ANNAMACHARYA UNIVERSITY, RAJAMPET

(ESTD UNDER AP PRIVATE UNIVERSITIES (ESTABLISHMENT AND REGULATION) ACT, 2016
RAJAMPET, Kadapa District, AP, INDIA

Course : **Blockchain Technologies**

Course Code : **24FMCA3CT**

Branch : **MCA**

Prepared by : **J.HARIKRISHNA**

Designation : **Assistant Professor**

Department : **MCA**

ANNAMACHARYA UNIVERSITY, RAJAMPET

(ESTD UNDER AP PRIVATE UNIVERSITIES (ESTABLISHMENT AND REGULATION) ACT, 2016
RAJAMPET, Kadapa District, AP, INDIA

Title of the Course : **Blockchain Technologies**
Category : **PE**
Course Code : **24FMCA3CT**
Branch : **MCA**
Semester : **III Semester**

Lecture Hours	Tutorial Hours	Practice Hours	Credits
3	0	0	3

COURSE OBJECTIVES:

- Understand how block chain systems (mainly Bitcoin and Ethereum) work.
- Introduce key vocabulary and concepts related to block chain and Bitcoin in business situations.
- Discuss the current state of the Block chain landscape.
- Design, build, and deploy smart contracts and distributed applications.
- Integrate ideas from blockchain technology into their own projects.

COURSE OUTCOMES:

The Student will be able to

1. Explain advantages, limitations and decentralize currency using block chain technology.
2. Apply Cryptography for Block chain technology
3. Analyze Bitcoin crypto currency and alternate to bitcoin technology
4. Describe the smart contracts to add and investigate the challenges and securiry issues of ethereum
5. Analyze the Hyperledger project and open source ledger framework

UNIT I BLOCKCHAIN & DECENTRALIZATION 10 Hrs

Block Chain : Distributed systems, the history of block chain, introduction to block chain, CAP theorem and block chain, Benefits and limitations of block chain.

Decentralization: Decentralization using block chain, Methods of decentralization, Routes to decentralization, Block chain and full ecosystem decentralization, Smart contract, Decentralized organizations, Decentralized autonomous organizations, Decentralized autonomous corporations, Decentralized autonomous societies, Decentralized applications, Platforms for decentralization.

UNIT II CRYPTOGRAPHY AND TECHNICAL FOUNDATIONS 9Hrs

Introduction, Cryptography, Confidentiality, Integrity, Authentication, Cryptographic primitives, Public and private keys, Financial markets and trading.

UNIT III BITCOIN & ALTERNATIVE COINS 9 Hrs

Bitcoin: Bitcoin Transactions, Blockchain, Bitcoin payments, Bitcoin programming and the command- line interface, Bitcoin improvement proposals (BIPs).

Alternative Coins: Theoretical foundations, Bitcoin limitations, Name coin, Litecoin

UNIT IV SMART CONTRACTS & ETHEREUM 8 Hrs

Smart Contracts: History, Definition, Ricardian contracts

Ethereum: Introduction, Ethereum blockchain, Elements of the Ethereum blockchain, Precompiled contracts, Accounts, code Block.

UNIY V Hyper LEDGER 10 Hrs

Hyper Ledger: Projects, Hyperledger as a protocol, Fabric, Hyper ledger Fabric, Sawtoothlake, Corda.

TEXTBOOK:

1. Imran Bashir, “Mastering Block chain” Packt Publishing Ltd, March 2017.

REFERENCE BOOKS:

1. Josh Thompson, “Block chain: The Block chain for Beginnings, Guild to Block chain Technology and Block chain Programming”, Create Space Independent Publishing Platform,2017.
2. Melanie swan, “Blok chain blueprint for a new economy”, O"REILLY, 1 st Edition, 2015.
3. Paul Vigna & Michael J. Casey, The age of crypto currency, 1 st Edition 2015.

