by a notary service

Validity consensus

Validity consensus is the process of checking that the following conditions hold both for the proposed transaction, and for every transaction in the transaction chain that generated the inputs to the proposed transaction:

The transaction is accepted by the contracts of every input and output state The transaction has all the required signatures

It is not enough to verify the proposed transaction itself. We must also verify every transaction in the chain of transactions that led up to the creation of the inputs to the proposed transaction.

This is known as walking the chain. Suppose, for example, that a party on the network proposes a transaction transferring us a treasury bond. We can only be sure that the bond transfer is valid if:

The treasury bond was issued by the central bank in a valid issuance transaction Every subsequent transaction in which the bond changed hands was also valid

The only way to be sure of both conditions is to walk the transactions chain. We can visualize this process as follows:

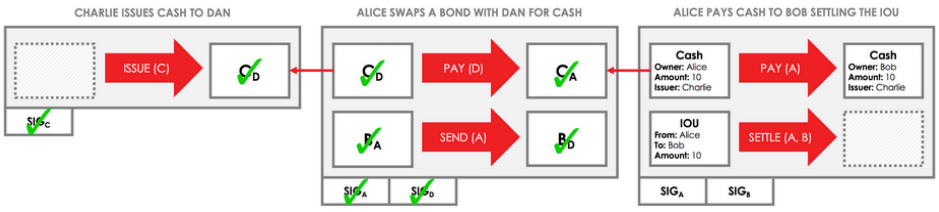

Figure 1.22: Corda Consensus [7]

When verifying a proposed transaction, a given party may not have every transaction in the transaction chain that they need to verify. In this case, they can request the missing transactions from the transaction proposer(s).

The transaction proposer(s) will always have the full transaction chain, since they would have requested it when verifying the transaction that created the proposed transactions input states.

Uniqueness consensus

Imagine that Bob holds a valid central-bank-issued cash state of 1,000,000. Bob can now create two transaction proposals:

A transaction transferring the 1,000,000 to Charlie in exchange for 800,000
A transaction transferring the 1,000,000 to Dan in exchange for 900,000

This is a problem because, although both transactions will achieve validity consensus, Bob has managed to double-spend his USD to get double the amount of GBP and EUR. We can visualize this as follows:

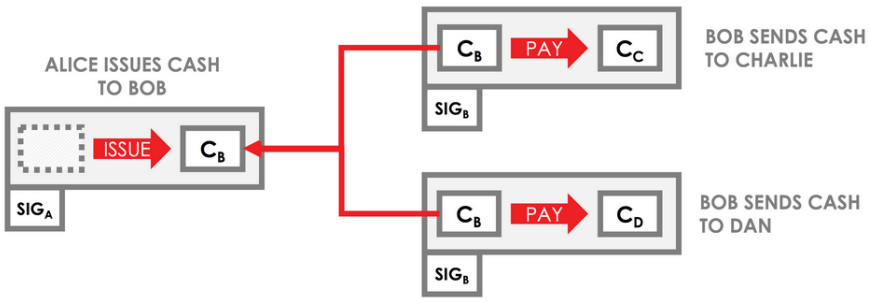

Figure 1.23: Corda Uniqueness Consensus [7]

To prevent this, a valid transaction proposal must also achieve uniqueness consensus. Uniqueness consensus is the requirement that none of the inputs to a proposed transaction have already been consumed in another transaction.

If one or more of the inputs have already been consumed in another transaction, this is known as a double spend, and the transaction proposal is considered invalid.

1.2.3.4 Notary

Notary is a service that provides transaction ordering and timestamping. It is composed of multiple mutually distrusting parties, which uses standard consensus algorithm. Notaries accept transactions submitted to them for processing and return a signature over the transaction if it is valid, or a rejection error that states that a double spend attempt has occurred. Every

state or transaction has an appointed notary cluster, and those notary cluster will only notarise a transaction if it is the appointed notary cluster of all the transactions input states.

1.2.3.5 Data distribution

In Corda, nodes will not have global ledger because most transactions are irrelevant for you and having to download them just wastes resources. In some case, node wish to aware about the ledger to identify the double spend. In such case, the node can request the notary to issue all approved transactions by sending the certificate. Once the notary cluster has committed the transaction, key identities are looked up and if any then copies of the transaction will be sent.

Consensus algorithms

Corda allows notary clusters to choose a consensus algorithm based on their requirements in terms of privacy, scalability, legal-system compatibility and algorithmic agility. A notary cluster may be a single node, several mutually-trusting nodes, or several mutually-distrusting nodes. Notaries may choose to run a high-speed, high-trust algorithm such as RAFT, a low-speed, low-trust algorithm such as BFT, or any other consensus algorithm it chooses

Validation A notary cluster can decide whether to provide consensus or not to any transactions. This decision can bring the following tradeoff.

- If a transaction is not checked for validity (non-validating notary), it creates the risk of denial of state attacks, where a node knowingly builds an invalid transaction consuming some set of existing states and sends it to the notary cluster, causing the states to be marked as consumed - Notary cluster can store the identity of the party that created the denial of state transaction, allowing the attack to be resolved off-ledger.

- If the transaction is checked for validity (validating notary), the notary will need to see the full contents of the transaction and its dependencies. This leaks potentially private data to the notary cluster - This problem can be solved using freshly generated public keys.

Multiple notaries

Each Corda network can have multiple notary clusters, each potentially running a different consensus algorithm. Multiple notaries provides the following benefits.

- Privacy - Both validating and non-validating notary clusters will be available on the same network, each running a different consensus algorithm. This allows nodes to choose the preferred notary cluster on a per-transaction basis. Hence, the node has provision to choose the notary and different notaries for different transactions, privacy of the node may be preserved.
- Load balancing - Spreading the transaction load over multiple notary clusters allows higher transaction throughput for the Corda platform.
- Low latency - Latency can be minimized by choosing a notary cluster physically closer to the transacting parties.

There are cases in which we may need to change a state's appointed notary cluster. These include:

- When a single transaction needs to consume several states that have different appointed notary clusters
- When a node would prefer to use a different notary cluster for a given transaction due to privacy or efficiency concerns

1.2.3.6 compatibility zone

Every Corda node is part of a *zone* also called a Corda network that is permissioned network with as secure certificate authority. The term *zone* is used because it is a set of technically compatible nodes reachable over a TCP/IP connection like the internet. In some case, a network in Corda can be called business network, which is usually more like a membership list or subset of nodes in a zone that have agreed to trade with each other.

1.2.3.7 Network parameters

There are parameters predefined in the Corda network that every node participating in the network or zone needs to agree on and use to correctly interoperate with each other. In other blockchain or decentralized ledger systems, source code fork is used to alter various constants such as the total number of coins in a cryptocurrency, port numbers to use etc. In Corda, the changes are sent as a separate file and allow zone operators to make decisions on it. There are various reasons that may lead to change of parameters for

example adding a notary, setting new fields that were added to enable smooth network interoperability, or a change of the existing compatibility constants is required, etc. If the network operator wish to update the parameter then the network map service starts to advertise the additional information with the usual network map data. An additional information includes new network parameters hash, description of the change and the update deadline. After the advertisement, nodes query the network map server for the new set of parameters.

The node administrator can review the change and decide if they are going to accept it. The approval should be sent before the update Deadline. Nodes that do not approve before the deadline will likely be removed from the network map by the zone operator, but that is a decision that is left to the operator's discretion. For example the operator might also choose to change the deadline instead. The new parameters advertised will overrides the previous set that means only the latest update can be accepted.

1.2.3.8 Smart Contract

A smart contract is a pre-defined protocol which is intended to digitally facilitate, verify, or enforce the negotiation or performance of a contract between two or more nodes. The smart contracts in Corda have three key elements:

- Executable code - It is for validating the changes to state objects in transactions
- State objects - It is the data held on the ledger, which represent the current state of an instance of a contract, and are used as inputs and outputs of transactions
- Commands - It is an additional data included in transactions to describe what is going on, used to instruct the executable code on how to verify the transaction. For example an Issue command may indicate that the validation logic should expect to see an output which does not exist as an input, issued by the same entity that signed the command.

Figure **??** shows an sample contract of the Corda for

The lifecyle of a new contract is, how are they issued, what happens to them after they are issued, and how are they destroyed. For the commercial

paper contract in figure **??**, states are issued by a legal entity which wishes to create a contract to pay money in the future (the maturity date), in return for a lesser payment now. They are then transferred to another owner as part of a transaction where the issuer receives funds in payment, and later that is after the maturity date it is destroyed (redeemed) by paying the owner the face value of the commercial paper.

Contract Sandbox: In Corda, transaction verification must be deterministic that is a contract should either always accept or always reject a given transaction. For example, transaction validity should not depend on the time at which validation is conducted, or the amount of information the peer running the contract holds. This is a necessary condition to ensure that all peers on the network reach consensus regarding the validity of a given ledger update. In future version, Corda will follow the deterministic sandbox. The sandbox will have the white list that prevents the contract from importing libraries that could be a source of non-determinism. For example, these libraries includes the current time, random number generators. Eventually, the information available to the contract when verifying the transaction is the information included in the transaction itself.

1.2.3.9 Privacy

Corda improves user privacy over other distributed ledger systems

- *Partial Data Visibility*: In Corda, transactions are not required to broadcast globally, it is only for the intended users of the network. Hence, the transaction information is visible only to the respective users.

- *Transaction tear-offs*: As discussed earlier, transactions are structured as Merkle trees and the individual subcomponents are revealed to parties who already know the Merkle root hash. Those parties can sign the transaction without being able to see all of it.

- *Key Randomization*: Like other cryptocurrency implementation, the vault uses random keys that are unlinkable to an identity. The key and identity cannot be linked without the corresponding linkage certificate.

- *Graph pruning*: Large transaction graphs that involve liquid assets can be pruned by requesting the asset issuer to re-issue the asset onto the

ledger with a new reference field. For example, a transaction having many predecessors can be pruned, revised transaction can be prepared and sent to the network. This operation unlinks the revised transaction from the old assets and nodes will get chance to explore the original dependency graph during verification.