CO-PO MAPPING:

Course Outcomes	Foundation Knowledge	Problem Analysis	Development of Solutions	Modern Tool Usage	Individual and Teamwork	Project Management and Finance	Ethics	Life-long Learning
24FMCA3CT.1	2	1	2	-	-	-	-	-
24FMCA3CT.2	3	1	2	-	-	-	-	-
24FMCA3CT.3	3	1	3	-	-	-	-	-
24FMCA3CT.4	2	1	2	-	-	-	-	-
24FMCA3CT.5	3	1	3	-	-	-	-	-

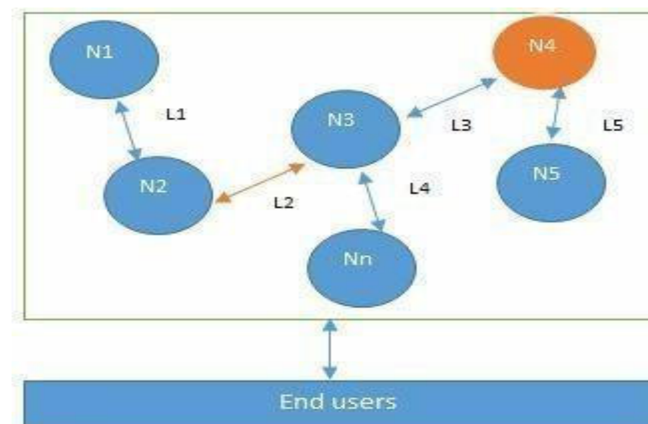
UNIT - I

Distributed systems

Understanding distributed systems is essential in order to understand block chain because basically block chain at its core is a distributed system. More precisely it is a decentralized distributed system.

Distributed systems are a computing paradigm whereby two or more nodes work with each other in a coordinated fashion in order to achieve a common outcome and it's modeled in such a way that end users see it as a single logical platform.

A node can be defined as an individual player in a distributed system. All nodes are capable of sending and receiving messages to and from each other. Nodes can be honest, faulty, or malicious and have their own memory and processor. A node that can exhibit arbitrary behavior is also known as a Byzantine node. This arbitrary behavior can be intentionally malicious, which is detrimental to the operation of the network. Generally, any unexpected behavior of a node on the network can be categorized as Byzantine. This term arbitrarily encompasses any behavior that is unexpected or malicious:



Design of a distributed system; N4 is a Byzantine node, L2 is broken or a slow network link

The main challenge in distributed system design is coordination between nodes and fault tolerance. Even if some of the nodes become faulty or network links break, the distributed system should tolerate this and should continue to work flawlessly in order to achieve the desired result. This has been an area of active research for many years

and several algorithms and mechanisms has been proposed to overcome these issues.

Here's a breakdown of how blockchain leverages distributed systems:

1. Decentralization: Unlike traditional systems with a central authority, blockchain relies on a network of independent nodes to validate and record transactions. This eliminates single points of failure and reduces the risk of censorship or control by a single entity.

2. Distributed Ledger: Each node in the network maintains a copy of the blockchain, ensuring that all participants have access to the same, up-to-date record of transactions. This distributed ledger makes it incredibly difficult to alter or tamper with the data, as any change would need to be reflected across the entire network.

3. Consensus Mechanisms: Blockchain networks use various consensus mechanisms (like Proof-of-Work or Proof-of-Stake) to ensure that all nodes agree on the validity of transactions and the state of the ledger. This mechanism is crucial for maintaining data integrity and preventing inconsistencies within the distributed system.

4. Security and Transparency: The distributed nature of blockchain, combined with cryptographic techniques, makes it highly secure and transparent. Each transaction is grouped into a block, and these blocks are linked together using cryptographic hashes, forming an immutable chain. This structure ensures that data cannot be altered without detection, and all transactions are publicly verifiable.

5. Examples: Cryptocurrencies like Bitcoin and Ethereum are prime examples of blockchain technology utilizing distributed systems. However, the applications of blockchain extend far beyond cryptocurrencies, including supply chain management, digital identity, and voting systems.

Byzantine Generals problem

Before discussing consensus in distributed systems, events in history are presented that are precursors to the development of successful and practical consensus mechanisms.

In September 1962, *Paul Baran* introduced the idea of cryptographic signatures with his paper *On distributed communications networks*. This is the paper where the concept of decentralized networks was also introduced for the very first time. Then in 1982 a thought experiment was proposed by *Lamport* whereby a group of army generals who are leading different parts of the Byzantine army are planning to attack or retreat from a city. The only way of communication between them is a messenger and they need to agree to attack at the same time in order to win. The issue is that one or more generals can be traitors and can communicate a misleading message. Therefore there is a need to find a viable mechanism that allows agreement between generals even in the presence of treacherous generals so that the attack can still take place at the same time. As an analogy with distributed systems, generals can be considered as nodes, traitors can be considered Byzantine (malicious) nodes, and the messenger can be thought of as a channel of communication between the generals.

This problem was solved in 1999 by *Castro* and *Liskov* who presented the **Practical Byzantine Fault Tolerance** (**PBFT**) algorithm. Later on in 2009, the first practical implementation was made with the invention of bitcoin where the **Proof of Work** (**PoW**) algorithm was developed as a mechanism to achieve consensus.

Consensus

Consensus is a process of agreement between distrusting nodes on a final state of data. In order to achieve consensus different algorithms can be used. It is easy to reach an agreement between two nodes (for example in client-server systems) but when multiple nodes are participating in a distributed system and they need to agree on a single value it becomes very difficult to achieve consensus. This concept of achieving consensus between multiple nodes is known as distributed consensus.

Consensus mechanisms

A consensus mechanism is a set of steps that are taken by all, or most, nodes in order to agree on a proposed state or value. For more than three decades this concept has been researched by computer scientists in the industry and Academia. Consensus mechanisms have recently come into the limelight and gained much popularity with the advent of bitcoin and blockchain.

There are various requirements

Agreement: All honest nodes decide on the same value.

- **Termination**: All honest nodes terminate execution of the consensus process and eventually reach a decision.
- **Validity**: The value agreed upon by all honest nodes must be the same as the initial value proposed by at least one honest node.
- **Fault tolerant**: The consensus algorithm should be able to run in the presence of faulty or malicious nodes (Byzantine nodes).
- **Integrity**: This is a requirement where by no node makes the decision more than once. The nodes make decisions only once in a single consensus cycle.

Types of consensus mechanism

There are various types of consensus mechanism; some common types are described as follows:

- **Byzantine fault tolerance-based**: With no compute intensive operations such as partial hash inversion, this method relies on a simple scheme of nodes that are publishing signed messages. Eventually, when a certain number of messages are received, then an agreement is reached.
- **Leader-based consensus mechanisms**: This type of mechanism requires nodes to compete for the *leader-election lottery* and the node that wins it proposes a final value.

The history of blockchain

Blockchain was introduced with the invention of bitcoin in 2008 and then with its practical implementation in 2009.

1991: In 1991, researcher scientists named Stuart Haber and W. Scott Stornetta introduce Blockchain Technology. These scientists wanted some Computational practical Solution for time-stamping the digital documents so that they couldn't be tempered or misdated. So both scientists together developed a system with the help of Cryptography. In this System, the time-stamped documents are stored in a Chain of Blocks.

1992: After that in 1992, Merkle Trees formed a legal corporation by using a system developed by Stuart Haber and W. Scott Stornetta with some more features. Hence, Blockchain Technology became efficient to store several documents to be collected into one block. Merkle used a Secured Chain of Blocks that stores multiple data records in a sequence. However, this technology became unused when the Patent came into existence in 2004.

2000: In the year 2000, Stefan Konst published his theory of cryptographic secured chains, plus ideas for implementation.

2004: In the year 2004, Cryptographic activist Hal Finney introduced a system for digital cash known as "Reusable Proof of Work". This step was the game-changer in the history of Blockchain and Cryptography. This System helps others to solve the Double Spending Problem by keeping the ownership of tokens registered on a trusted server.

2008: After that 2008, Satoshi Nakamoto conceptualized the concept of "Distributed Blockchain" in his white paper: "A Peer to Peer Electronic Cash System". He modified the model of Merkle Tree and created a system that is more secure and contains the secure history of data exchange. His System follows a peer-to-peer network of time stamping. His system became so useful that Cryptography became the backbone of Blockchain.

2009: After that, the evolution of Blockchain is steady and promising and became a need in various fields. In 2009, Satoshi Nakamoto Releases Bitcoin White Paper. Blockchain technology is so secure that the following surprising news will give proof of that. A person named, James Howells was an IT worker in the United Kingdom, he starts mining bitcoins which are part of Blockchain in 2009 and stopped this in 2013. He spends $17,000 on it and after he stopped he sells the parts of his laptop on eBay and keep the drive with him so that when he needs to work again on bitcoin he will utilize it but while cleaning his house in 2013,he thrashed his drive with garbage and now his bitcoins cost nearly $127 million. This money now remains unclaimed in the Bitcoin system.

2014: The year 2014 is marked as the turning point for blockchain technology. Blockchain technology is separated from the currency and Blockchain 2.0 is born. Financial institutions and other industries started shifting their focus from digital currency to the development of blockchain technologies.