- *Mix networks*: There are chances that some users or nodes learns about transactions that are not directly related to trades they are doing, for example notaries or regulator nodes. Even though, key randomisation is used, these nodes can still learn valuable identity information by simply examining the source IP addresses or the authentication certificates of the nodes sending the data for notarisation. The traditional cryptographic solution to fix this problem is mix network and the most famous Tor, but a more appropriate design for Corda would be that of an anonymous remailer. In a mix network, a message is repeatedly encrypted like a layer using keys owned by a small set of randomly selected nodes. Each layer contains the address of the next hop in the network. The first hop decrypts the message first layer and reveal the next encrypted layer and forwards it to second hop. The second hop decrypts the message second layer and reveal the next encrypted layer and forwards it to the next and similarly till the destination. Adding mix network in Corda protocol may upgrade user privacy.

1.3 Algorand

Algorand is an implementation of blockchain based cryptocurrency and uses Byzantine Agreement (BA* - '*' is added because of scalability) to achieve the distributed consensus [22]. It confirms the transaction faster than other cryptocurrencies, which uses Proof of Work (PoW). Also it is scalable to many users and ensures no fork on the chain. This section gives the brief overview of Algorand.

1.3.1 How it works

Users wish to join the Algorand network requires to have a private and public key pair. Like Bitcoin and other cryptocurrencies, the keys are used for the transaction and owning the coin. The transaction is a payment signed by one

user's private key transferring money to another user's public key. Algorand grows the blockchain in asynchronous rounds like Bitcoin and Ethereum. In every round, a new block, containing a set of transactions and a pointer to the previous block, is constructed and appended to the blockchain. The eligible users identified through a Verifiable Random Function (VRF) [25] generate the blocks with the available pending transactions and communicate with other users in the network through a gossip protocol. After receiving the blocks, eligible users run the Byzantine Agreement to reach the consensus on one of the blocks, which will be appended with the blockchain. In our discussion, Byzantine Agreement ($BA*$) is used as BA for better representation.

1.3.2 Features

Algorand is the energy efficient and fast confirmation cryptocurrency. It utilized different features to achieve the efficiency. The features used in the implementation of the Algorand are discussed in the following.

- *Byzantine Agreement*: A group of nodes needs to agree on some value. The protocol needs to
  - (a) Terminate within finite time.
  - (b) Have all honest nodes reach the same result.
  - (c) The result reached must match the input of one of the nodes.
- *Gossip protocol*: Like Bitcoin, Algorand implements a gossip network where each user selects a small random set of peers to gossip messages to. To ensure messages cannot be forged, every message is signed by the private key of its original sender; other users check that the signature is valid before relaying it. To avoid forwarding loops, users do not relay the same message more than once until it is required.
- *Weighted users*: In Algorand, every user is assigned with a weight. The byzantine agreement protocol is designed to guarantee consensus as long as a weighted fraction (a constant greater than 2/3) of the users are honest. Users are weighted based on the money in their account. An adversary can also participate in Algorand and own some money. In order to successfully attack Algorand, the attacker must invest substantial financial resources in it.

- *Consensus Committee*: A small set of representatives that is the randomly selected set of users participate to run the protocol for consensus. The weight of the user is one of the parameters to choose the users for the committee because more the weight user have are honest. However, relying on a committee creates the possibility of targeted attacks against the chosen committee members.

- *Cryptographic sortition*: To prevent an adversary from targeting committee members, Byzantine agreement selects committee members in a private and non-interactive way. This means that every user in the system can independently determine if they are chosen to be on the committee or not, by using the result of a random function called Verifiable Random Function (VRF), which takes input as user private key and public information from the blockchain. If the function produces the output, which indicates that the user is chosen, it returns a short string that proves this user's committee membership to other users, which the user can include in his network messages for other users verification. Since membership selection is non-interactive, an adversary does not know which user to target until that user starts participating.

- *Participant replacement*: An adversary may target a committee member once that member sends a message. The Algorand mitigates this attack by requiring committee members to speak just once. Thus, once a committee member sends his message, the committee member becomes irrelevant to the protocol. Each users are equally capable of participating if they have enough coin in their account and for every round new committee members will be elected.

- *Byzantine consensus*: In Practical Byzantine Fault Tolerance protocol, fixed set of servers to be determined ahead of time and allowing anyone to join the set of servers would open up the protocols to Sybil attacks. Byzantine Agreement does not rely on a fixed set of servers, which avoids the possibility of targeted attack son well-known servers. According to the user account balance, BA allows users to join the consensus without risking Sybil attacks.

- *Block Proposal*: Users execute cryptographic sortition to determine if they are selected to propose a block in a given round. Users are selected at random, weighed by their account balance, and provides each

selected user with a priority, which can be compared between users, and a proof of the chosen user's priority. The selected users generate and distribute their block containing pending transactions through the gossip protocol, together with their priority and proof. Users wait for a certain amount of time to receive the block and proceed further.

- *Block Agreement*: Above Block proposal does not guarantee that all users received the same block. To reach consensus on a single block, users again go for cryptographic sortition to check whether they have been selected as committee members in that step. The Committee members then broadcast a message which includes their proof of selection and these steps repeat until, in some step, enough users in the committee reach consensus on a Block.

- *Bootstrapping*: A common genesis block will be provided to all users, along with the initial cryptographic sortition seed. The initial seed specified in the genesis block is decided using distributed random number generation, after the public keys andweights for the initial set of participants are publicly known

- *Proof of Stake*: Each user in Algorand gets weight proportionally to its monetary value have in the system, inspired by proof-of-stake approaches.

- *Block Structure*: Block consists of a list of transactions, along with meta-data for Byzantine Agreement. The meta-data consists of the round number, the proposer's VRF-based seed, a hash of the previous block in the ledger, and a timestamp indicating when the block was proposed. Once a user receives a block from the proposer, the user validates the block contents before passing it on consensus steps. In particular, the user validates transactions, seed, hash of the previous block, block round number and timestamp. If any of the validation fails, the user passes an empty block to consensus step.

1.4 Exercise

1.

Bibliography

[1] How addresses are used in iota. [Online; accessed 23-April-2019].

[2] Masked authenticated messaging. [Online; accessed 26-April-2019].

[3] Transaction validation. [Online; accessed 18-April-2019].

[4] What is iota? a beginner's guide. [Online; accessed 22-April-2019].

[5] How does a full-node validate transactions? [Online; accessed 18-April-2019].

[6] What is the iota transaction data structure? [Online; accessed 18-April-2019].

[7] Consensus. [Online; accessed 04-May-2019].

[8] Contracts. [Online; accessed 16-May-2019].

[9] Flows. [Online; accessed 04-May-2019].

[10] Iota transactions, confirmation and consensus. [Online; accessed 26-April-2019].

[11] The ledger. [Online; accessed 04-May-2019].

[12] States. [Online; accessed 04-May-2019].

[13] Transaction tear-offs. [Online; accessed 19-May-2019].

[14] Transactions. [Online; accessed 04-May-2019].

[15] Vault. [Online; accessed 04-May-2019].

[16] Vault. [Online; accessed 04-May-2019].

[17] Breier, B. Technical analysis of the tangle in the iota-environment. *Bachelors Thesis in Informatics,DEPARTMENT OF INFORMATICS TECHNICAL UNIVERSITY OF MUNICH* (2017). [Online; accessed 18-April-2019].

[18] Courtois, N.T., D. M., and Felke, P. On the security of hfe, hfev- and quartz. *In Public Key Cryptography PKC 2003, volume 2567 of Lecture Notes in Computer Science, pages 337350. Y. Desmedt, ed., Springer* (2002).

[19] Cullen, A., Ferraro, P., King, C., and Shorten, R. Distributed ledger technology for iot: Parasitechain attacks. [Online; accessed 26-April-2019].

[20] Divya M, N. B. B. Iota - next generation block chain. *International Journal Of Engineering And Computer Science, Volume 7 Issue 4, pp.23823-23826* (2018). [Online; accessed 20-April-2019].

[21] Gal, A. Alpha: playing with randomness. [Online; accessed 20-April-2019].

[22] Gilad, Y., Hemo, R., Micali, S., Vlachos, G., and Zeldovich, N. Algorand: Scaling byzantine agreementsfor cryptocurrencies. [Online; accessed 21-May-2019].

[23] Hearn, M. Corda: A distributed ledger. [Online; accessed 07-May-2019].

[24] Lu, L. In-depth explanation of how iota making a transaction. [Online; accessed 18-April-2019].

[25] Micali, S., Rabin, M. O., and Vadhan, S. P. Verifiable random functions. *Proceedings of the 40th Annual IEEE Symposium on Foundations of Computer Science (FOCS)* (1999). [Online; accessed 21-May-2019].

[26] Popov, S. The tangle. [Online; accessed 18-April-2019].

Contents

List of Figures

Chapter 1

Smart Contracts

Blockchain's use of consensus to validate interaction amongst participant nodes is a key enabler for applications that require mutually distrusting peers to conduct business without the need for a trusted intermediary. One such use is to enable a smart contract, which programatically encodes rules to reflect any kind of multi-party interaction. With an increasing trend towards autonomous applications, smart contracts are fast becoming the preferred mechanism to implement financial instruments (e.g., currencies, derivatives, wallets, etc.) and applications such as decentralized gambling.

This chapter is divided into three sections, and focuses on the common use cases for smart contracts, and its implementation and working across major permissionless and permissioned Blockchain platforms.

- Smart Contract Introduction: This sections gives a primer on the workings of a smart contract and presents the common use cases where it is useful.

- Ethereum: This section discusses the smart contract representation and working in the Ethereum [12] permissionless Blockchain platform.

- Hyperledger Fabric: This sections focuses on the smart contract representation and working in Hyperledger Fabric [23, 31], which the most popular permissioned Blockchain platform.

1.1 Introduction

A key aspect of a Blockchain is that it represents a decentralized ledger system, which exists between the transacting parties. Thus, there is no requirement to pay intermediaries, thereby saving time, money and conflict. While Blockchains have operational constraints, they are more economical and secure than traditional systems, which is why banks and governments are adopting them [6, 14].

Nick Szabo [26], a legal scholar and cryptographer, is widely credited for inventing the idea of a smart contract [37]. He realized that the decentralized ledger system could be used for enabling self-executing, digital contracts (or smart contracts), which are 'a set of promises, specified in the digital form, including protocols within which the parties perform on these promises'. These contracts, which is legal tender between the parties that implement them, could be implemented as computer programs, stored and replicated on the decentralized ledger, and supervised by the network of computers that run the Blockchain. Such a framework would result in an automated exchange of good and services (having digital value) in a transparent, conflict-free way while avoiding expensive intermediaries.

1.1.1 Smart Contracts

A smart contract is a computer program that constitutes the logic to operate upon a logical representation of an asset, which can be either physical or digital in form. Specifically, this program, which has been agreed upon by multiple parties involved in the transaction, executes atop a Blockchain and implements the myriad rules of engagement amongst them and the outcomes (see Fig. 1.1). It accepts the digital asset as input, and executes the logic to automatically validate, without the need for intermediaries, whether at the end of the transaction the asset should go to one or more participants, or immediately refunded to the source who sent it, or some combination thereof. In other words, it enables decentralized automation by facilitating, verifying, and enforcing the conditions of an underlying agreement in a transparent manner eliminating the need for middlemen and keeping the system conflict-free.

To summarize, a smart contract has the following key characteristics:

1. It is encoded in digital form with the contractual clauses embedded as code in software (or hardware).

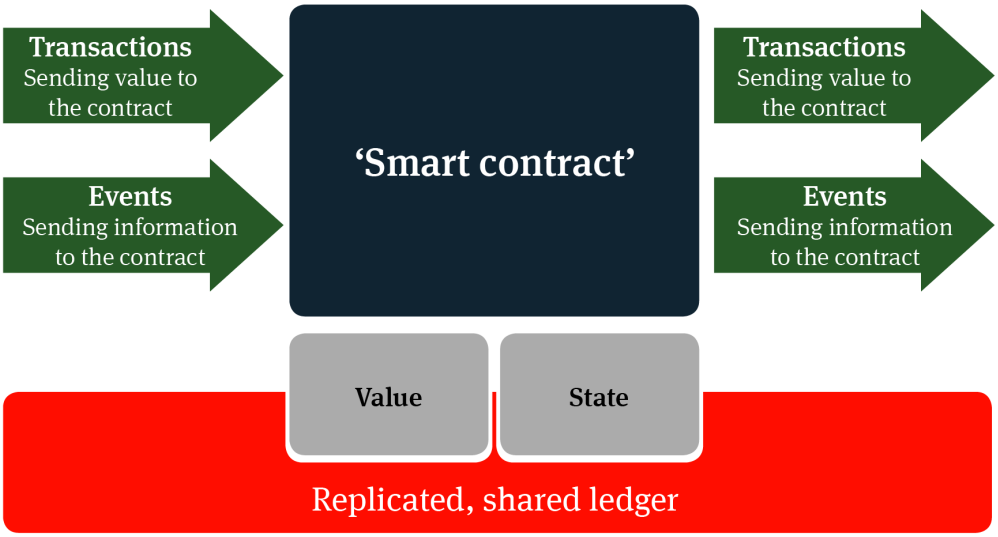

Figure 1.1: **A simple model for smart contracts (adapted from [1]).**

2. The actions are enabled by technology and rule-based operations.

3. It performs automatically and its actions are irrevocable, i.e., once initiated, the outcomes for which a smart contract is encoded to perform cannot be stopped (unless an outcome depends on an unmet condition).

How do smart contracts work?

In order to understand how a smart contract works in real world, let us consider a couple of examples.

1. Usually for getting a court-registered document as proof, one needs to visit a lawyer/notary, give them money for their services and wait till one gets the document that was required. However, with smart contracts the scenario changes completely. Specifically, one would simply get the document needed by paying just for it and this will be done without the involvement of any third-party such as the lawyer/notary in this case. Moreover, smart contracts are not limited to only defining the rules of an agreement but they are also responsible for the automatic execution of those rules and discharge any pending obligations.

2. Let us take a little more complex, real-world example and see how smart contracts can help. Consider that you need to sell a property of your own. The process of selling properties demands a lot of paperwork as well as communication with multiple parties. Other than the communication complexity, it also involves the risk of frauds.

   Today, most people who want to deal in properties make their way ahead through real-estate agents. These agents are responsible for dealing with the paperwork and markets. They act as intermediaries in the overall process and work on negotiations and overseeing deal. In such cases, you cannot simply rely on the person that you are dealing with, and therefore, the agencies provide escrow services which transfer the funds from one party to the other. When the deal is finalized, you will have to pay both, the agent and the escrow service their commission in terms of the decided percentages. This leads to an extra loss of money and more risk on the seller's end.

   However, in such situations, using smart contracts can result in more effectiveness by reducing the burden. Smart contracts are designed to work on rule-based principles (often encoded as the standard asynchronous `event` model coupled with the `if-then-else` ladder), which will resolve the ownership issue by transferring it to the buyer only when the monetary, as well as other conditions, are agreed upon. Moreover, when it comes to escrow services, smart contracts can replace those too. Both money and the right of possession of the property can be stored in a distributed system, which can be viewed by the involved parties in real-time. Since the money transfer will be witnessed by all the network participants, the chances of fraud are eliminated. Moreover, there is no chance of an intermediary to be involved as the trust between parties is not an issue anymore. All the functions performed by the estate agent can be coded into the smart contract, thus, saving a considerable amount of money on both, buyer and seller end.

Benefits of Smart Contracts

Smart contracts execute atop the Blockchain and offer several benefits. We now briefly enumerate them.

- **Autonomy**: Smart contracts require verification and validation from the various parties involved; there is no requirement to rely on a broker,

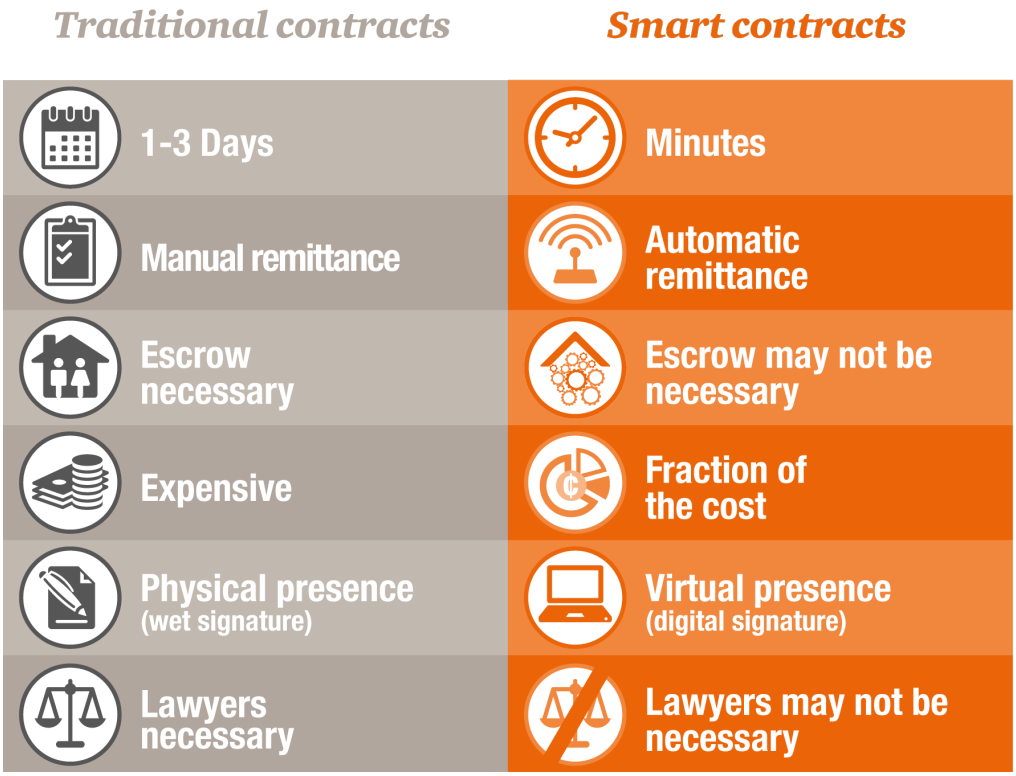

Figure 1.2: **The various benefits of using smart contracts** [21].

lawyer or other intermediaries to confirm. This level of autonomy also prevents the danger of manipulation by a third party, since a contract execution is managed automatically by the network itself, rather than by one or more, possibly biased, individuals who may err.

- **Transparency**: A key Blockchain characteristic that smart contracts build atop is transparency. As described earlier, a smart contract constitutes the terms of engagement amongst the various parties involved in the transaction. Since these terms are verified by the various parties involved in the agreement, it eliminates any chance of disputes and issues often arising at a later stage due to miscommunication or misrepresentation of information by the intermediaries. This characteristic of smart contracts enables transparency in multi-party interactions and improves the efficiency of communication.

- **Efficiency**: Current business practices often employ the use of intermediaries and paper documentation. This overhead usually incurs delays ranging from days to weeks. On the other hand, smart contracts autonomously execute transactions atop the Blockchain and the time to determine the results is only limited by the cost of consensus between the interacting participants, thereby saving huge amounts of time as compared to traditional business processes. Furthermore, errors due to manual involvement are also eliminated.

- **Precision**: A smart contract is often coded in explicit detail covering all possible terms and conditions typically required for multi-party interactions. Thus, a well-written smart contract offers no ambiguity; it is accurate and precise to the minutest level of the agreement. Since any missing condition may result in errors during contract execution, all participants verify and validate the smart contract code for correctness, fairness, and ambiguity before it is put to work atop the Blockchain. Unlike manual contracts, where human involvement often leads to errors and accountability is hard to manage, smart contracts offer a much better alternative while providing better accuracy and precision.

- **Trust**: Smart contracts are transparent and secure. They reduce any possibility of manipulation as well as manual errors. With smart contract data managed atop the Blockchain, the system offers a distributed, tamper-resistant, append-only logging scheme. In fact, it would take unrealistic amounts of resources to tamper with any computation. Autonomous self-execution coupled with trust in computation establishes confidence in smart contracts and enables participants to commit and abide by the rules of engagement as encoded within them.

- **Safety** Smart contracts work atop the Blockchain, which leverages state-of-the-art cryptographic primitives to implement a distributed, tamper-resistant append-only logging scheme. Thus, smart contracts are safeguarded by the same safety mechanisms that protect the Blockchain operations. Specifically, the use of cryptographic primitives enables much higher levels of confidentiality and integrity in operations as compared to manual efforts. Note that the Blockchain's open nature makes it hard to achieve complete confidentiality, and requires the use of novel hardware support, such as secure enclaves [32, 34].

- **Data Storage**: Smart contracts leverage the Blockchain for irrevocably recording transaction operations. Specifically, results from each and every transaction are stored in the Blockchain and are replicated amongst all participating entities, which further eliminates possibilities of any data loss. This ubiquity of tamper-resistant data records enables all concerned parties to access any Blockchain data at a given instance, thereby preventing disputes arising from potential data inconsistencies.