2015: In 2015, Ethereum Frontier Network was launched, thus enabling developers to write smart contracts and dApps that could be deployed to a live network. In the same year, the Linux Foundation launched the Hyperledger project.

2016: The word Blockchain is accepted as a single word instead of two different concepts as they were in Nakamoto's original paper. The same year, a bug in the Ethereum DAO code was exploited resulting in a hard fork of the Ethereum Network. The Bitfinex bitcoin exchange was hacked resulting in 120,000 bitcoin being stolen.

2017: In the year 2017, Japan recognized Bitcoin as a legal currency. Block.one company introduced the EOS blockchain operating system which was designed to support commercial decentralized applications.

2018: Bitcoin turned 10 in the year 2018. The bitcoin value continued to drop, reaching the value of $3,800 at the end of the year. Online platforms like Google, Twitter, and Facebook banned the advertising of cryptocurrencies.

2019: In the year 2019, Ethereum network transactions exceeded 1 million per day. Amazon announced the general availability of the Amazon Managed Blockchain service on AWS.

2020: Stablecoins were in demand as they promised more stability than traditional cryptocurrencies. The same year Ethereum launched Beacon Chain in preparation for Ethereum 2.0.

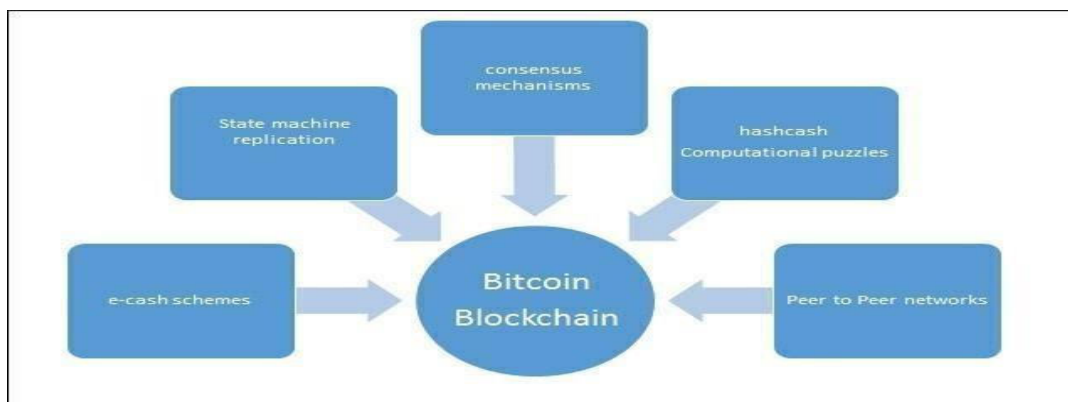
2022 : Ethereum has shifted from Proof of Work(PoW) to Proof of Stake(PoS) consensus mechanism. The original Ethereum mainnet merged with Beacon Chain which has Proof-of-Stake. Now it is existing as one chain. Ethereum's energy consumption has reduced by ~99.95%

Electronic cash

Just as understanding the concepts of distributed systems is necessary in order to understand blockchain technology, the idea of electronic cash is also essential to appreciate the first and astonishingly successful application of blockchain: the bitcoin, or broadly cryptocurrencies. Theoretical concepts in distributed systems such as consensus algorithms provided the basis of the practical implementation of Proof of Work algorithms in bitcoin; moreover, ideas from different electronic cash schemes also paved the way for the invention of cryptocurrencies, specifically bitcoin.

The aforementioned technologies and their history, it is easy to see how ideas and concepts from electronic cash schemes and distributed systems were combined together to invent bitcoin and what now is known as blockchain.

This can also be visualized with the help of the following diagram:



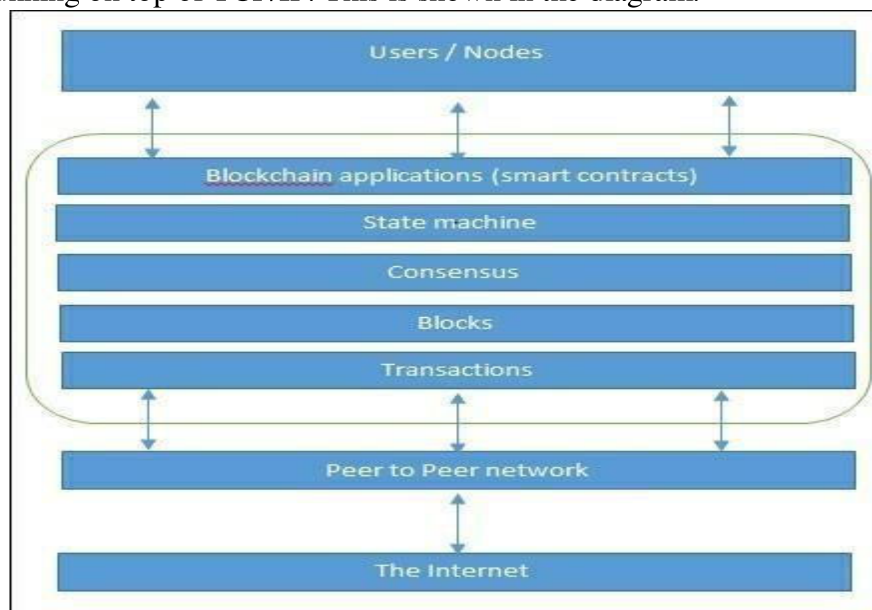
The various ideas that helped with the invention of bitcoin and block chain

Introductionto block chain

There are various definitions of block chain: it depends on how you look at it. If you look at it from a business perspective it can be defined in that context, if you look at it from a technical perspective one can define it in view of that.

Blockchain at its core is a peer-to-peer distributed ledger that is cryptographically secure, append- only, immutable (extremely hard to change), and updateable only via consensus or agreement among peers.

Blockchain can be thought of as a layer of a distributed peer-to-peer network running on top of the Internet, as can be seen below in the diagram. It is analogous to SMTP, HTTP, or FTP running on top of TCP/IP. This is shown in the diagram.

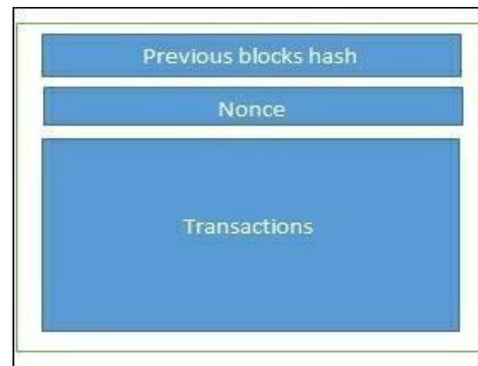


The network view of a blockchain

From a business point of view a blockchain can be defined as a platform whereby peers can exchange values using transactions without the need for a central trusted arbitrator. This is a powerful concept and once readers understand it they will realize the tsunamic potential of blockchain technology. This allows blockchain to be a decentralized consensus mechanism where no single authority is in charge of the database.

A block is simply a selection of transactions bundled together in order to organize them logically. It is made up of transactions and its size is variable depending on the type and design of the blockchain in use. A reference to a previous block is also included in the block unless it's a genesis block. A genesis block is the first block in the blockchain that was hardcoded at the time the blockchain was started. The structure of a block is also dependent on the type and design of a blockchain, but generally there are a few attributes that are essential to the functionality of a block, such as the block header, pointers to previous blocks, the time stamp, nonce, transaction counter, transactions, and other attributes.