- **Savings**: Smart contracts significantly reduce unnecessary overheads, often involving intermediaries; they only need entities that are part of the actual transaction. Further, as smart contracts are automated computer programs, their use considerable reduces the paper footprint in the entire business process. While, on one hand, it saves on resources, smart contracts also offer increased security, transparency, and efficiency, quite unlike the paper-based manual process.

As is evident from the above features, by applying smart contracts to modern businesses and leveraging the multiple advantages, we can make phenomenal improvements over the traditional contracts and processes. Smart contracts offer the right blend of convenience, efficiency, security, and trust that makes it easier to streamline business workflows.

1.1.2 Applications of Smart Contracts

While the idea of smart contracts is fairly old [37], the present day world we live in still depends heavily on two aspects: paper-based contracts, and one or more trusted intermediaries. Even if a digital/smart contract were in place, it is extremely difficult to eliminate the extremely intertwined role of the trusted third-party from the system, even though over the years it has been evident that the involvement of third-party might lead to security issues or fraudulent activities. Thus, the system as defined currently is not always smooth.

Blockchain, however, has heralded a new era of innovation. With the introduction of Blockchain in the digital technology space, such issues can be addressed efficiently. A Blockchain-based system participating entities to interact with each other in a secure, distributed but still efficient manner, thereby eliminating the need of any trusted third-party. While Blockchain is the technology that underpins the Bitcoin cryptocurrency, it powers sev-

Figure 1.3: **The various use cases of smart contracts [21].**

eral applications other than cryptocurrency. Over the years, Blockchain has evolved and its use cases are coming forward in different industries.

Smart contracts can be termed as the most utilized and successful applications of the Blockchain technology in the current times. As listed earlier, use of smart contracts in place of traditional contracts can significantly remove the transaction overheads. All modern Blockchain platforms such as Ethereum [12] and Hyperledger Fabric [23, 31] support creating smart contracts. They allow the use of Turing-complete languages to enable developers to quickly script powerful smart contracts that are customized to the developer's needs or target an industry use-case. Modern use cases for smart contracts have spanned different industries and fields such as smart homes, e-commerce, real-estate and asset management etc. Below we discuss a few major use cases.

Government

With the increased sophistication in technology, it has become extremely hard to provide a tamper-resistant governance mechanism. One of the most critical government activity is to enable fair elections. However, there have been documented attacks on our voting system [8,17,35], which is now considered to be insecure and elections are often deemed to be rigged. Blockchain-based smart contracts allay such concerns by providing a provably more

secure system, since tampering Blockchain-protected votes would require excessive computing power. Furthermore, smart contracts can fix the problem of digital identity (and its thefts) [4, 7, 19], thereby allowing governments to carry out identity-based welfare schemes [11, 18, 27] without the threat of funds being siphoned off by intermediaries, as it happens today.

Management

Blockchain-based smart contracts not only provide trust, autonomy, and transparency by operating atop a common, secure distributed, tamper-resistant ledger (that acts as a source of trust), but also eliminate possible transaction delays in communication and workflow because of its accuracy and automation. In the traditional setting, business operations must endure numerous back-and-forth, while waiting for approvals or due to internal or external issues. Smart contracts streamline these problems by eliminating discrepancies associated with independent processing at the intermediary, which may lead to costly delays. Due to the robustness and power of smart contracts, they have been widely used in the financial sector.

For example, in 2015, the Depository Trust & Clearing Corp. (DTCC) used a blockchain ledger to process more than $1.5 quadrillion worth of securities, representing 345 million transactions [16]. Barclays uses smart contracts to log change of ownership and automatically transfer payments to other financial institutions upon arrival [2].

Supply Chain

Supply chains are often hampered by the numerous approvals that are required for goods to pass from one intermediary to another. Since, majority of these approvals often rely on manual, paper-based mechanisms, it exposes the system to huge amounts of delays and subsequent losses (and even frauds) incurred due to these delays. Blockchain-based smart contracts, in principle, eliminate such issues by providing a secure, accessible digital version to all parties on the chain and automates the tasks and payment. Further, since smart contracts work on rule-based principles, they also defend against any tampering and fraud attempts.

There are several examples where Blockchain's have revolutionized supply chains across various domains. Let us consider two examples.

- Blockchain is helping verify the authenticity, provenance, and custody of diamonds across the supply chain. Everledger has developed a Blockchain solution for the diamond supply chain that is designed to help prevent fraud and illicit global diamond trading. Its Blockchain stores diamond fingerprints corresponding to around 1.6 million diamonds [9].

- Bext360 is using Blockchain technology to better track all elements of the worldwide coffee trade—from farmer to consumer—and thereby boost supply-chain productivity. After pilots in California, Uganda, and Ethiopia during 2017, it is now live for Ireland's Moyee Coffee [30], which provides specialty coffee to offices, independent retailers, and online subscribers, expects to have all its coffee fully blockchain-traceable now [3, 29].

Real Estate

As explained earlier in § 1.1.1, smart contracts can ease handling of physical goods and assets, like real-estate. The challenges facing the real-estate industry include few property offerings, the dearth of investors investing in homes, home price increases, and a lack of transparency in the marketplace, among other concerns. One major benefit of Blockchain technology is the idea that tokens can represent physical assets rather than just items that have no existence offline. For example, if we wanted to rent an apartment, today we need to pay an intermediary to (a) advertise the property, (b) prepare the legal documents, and (c) confirm that rent has been paid on time every month. Blockchain-based smart contracts can cut such costs. However, with smart contracts and Blockchain, it can all be streamlined and done in a secure and trusted manner. As a result, several governments, including the European Union and Dubai, have initiated Blockchain-based real-estate initiatives [5, 10].

Healthcare

While compliance in the pharmaceutical supply chain, and tracking authenticity of returned drugs, etc., seem obvious applications of the Blockchain technology to the healthcare industry, other applications such as transparency and traceability of clinical trials and improving patient healthcare are also

possible using Blockchain.For example, personal health records could be encoded and stored on the Blockchain with a private key which would grant access only to specific individuals [25]. The same strategy could be used to ensure that research is conducted via HIPAA laws (in a secure and confidential way) [24]. Receipts of surgeries could be stored on a Blockchain and automatically sent to insurance providers as proof-of-delivery. The ledger, too, could be used for general healthcare management, such as supervising drugs, regulation compliance, testing results, and managing healthcare supplies. Since medical data will no longer exist in silos, and can be shared in a secure and trustworthy manner, patient care improves.

1.2 Ethereum

Ethereum [12] is often referred to as a general purpose, programmable Blockchain. It is essentially a distributed state machine that uses the Blockchain to track not only currency ownership but also state transitions to the general-purpose data store, i.e., the Ethereum Blockchain, which can hold any data expressible as a key-value tuple. This data storage model of Random Access Memory (RAM) used by most general-purpose computers.

Like a general-purpose stored-program computer, Ethereum can also load code into its state machine and execute that code, storing the resulting state changes in its Blockchain. Specifically, Ethereum allows for distributed execution of arbitrary code and provides a suitable platform for the design and development of smart contracts and decentralized applications. It supports several scripting languages that can be compiled to byte code that is executed on the Ethereum Virtual Machine (EVM).

However, there are two key differences that set Ethereum apart from most general-purpose computers. First, unlike usual imperative programs, the state in Ethereum is distributed and tracked at a global level, i.e., any modifications to a state variable is visible across all nodes in the Ethereum Blockchain network. Second, each state change is governed by the rules of consensus.

The following subsections discuss a few more key aspects of Ethereum.

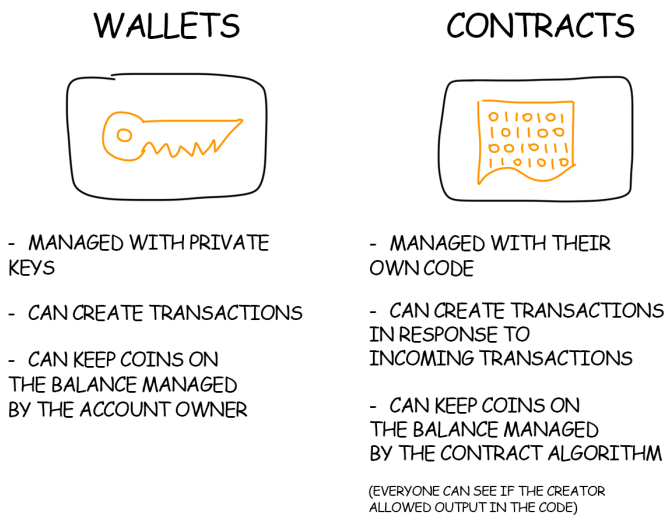

Figure 1.4: **Differences between wallet and contract accounts [20].**

Smart Contracts

In § 1.1, smart contracts were introduced as computer programs and equated in power akin to general purpose software applications. However, this analogy is not completely correct for Ethereum-based smart contracts, which are more akin to `class`es in conventional object-oriented programming.

Accounts

There are two types of accounts in Ethereum—a user/wallet account, and a contract account—and the EVM does not differentiate between them. A user/wallet account is managed by a private key, which is used to create and sign transactions. A contract account has its state managed by a smart contract. Specifically, on receipt of a message to a contract account, its corresponding smart contract code is executed. Every account maintains a balance and a key-value store (also called its storage)—both of which can be updated on send/receipt of transactions. Fig. 1.4 briefly lists the key differences between the two types of accounts.

Transactions

A transaction is a digitally signed message that contains details of the sender of the transaction, its recipient, the amount of ether sent, an optional data field, a gas limit, and a gas price field.

Messages

A message is similar to a transaction except that is sent from one contract to another contract, and resembles an ordinary function call in imperative programming languages. A message is created when the contract account executes a specific opcode, and its invocation typically modifies the storage state of the contract.

Ether

Similar to Bitcoin, Ethereum uses its own native currency called Ether, which can be exchanged against other fiat currencies via various cryptocurrency exchanges. It is primarily used to pay for invocation of publicly defined methods in a smart contract.

Gas

Gas is used to pay transaction fees in Ethereum. It is not a currency, but an internal pricing unit that decouples the market price of the ether from the cost of transactions. Gas price can be defined by the originator of a transaction and miners decide whether they want to include the transactions in their blocks or not. The more lines of code a smart contract executes and the more memory and storage it uses, the higher is the gas requirement.

1.2.1 Life Cycle of a Smart Contract

When one user uploads a smart contract through their Ethereum node, it is included in the latest block and propagated around the network, where it is stored on every other node in the network. Now each and every node in the network executes the same code, as part of the block processing protocol. The nodes go through the block they process and run any code enclosed within the transactions. Each node does this independently. As a result, the process is not only highly parallelized, but highly redundant. In other

words, every node in the entire distributed network performs every program executed on the platform.

The EVM running on each node in the Ethereum Blockchain can be thought of as a global 256-bit distributed, decentralized computer containing millions of executable objects, each with its own permanent data store, and where all transactions execute atop local nodes in relative synchrony. It handles smart contract deployment and execution. Specifically, a smart contract runs on the EVM, which in turn runs on each node in the Blockchain network. While powerful in its own right, the EVM operates at the level of bytecodes (like the JVM or Java Virtual Machine) and is not convenient to directly program against. Thus, smart contracts are typically written in a high-level language, such as Solidity. However, to execute, it must be compiled down to the low-level EVM bytecode [13]. Specifically, Ethereum provides the developers with a runtime that support several high-level languages to write smart contracts encoding operational logic for digital/logical representation of assets expressed as data or program variables in code) and their subsequent transfer to other entities upon code execution.

A smart contract can only execute if it is invoked by a transaction from a user account. In other words, a contract can invoke another contract creating multiple levels of the invocation call chain. However, the first contract in such a contract call chain always belongs to a transaction initiated by a user account. Note that the Ethereum Blockchain transactions are atomic (much like their database counterparts), i.e., they execute in their entirety, with any changes in the global state (contracts, accounts, etc.) recorded if and only if all execution terminates successfully, regardless of the depth of the contract call chain. A successful termination means that the smart contract execution proceeded without any exception. If execution fails, all of its state changes are reverted. A key difference between database transaction and the Ethereum Blockchain transaction is that failed transactions leave a trace. In other words, they are still recorded as having been attempted, and the gas consumed for any contract computation is deducted from the originating account.

A smart contract is immutable, i.e., it cannot be modified upon deployment. The exact process of deployment occurs using a special contract creation transaction and is described later in § 1.2.1. However, a smart contract can be deleted (only if it has been designed to do so), i.e., removing the code and internal state (storage) from its address, leaving a blank account. Any transaction sent to this deleted contract's address will not result in any code

execution since there is no executable code at that address. Note that deleting a contract does not remove the transaction history of the contract since the Blockchain itself is immutable. To delete a smart contract, the EVM provides a special bytecode instruction, which if correctly handled within the smart contract code upon execution issues a gas refund. This refund is akin to release of network client resources upon the deletion of stored state. If the smart contract does not handle the concerned opcode, the resources cannot be recovered.

How does the EVM work?

The EVM operates upon a 256-bit addressable stack-based architecture, and has three key addressable data components:

- an immutable program code ROM, loaded with the bytecode of the smart contract to be executed,
- a volatile memory with every location explicitly initialized to zero,
- permanent storage that is also zero-initialized.

Unlike other traditional VMs (or hypervisors), which use software to abstract away hardware details, the EVM is much closer to the JVM in its operation and is primarily used as a computation engine. Like the JVM, the EVM provides the developers with a runtime (that is agnostic of the underlying OS and/or hardware) for executing smart contracts by enabling compilation of high-level programming languages into its own bytecode instruction set. Additionally, the EVM, which is single-threaded, has no scheduling capability of its own. Thus, contracts never execute in parallel or in the background. Ethereum clients run through verified block transactions to determine which smart contracts need executing and in what order.

When creating and deploying a new contract on the Ethereum platform, a special transaction is needed that has its `to` field (see Fig. 1.5) set to the special $0x0$ address and its `data` field set to the smart contract's initiation code. When such a contract creation transaction is processed, the code for the new contract account is not the code in the `data` field of the transaction. Instead, an EVM is instantiated with the code in the `data` field of the transaction loaded into its program code ROM, and then the output of the execution of that deployment code is taken as the code for the new smart contract. This is so that new contracts can be programmatically initialized using

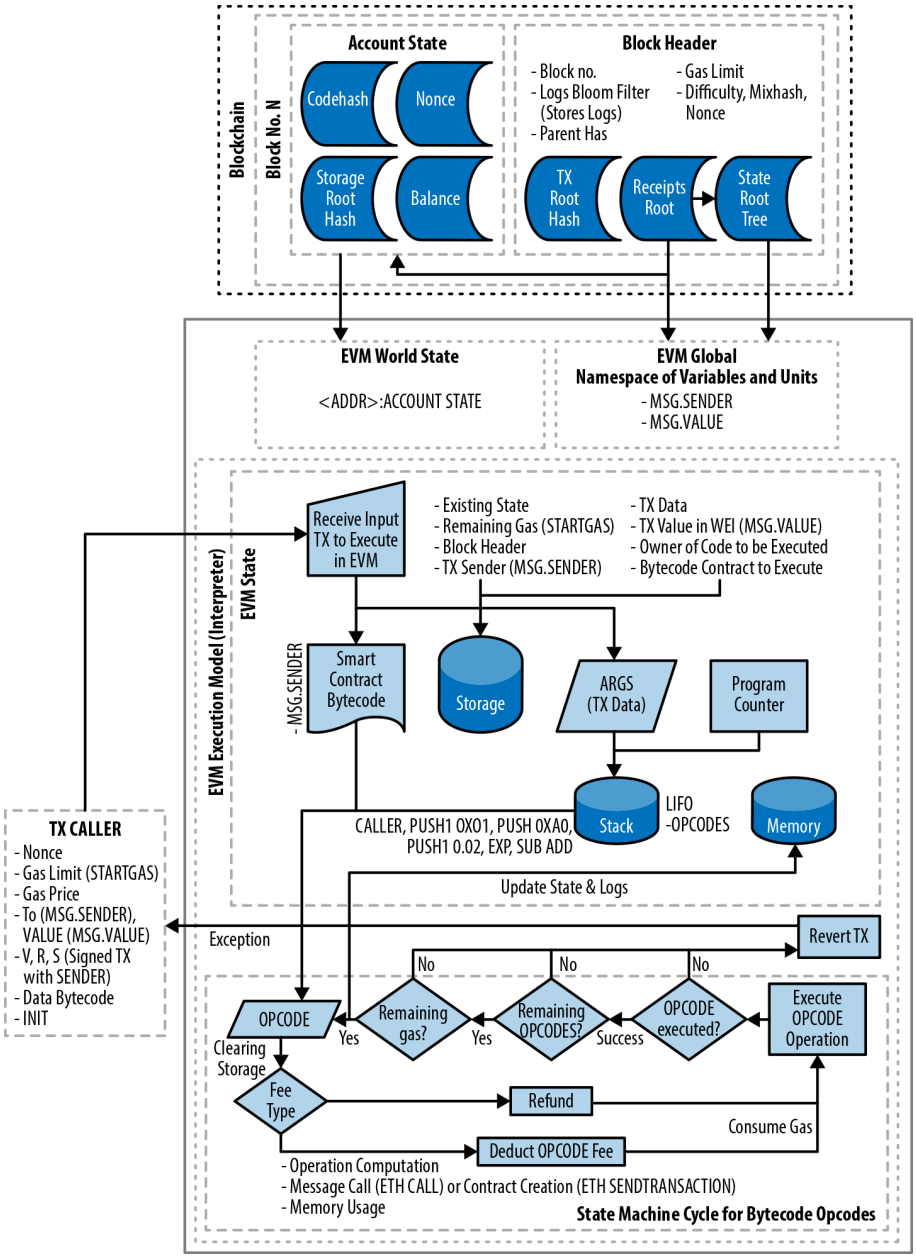

Figure 1.5: **Ethereum Virtual Machine (EVM) Architecture and Execution Context** [28].

the Ethereum world state at the time of deployment, setting values in the contract's storage and even sending ether or creating further new contracts.

When a transaction results in smart contract execution, an EVM is instantiated with all the state corresponding to the current block being created and the specific transaction being processed. Specifically, the smart contract's code is loaded into the EVM's ROM and the stack program counter is set to zero. Additionally, the storage is loaded from the contract account's storage, the memory is initialized, and the corresponding block and environment variables are set. Finally, the gas for the transaction is set to the amount of gas paid for by the sender. As the smart contract code execution proceeds, this gas is reduced in accordance with the cost of the bytecode being executed currently. If at any state the gas drops to zero, the EVM first halts the smart contract execution by throwing an "Out of Gas" (OOG) exception, and subsequently aborts the transaction.

Note that since all computation is bounded, the EVM is a "quasi" Turing-complete state machine. In other words, since a smart contract execution is subject to a finite number of computational steps limited by the amount of gas available, there is not execution where the EVM runs forever, thereby avoiding the "halting problem" by design.

Each smart contract executes within a sandboxed copy of the Ethereum world state. No changes to the Ethereum state are applied until the transaction commits successfully, except for any updates to the sender's `nonce` and the `gas` balance to reflect the code execution. If a transaction aborts, the EVM discards all changes to the Ethereum world state made within the sandbox. however, if the transaction completes successfully, then the real-world state is updated to match the sandbox's version, including any changes to the contract's storage data, any new contracts created, and any ether balance transfers that were initiated.

Note that because a smart contract can itself initiate new transactions, code execution is a recursive process. A contract can call other contracts, with each call resulting in another EVM being instantiated around the new target of the call. Each instantiation has its sandboxed world state initialized from the sandbox of the EVM at the level above. Each instantiation is also given a specified amount of gas for computation (not exceeding the amount of gas remaining in the level above, of course). In case a nested EVM sandbox encounters an exception and subsequently halts the execution, the sandbox state is discarded, and execution returns to the EVM at the level above.

1.2.2 Solidity and Smart Contracts

While imperative (or procedural) languages are more commonly used by the developers due to their ease of use, they make it difficult to reason about the program's state change. In contrast, declarative (or functional) languages make it easier to understand how a program will behave since such programs have no side-effects and any part can be understood in isolation. In the context of smart contracts, it is extremely critical to write correct and fair smart contracts, since bugs can cause losses worth millions of dollars [33,36]. As a result, it is important to clearly reason about the expected behavior of the program, and write smart contracts without any unintended effects. However, in spite of the obvious advantages of using declarative languages for encoding smart contracts, programmers use imperative languages for smart contract development owing to their ease of use.

In the Ethereum ecosystem, Solidity is the most popular language. Solidity follows a procedural programming paradigm with a syntax similar to JavaScript and C++. It is statically typed, supports inheritance, libraries, and complex user-defined types, among other features. In order to make Solidity amenable for smart contract development, i.e., generate deterministic outputs and still prevent bugs, the developers of Solidity restricted some of its capabilities, like limitations on the use of `Arrays` and `Strings`. However, it is the "quasi" Turing-complete EVM with its finite resource allocation per contract and pre-defined cost per bytecode computation model which ensures that the smart contract code execution is always deterministic and bounded, i.e., the same algorithms with the same inputs will always yield the same results.

Solidity Basics

Solidity has functionalities similar to C and JavaScript. It supports state variables, general data types, along with the variable data types. It also supports the generic value types, namely:

- **Boolean**: Returns value as either true or false. The logical operators returning Boolean data types are—`!` Logical negation, `&&` logical conjunction, `||` logical disjunction, `==` equality, and `!=` inequality.
- **Integer**: Solidity supports `int/unit` for both signed and unsigned integers respectively. These storage allocations can be of various sizes,

such as `uint8` and `uint256` can be used to allocate a storage size of 8 bits to 256 bits respectively. By default, the allocation is 256 bits, i.e., uint and int can be used in place of uint256 and int256. Standard arithmetic, bit operators, and comparison operators are compatible with these integer data types.

- **Address**: An `address` can hold a 20 byte value that is equivalent to the size of an Ethereum address.

- **String**: Solidity has limited support for `Strings`, which can be represented using either single or double quotes. Unlike C, Solidity `string` literals do imply trailing value zeroes. For instance, "bar" will represent a three byte element instead of four. Similarly, integer literals are convertible inherently using the corresponding fit, i.e., `byte` or `string`.

- **Modifier**: Modifiers are used to ensure the coherence of the conditions defined before executing the code.

Solidity also provides basic arrays, enums, operators, and hash values to create a data structure known as "mappings". Arrays can be both statically and dynamically allocated.. For example, `uint[][6]` initializes a dynamic array with 6 contiguous memory allocations. Similarly, a two-dimensional array can be initialized as `arr[2][4]`, where the two indices point towards the dimensions of the matrix.

Example: A Simple Voting Smart Contract

We use Solidity for writing our simple voting smart contract (see Fig. 1.6). It consists of a `contract` declaration with two state variables: one to store an array of candidates, and another to maintain a count of their corresponding votes, a constructor to initialize the array of candidates, and two methods, one to return the total votes a candidate has received and another method to increment vote count for a candidate. Note that the constructor is invoked only once when we deploy the smart contract on the Blockchain.

Line `1` specifies the version of Solidity compiler this code is compatible with. Line `2` declares a `contract` with two state variables—`votesReceived` and `candidateList`—at lines `3` and `4` respectively. `votesReceived` is a mapping and is equivalent to an associative array or hash. The key to the mapping is a candidate's name stored as type `bytes32` and its value is an

```
(1)   pragma solidity ^0.4.25;

(2)   contract Voting {

(3)     mapping (bytes32 => uint8) public votesReceived;
(4)     bytes32[] public candidateList;

(5)     constructor (bytes32[] candidateNames) public {
(6)       candidateList = candidateNames;
(7)     }

(8)     function totalVotesFor(bytes32 candidate) view public returns (uint8) {
(9)       require(validCandidate(candidate));
(10)      return votesReceived[candidate];
(11)    }

(12)    function voteForCandidate(bytes32 candidate) public {
(13)      require(validCandidate(candidate));
(14)      votesReceived[candidate] += 1;
(15)    }

(16)    function validCandidate(bytes32 candidate) view public returns (bool) {
(17)      for(uint i = 0; i < candidateList.length; i++) {
(18)        if (candidateList[i] == candidate) {
(19)          return true;
(20)        }
(21)      }
(22)      return false;
(23)    }

(24)  }
```

Figure 1.6: **A simple voting smart contract (adapted from [15]).**

unsigned integer to store the vote count. `candidateList` is an array of `bytes32` and stores the list of candidates.

Line `5` declares the contract's constructor that is invoked once when the contract is deployed on to the Blockchain. Specifically, the constructor's execution instantiates an initializes an array of candidates who will be contesting in the election. Line `8` describes the function `totalVotesFor` that returns the total votes a candidate has received so far. Line `12` describes the `voteForCandidate` function that increments the vote count for the specified candidate, which is equivalent to casting a vote. Finally, lines `16` to `23` list the `validCandidate` function that is responsible for sanity checking whether the candidate is a legitimate candidate or not. It compares the given candidate with `candidateList`, which stores the list of all candidates.

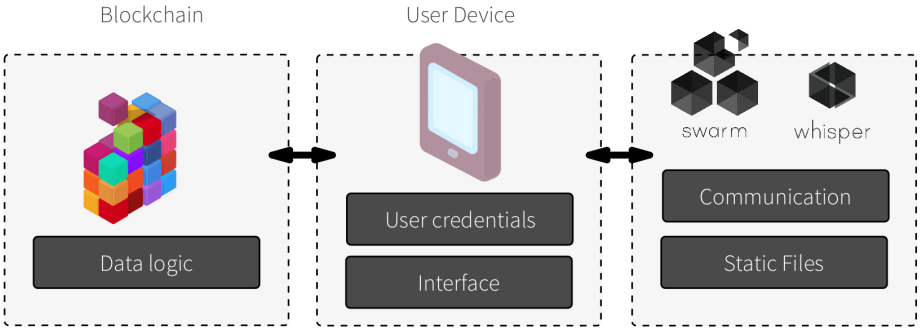

Figure 1.7: **DApp architecture using Blockchain and P2P technologies.**

1.2.3 Decentralized Applications

A smart contract is a unit of functionality that executes atop the EVM. A decentralized application (or DApp as it is commonly known) represents a broader perspective than smart contracts and typically describes a JavaScript/HTML-based web- or smartphone-accessible GUI front-end application built atop open, decentralized, peer-to-peer infrastructure services. In the context of Ethereum-powered DApps, they leverage `NodeJS/web3js` powered front-end and the EVM as its back-end. Unless it is a very simple DApp, its back-end functionality will rely on several smart contracts, and a decentralized (P2P) messaging protocol and storage platform (see Fig. 1.7). A good DApp utilizing Blockchain technology allows actions similar to a centralized application (like transferring money) but without the need for a trusted third-party.

Decentralized v/s Distributed Applications

Decentralized applications (or DApps) are often confused with distributed applications. Fig. 1.8 highlights the architectural differences between centralized, distributed and decentralized applications. An application is distributed if it runs atop multiple servers within a network. The simplest example of a distributed application is a web application, which is typically distributed over a web server, an application server and a database server.

A decentralized application is replicated in its entirety over each node of

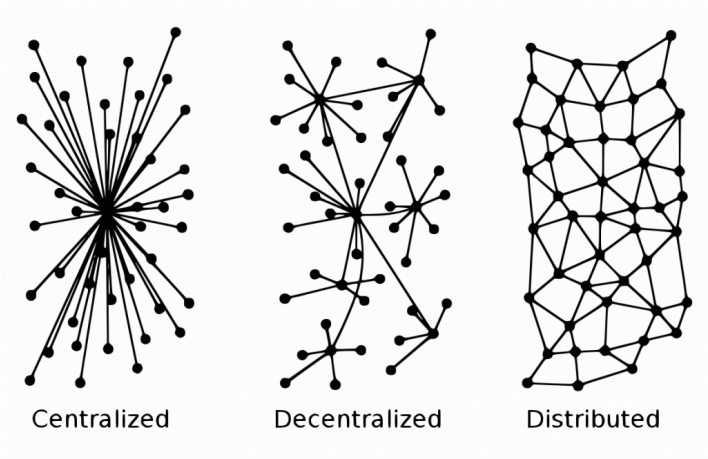

Figure 1.8: **Centralized, distributed and decentralized architectures.**

a wide network, with each node (in principle) owned by a different entity. The higher the number of entities owning nodes of the network, the more trustful the network in its entirety is. Conversely, networks owned only by a few owners cannot be considered trustful because they do not truly decentralize the processing. A centralized application is generally distributed, but decentralized applications can also be distributed over multiple servers within each logical node.

Trust and Security in DApps

A decentralized application makes trust and security breaches pointless by replicating its execution over a network of servers, where each server may be in principle owned by a different entity. Consider a voting DApp powered by the simple voting smart contract listed in Fig. 1.6. If it were a centralized application, i.e., votes were processed and verified by one single server, then both trust and security are a concern. However, with vote verification done independently across many servers owned by different parties, stored across a Blockchain, with ledgers local to the nodes, both trust and security are preserved.

- **Trust**: If a malicious participants altered a vote and propagate it through the network, other participants would detect it as a tampered vote during their validation and subsequently reject it. They would

not store it into their local ledgers and thus prevent the tampered vote from being registered on the Blockchain, thereby rendering the malicious modification ineffective.

- **Security**: If an attacker were to modify the votes on one server in the network, the alteration would be spotted and rejected by other participants, as seen earlier. A truly successful modification attempt requires the support of at least 51% of the nodes in the network to achieve consensus on the modified state of the votes. As is obvious, manipulating a large fraction of the decentralized network of independently managed nodes is an incredibly challenging task, thereby ensuring security.

Life Cycle of a Transaction in a Voting DApp

A voting DApp differs from the simple voting smart contract in the way that it also has a JavaScript/HTML-based front-end along with additional P2P-based messaging and storage capabilities. We now discuss the full life cycle of a voting transaction in such a DApp.

1. The user selects one of the possible voting options from a dropdown list box on the web client and then clicks the Vote button.

2. The click event is handled by a JavaScript function that captures the voting selection, and subsequently, through various third-party JavaScript library functions set up communication with an Ethereum configured node, and invokes the `voteForCandidate` function in the voting smart contract. The invocation of the `voteForCandidate` function generates a transaction message, which is digitally signed by the user's private key.

3. The contacted local Ethereum node broadcasts the transaction message to other peers and miners in the network for validation.

4. Based on the transaction metadata, a miner will determine whether it wants to generate a block corresponding to this transaction or not. If yes, then the miner executes the code of the function and create a new block. However, the miner will be able to propagate this block in the network only if beats other miners in solving a proof-of-work puzzle. If it does win the race, the miner relays the new block to all its peer nodes.

5. Eventually, the local Ethereum nodes receive this new incoming block and synchronize its local copy of the Blockchain. Upon receiving the new block, each local node executes all the transactions in the block. One of these is the voting transaction, which upon successful completion has been programmed to raise a callback event. This vote confirmation event is published to all the clients subscribed to it, including the DApp web UI.

6. The JavaScript code present on the voting web client itself contains a callback function registered against the vote confirmation event, which then gets triggered. Finally, the callback function shows a vote confirmation notification on the voter's screen.

1.3 Hyperledger Fabric

Hyperledger Fabric [23] (or Fabric as it is commonly known) is an open-source permissioned Blockchain platform for enterprise settings. It has several key features that differentiate it from permissionless Blockchain platforms such as Ethereum. Let us know briefly discuss some of the features concerning smart contract development, deployment, and operations in Fabric.

- Fabric's permissioned Blockchain model (as opposed to Ethereum's permissionless model) makes it an attractive option for several industry use cases including banking, finance, insurance, healthcare, human resources, supply chain, and even governance. Furthermore, since Fabric has a highly modular and configurable architecture, it enables innovation, versatility, and optimization. In contrast, Ethereum offers a very rigid and monolithic architecture, which may not be suitable for a large class of applications.

- Unlike Ethereum, which requires developers to learn Solidity—a new constrained domain specific language (DSL) for writing smart contract's, Fabric supports smart contracts authored in general-purpose programming languages such as Java, Go and Node.js. This enables enterprise developers to quickly transition to developing smart contracts from writing enterprise applications, without any additional training to learn a new language or DSL.

- As mentioned earlier, Hyperledger Fabric is a permissioned Blockchain platform, meaning that, unlike with Ethereum—a public permissionless network where participants can stay anonymous (and therefore untrustworthy) and still carry out transactions, the identity of the participants in Fabric is known to each other. Thus, while the participants may not fully trust one another, since they may be competitors, they can still transact under a legal agreement or framework for handling disputes.

- Fabric supports pluggable consensus protocols, which enable it to be customized to specific use cases and trust models. For instance, when deployed within a single enterprise, or operated by a trusted authority, fully Byzantine fault-tolerant consensus might be unnecessary and significantly impact performance and throughput. In such situations, a crash fault-tolerant (CFT) consensus protocol might be adequate. In contrast, Ethereum provides no flexibility and consensus is baked into the operating protocol itself.

- Lastly, Fabric, unlike Ethereum, does not require a native cryptocurrency to execute smart contracts, which significantly mitigates certain attack vectors. Moreover, Fabric does not require any cryptographic mining operations, which means no additional deployment overhead.

1.3.1 Smart Contracts in Hyperledger Fabric

A smart contract essentially implements the business logic running on the Blockchain. In Fabric terms, it is called as chaincode. Chaincodes in Fabric can be written in Go, Java, and JavaScript (node.js). Each chaincode runs in a secured Docker container isolated from other peer processes, and initializes and manages the ledger state through transactions submitted by applications.

A chaincode typically handles business logic that members of the network have agreed to. The state created by a chaincode is scoped exclusively to that chaincode and cannot be accessed directly by another chaincode. However, with appropriate permissions, a chaincode in the same network can invoke another chaincode to access its state.

Hyperledger Fabric considers two different types of chaincodes:

- System chaincode typically handles system-related transactions such as lifecycle management and policy configuration.

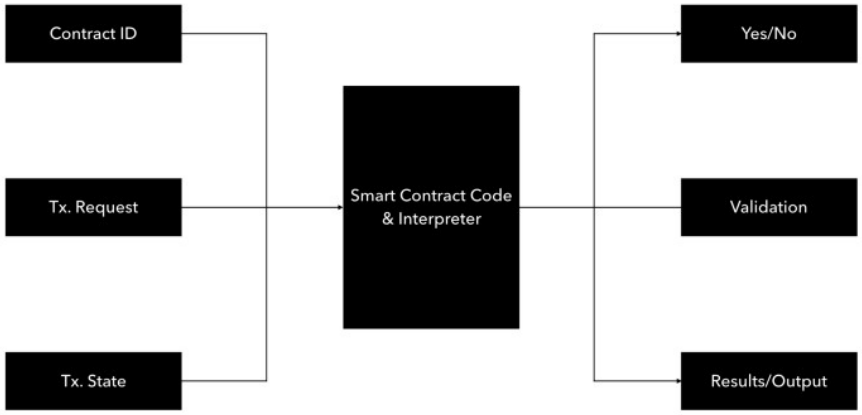

Figure 1.9: **A Hyperledger Fabric smart contract execution.**

- Application chaincode manages application states on the ledger, including digital assets or arbitrary data records.

A chaincode execution begins when a node in the network submits an instantiation transaction to the network. If the transaction is approved, the chaincode enters an active state where it can receive transactions from users via client applications. Any chaincode transactions that are validated are appended to the shared ledger, and can then modify the world state accordingly. Note that while Ethereum prevents any smart contract to be upgraded after deployment, Hyperledger Fabric enables to upgrade a chaincode, any time after it has been instantiated, by submitting an upgrade transaction.

Processing Smart Contracts

The smart contract layer in Hyperledger Fabric is responsible for executing the smart contract and updating the resultant state of the Blockchain. Fig. 1.9 highlights this processing. Specifically, the inputs include the contract identifier, the transaction request, any dependencies that may exist, and the current Blockchain state. The contract interpreter also has access to the current ledger state and the smart contract code. When the interpreter

receives a request, it immediately checks for any invalid requests. Subsequently, it generates outputs, which may include new ledger states, for all valid requests.

The smart contract layer validates each request by ensuring that it conforms to policy and the contract specified for the transaction. Invalid requests are rejected and may be dropped from inclusion within a block. Validation failures can be due to either syntax or logic errors. Syntax errors, like as invalid inputs, unverifiable signature, and any repeated transaction (due to errors or replay attacks), cause the request to be dropped. Logic errors are more complex, and the decision whether to continue processing is often policy-driven. The result of validation is a transaction that captures the transition to the new state.

When the processing is complete, the interpreter packages the new state (in the form of change sets) and an attestation of correctness, which is sent to the consensus service for final commitment to the Blockchain. In certain situations, the interpreter may also package any additional ordering hints for the consensus service.

This separation between contract execution and consensus complicates the commitment to the Blockchain. For example, two requests executed in parallel can result in inconsistencies in the contract state unless they are ordered in a specific way. In general, the consensus layer handles ordering. However, the smart contract can provide hints about properties like ordering. Processing a request can also generate side effects, such as event notifications, requests to other smart contract, modifications to global resources, some of which may not be captured in the change sets.

Fig. 1.10 shows how smart contracts interact with other Blockchain layers in Hyperledger Fabric. The smart contract layer works very closely with the consensus layer. Specifically, it receives a proposal from the consensus layer specifying which contract to execute, the details of the transaction including the identity and credentials of the entity asking to execute the contract, and any transaction dependencies.

The smart contract layer leverages the current state of the ledger and input from the consensus layer to validate the transaction. While processing the transaction, the smart contract layer uses the identity services to authenticate and authorize the entity asking to execute the smart contract. This ensures that the identity of the entity is known and that the entity has the appropriate access to execute the smart contract.

After processing the transaction, the smart contract layer returns whether

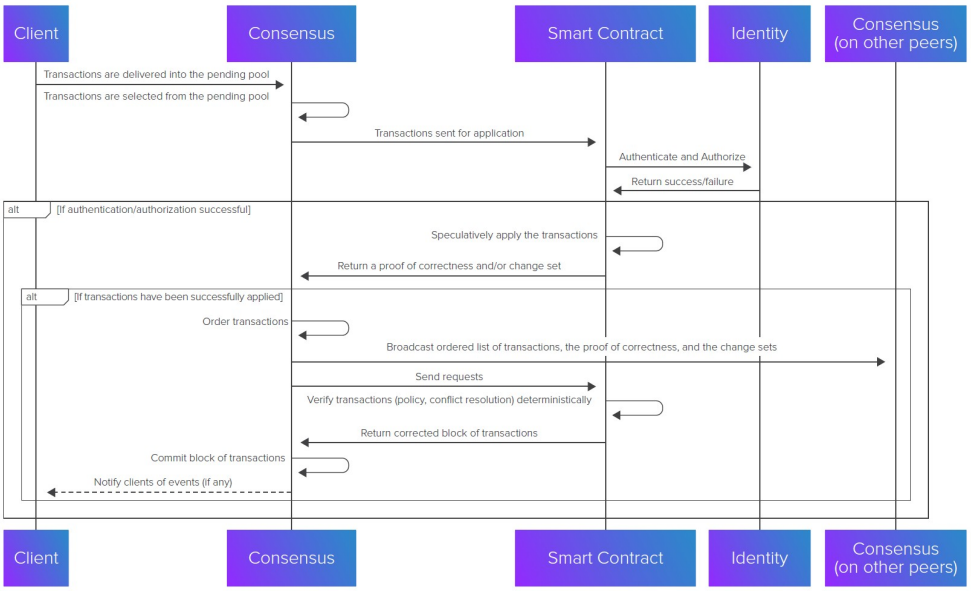

Figure 1.10: **Smart contract interaction with other layers in Hyperledger Fabric [22].**

the transaction was accepted or rejected. If the transaction was accepted, the smart contract layer also returns an attestation of correctness, a state delta, and any optional ordering hints needed to ensure the transaction dependencies are taken into account. The state delta includes the change sets and any side effects that should take place when the transaction is successfully committed by the peers.

Order-Execute: Existing Pipeline

Most existing Blockchain platforms, ranging from public/permissionless platforms such as Ethereum (with PoW-based consensus) to permissioned platforms such as Tendermint, Chain, and Quorum, follow an **order-execute** architecture in which the consensus protocol:

- validates and **orders** transactions then propagates them to all peer nodes,
- each peer then **executes** the transactions sequentially.

However, there are a couple of issues with the order-execute approach. First, the execution of the smart contract must be deterministic. Otherwise, consensus might never be reached. To address the non-determinism issue, platforms such as Ethereum require smart contracts to be written in a non-standard domain-specific language (such as Solidity) so that the non-deterministic operations can be eliminated at the language level itself. Note that Solidity has no non-determinism inducing function calls. However, this use of a DSL hinders wide-spread adoption because it requires developers to learn a new language and may lead to programming errors. Second, since all transactions are executed sequentially by all nodes, the order-execute architecture limits the performance and scale of the Blockchain platforms. Further, since smart contract code executes on every node in the system, it requires complex measures to protect the overall system from potentially malicious contracts in order to ensure the resiliency of the overall system.

Execute-Order-Validate: A Different Design

To overcome the problems of the order-execute architecture, Hyperledger Fabric introduces an **execute-order-validate** pipeline. It addresses the resiliency, flexibility, scalability, performance and confidentiality challenges faced by the **order-execute** model by separating the transaction flow into three steps:

- speculatively **execute** multiple transactions in parallel across different nodes and check their correctness, i.e., endorse the transactions,
- **order** the transactions via a (pluggable) consensus protocol, and
- **validate** the transactions against an application-specific endorsement policy before committing the transaction change sets to the Blockchain.

This three-part smart contract execution pipeline in Hyperledger Fabric ensures that transactions are executed before reaching final agreement on their order. An application-specific endorsement policy specifies which peer nodes, or how many of them, need to vouch for the correct execution of a given smart contract. Thus, each transaction need only be executed (endorsed) by the subset of the peer nodes necessary to satisfy the transaction's endorsement policy. This also allows for parallel execution of smart contracts, thereby increasing the overall parallelism, performance, and scale of

```
(1)   // ...  import headers

(2)   type SimpleChaincode struct {
(3)   }

(4)   func main() {
(5)      err := shim.Start(new(SimpleChaincode))
(6)      if err != nil {
(7)         fmt.Printf("Error starting Simple chaincode:  %s", err)
(8)      }
(9)   }

(10)  func (t *SimpleChaincode) Init(stub shim.ChaincodeStubInterface,
                           function string, args []string) ([]byte, error) {
(11)     if len(args) != 1 {
(12)        return nil, errors.New("Incorrect number of arguments.  Expecting 1")
(13)     }
(14)     err := stub.PutState("Hello World!", []byte(args[0]))
(15)     if err != nil {
(16)        return nil, err
(17)     }
(18)     return nil, nil
(19)  }

(20)  func (t *SimpleChaincode) Invoke(stub shim.ChaincodeStubInterface,
                           function string, args []string) ([]byte, error) {
(21)     fmt.Println("invoke is running " + function)
(22)     // Handle different functions
(23)     if function == "init" {
(24)        return t.Init(stub, "init", args)
(25)     } else if function == "write" {
(26)        // implements logic to write to the blockchain using PutState
(27)        return t.write(stub, args)
(28)     }
(29)     fmt.Println("invoke did not find func:  " + function)
(30)     return nil, errors.New("Received unknown function invocation")
(31)  }

(32)  func (t *SimpleChaincode) Query(stub shim.ChaincodeStubInterface,
                           function string, args []string) ([]byte, error) {
(33)     fmt.Println("query is running " + function)
(34)     // Handle different functions
(35)     if function == "read" { //read a variable
(36)        // implements logic to read from the blockchain using GetState
(37)        return t.read(stub, args)
(38)     }
(39)     fmt.Println("query did not find func:  " + function)
(40)     return nil, errors.New("Received unknown function query")
(41)  }
```

Figure 1.11: **An example Hyperledger Fabric chaincode in Golang.**

the system. It also eliminates any non-determinism, as inconsistent results are filtered out before ordering at the nodes, and thus also allows the use of standard programming languages to smart contract development.

An example Hyperledger Fabric chaincode

Fig. 1.11 lists an example Fabric chaincode in Golang, and highlights the three important components—`init`, `invoke` and `query`. `init` is called during chaincode instantiation to initialize or bootstrap the Blockchain ledger with initial data. In lines 10—19, Init function takes a `ChaincodeStubInterface` as a parameter, which is passed when invoking via the CLI or the ClientSDK. The function initializes the ledger using the `PutState` API and then returns. Lines 20—31 implement the `invoke` method. Specifically, `invoke` can modify the state of the variables in ledger using `PutState`, and each `invoke` transaction will be added to the "block" in the ledger. Finally, lines 32—41 implement the `query` interface, which reads the current state (using `GetState`) and sends it back to the client. This transaction is not saved in the Blockchain.

1.4 Summary

A smart contract is essentially business logic running on a Blockchain. It can simply update an account balance, or be more complex to manage and validate the entire supply chain for a particular product. While public adoption of Blockchain has been spearheaded by permissionless platforms like Ethereum (and Bitcoin), adopting Blockchain in an enterprise will require a balancing act. Organizations will not only have to run, but also manage, and maintain their existing infrastructure. Hyperledger Fabric is one promising Blockchain platform for the permissioned enterprise scenarios, which has gained wide prominence with both its modular architecture and ease of developing smart contracts in general purpose languages.

Bibliography

[1] A Simple Model for Smart Contracts. https://gendal.me/2015/02/10/a-simple-model-for-smart-contracts/.

[2] Barclays used blockchain tech to trade derivatives. https://www.cnbc.com/2016/04/19/barclays-used-blockchain-tech-to-trade-derivatives.html.

[3] Bext360 and Coda Coffee Release The Worlds First Blockchain-traced Coffee from Bean to Cup. https://globenewswire.com/news-release/2018/04/16/1472230/0/en/bext360-and-Coda-Coffee-Release-The-World-s-First-Blockchain-traced-Coffee-from-Bean-to-Cup.html.

[4] Blockchain Identity. https://www.ibm.com/blogs/blockchain/category/blockchain-identity/.

[5] Blockchain Used to Sell Real Estate for the First Time in the EU. https://www.prnewswire.com/news-releases/blockchain-used-to-sell-real-estate-for-the-first-time-in-the-eu-808847530.html.

[6] Comprehensive List of Banks using Blockchain Technology. https://hackernoon.com/comprehensive-list-of-banks-using-blockchain-technology-97c08fa88385.

[7] Decentralized digital identities and blockchain: The future as we see it. https://www.microsoft.com/en-us/microsoft-365/blog/2018/02/12/decentralized-digital-identities-and-blockchain-the-future-as-we-see-it/.

[8] Did Russia Affect the 2016 Election? Its Now Undeniable. https://www.wired.com/story/did-russia-affect-the-2016-election-its-now-undeniable/.

[9] Do You Know Your Diamond? https://diamonds.everledger.io/.

[10] Dubai Land Department Blockchain Project. https://www.dubailand.gov.ae/English/Pages/Blockchain.aspx.

[11] e-estonia. https://e-estonia.com/.

[12] Ethereum Blockchain Application Platform. https://www.ethereum.org/.

[13] Ethereum VM (EVM) Opcodes and Instruction Reference. https://github.com/trailofbits/evm-opcodes.

[14] Every week more Governments are announcing Blockchain adoption. https://www.globalbankingandfinance.com/every-week-more-governments-are-announcing-blockchain-adoption/.

[15] Full Stack Hello World Voting Ethereum Dapp Tutorial –Part 1. https://medium.com/@mvmurthy/full-stack-hello-world-voting-ethereum-dapp-tutorial-part-1-40d2d0d807c2.

[16] Getting Smart About Smart Contracts. https://www2.deloitte.com/us/en/pages/finance/articles/cfo-insights-getting-smart-contracts.html.

[17] Hackers attempt cyber attacks on state voting system. https://www.wmcactionnews5.com/2018/09/27/hackers-attempt-cyber-attacks-state-voting-system/.

[18] Hello Big Brother: EU wants to manage your digital identity on the blockchain. https://thenextweb.com/hardfork/2018/12/07/eu-digital-identity-governmental-use-blockchain/.

[19] How Blockchain Can Solve Identity Management Problems. https://www.forbes.com/sites/forbestechcouncil/2018/07/27/how-blockchain-can-solve-identity-management-problems/#14f41c4813f5.

[20] How Ethereum and Smart Contracts Work. https://vas3k.com/blog/ethereum/.

[21] How smart contracts automate digital business. https://usblogs.pwc.com/emerging-technology/how-smart-contracts-automate-digital-business/.

[22] Hyperledger Architecture, Volume II. https://www.hyperledger.org/wp-content/uploads/2018/04/Hyperledger_Arch_WG_Paper_2_SmartContracts.pdf.

[23] Hyperledger Fabric Blockchain Platform. https://www.hyperledger.org/projects/fabric.

[24] My Health My Data. http://www.myhealthmydata.eu/.

[25] Newly Formed US Healthcare Alliance to Trial Blockchain for Improved Data Quality. https://cointelegraph.com/news/newly-formed-us-healthcare-alliance-to-trial-blockchain-for-improved-data-quality.

[26] Nick Szabo. https://en.wikipedia.org/wiki/Nick_Szabo.

[27] Smart Dubai and TRA Launch National Digital Identity. https://smartdubai.ae/newsroom/news-details/2018/10/17/smart-dubai-and-tra-launch-national-digital-identity#listingPage=1&tab=event.

[28] The Ethereum Virtual Machine. https://fullstacks.org/materials/ethereumbook/14_evm.html.

[29] Using Blockchain to Unblock the Supply Chain. https://www.bext360.com/using-blockchain-to-unblock-the-supply-chain/.

[30] Worlds First Blockchain Coffee Project. https://moyeecoffee.ie/blogs/moyee/world-s-first-blockchain-coffee-project.

[31] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet

Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 30:1–30:15, New York, NY, USA, 2018. ACM.

[32] Marcus Brandenburger, Christian Cachin, Rüdiger Kapitza, and Alessandro Sorniotti. Blockchain and trusted computing: Problems, pitfalls, and a solution for hyperledger fabric. *CoRR*, abs/1805.08541, 2018.

[33] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. ZEUS: analyzing safety of smart contracts. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, 2018.

[34] Sukrit Kalra, Rishabh Sanghi, and Mohan Dhawan. Blockchain-based real-time cheat prevention and robustness for multi-player online games. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '18, pages 178–190, New York, NY, USA, 2018. ACM.

[35] John Kelsey, Andrew Regenscheid, Tal Moran, and David Chaum. Towards trustworthy elections. chapter Attacking Paper-based e2e Voting Systems, pages 370–387. Springer-Verlag, Berlin, Heidelberg, 2010.

[36] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 254–269, New York, NY, USA, 2016. ACM.

[37] Nick Szabo. Smart contracts: Building blocks for digital markets. 1996.