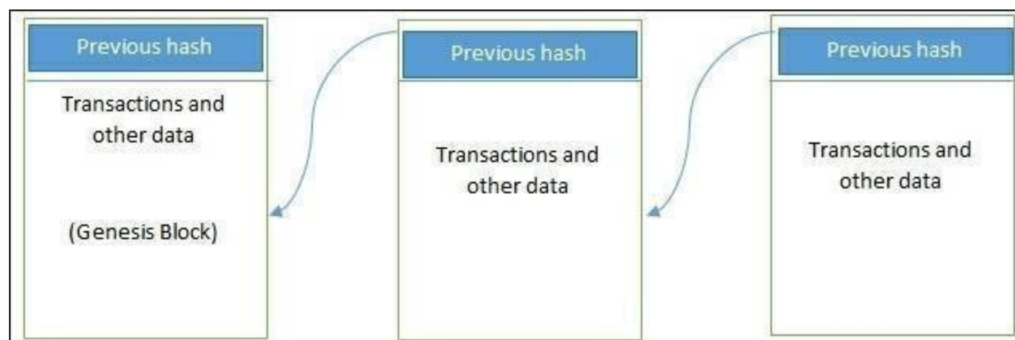
The structure of a block



Various technical definitions of blockchains

- Blockchain is a decentralized consensus mechanism. In a blockchain, all peers eventually come to an agreement regarding the state of a transaction.
- Blockchain is a distributed shared ledger. Blockchain can be considered a shared ledger of transactions. The transaction are ordered and grouped into blocks. Currently, the real-world model is based on private databases that each organization maintains whereas the distributed ledger can serve as a single source of truth for all member organizations that are using the blockchain.
- Blockchain is a data structure; it is basically a linked list that uses hash pointers instead of normal pointers. Hash pointers are used to point to the previous block.

The structure of a generic blockchain can be visualized with the help of the following diagram:



Generic structure of a blockchain

Generic elements of a blockchain

Addresses are unique identifiers that are used in a transaction on the blockchain to denote senders and recipients. An address is usually a public key or derived from a public key. While addresses can be reused by the same user, addresses themselves are unique. In practice, however, a single user may not use the same address again and generate a new one for each transaction.

Transaction: A transaction is the fundamental unit of a blockchain. A transaction represents a transfer of value from one address to another.

Block : A block is composed of multiple transactions and some other elements such as the previous block hash (hash pointer), timestamp, and nonce.

Peer-to-peer network : As the name implies, this is a network topology whereby all peers can communicate with each other and send and receive messages.

Scripting or programming language

This element performs various operations on a transaction. Transaction scripts are predefined sets of commands for nodes to transfer tokens from one address to another and perform various other functions. Turing complete programming language is a desirable feature of blockchains; however, the security of such languages is a key question and an area of important and ongoing research.

Virtual machine

This is an extension of a transaction script. A virtual machine allows Turing complete code to be run on a blockchain (as smart contracts) whereas a transaction script can be limited in its operation.
Virtual machines are not available on all blockchains; however, various blockchains use virtual machines to run programs, for example **Ethereum Virtual Machine** (**EVM**) and **Chain Virtual Machine** (**CVM**).

State machine

Ablockchain can be viewed as a state transition mechanismwhereby a state is modified from its initial form to the next and eventually to a final form as a result of a transaction execution and validation process by nodes.

Nodes

A node in a blockchain network performs various functions depending on the role it takes. A node can propose and validate transactions and perform mining to facilitate consensus and secure the blockchain. This is done by following a consensus protocol. (Most commonly this is PoW.) Nodes can also perform other functions such as simple payment verification (lightweight nodes), validators, and many others functions depending on the type of the blockchain used and the role assigned to the node.

Smart contracts

These programs run on top of the blockchain and encapsulate the business logic to be executed when certain conditions are met. The smart contract feature is not available in all blockchains but is now becoming a very desirable feature due to the flexibility and power it provides to the blockchain applications.

Features of a blockchain

A blockchain performs various functions. These are described below in detail.

Distributed consensus

Distributed consensus is the major underpinning of a blockchain. This enables a blockchain to present a single version of truth that is agreed upon by all parties without the requirement of a central authority.

Transaction verification

Any transactions posted from nodes on the blockchain are verified based on a predetermined set of rules and only valid transactions are selected for inclusion in a block.

Platforms for smart contracts

A blockchain is a platform where programs can run that execute business logic on behalf of the users. As explained earlier, not all blockchains have a mechanism to execute smart contracts; however, this is now a very desirable feature.

Transferring value between peers

Blockchain enables the transfer of value between its users via tokens. Tokens can be thought of as a carrier of value.

Generating cryptocurrency

This is an optional feature depending on the type of blockchain used. A blockchain can generate cryptocurrency as an incentive to its miners who validate the transactions and spend resources in order to secure the blockchain.

Smart property

For the first time it is possible to link a digital or physical asset to the blockchain in an irrevocable manner, such that it cannot be claimed by anyone else; you are in full control of your asset and it cannot be double spent or double owned. Compare it with a digital music file, for example,
which can be copied many times without any control; on a blockchain, however, if you own it no one else can claim it unless you decide to transfer it to someone. This feature has far-reaching implications especially in **Digital Rights Management** (**DRM**) and electronic cash systems where double spend detection is a key requirement. The double spend problem was first solved in bitcoin.

Provider of security

Blockchain is based on proven cryptographic technology that ensures the integrity and availability of data. Generally, confidentiality is not provided due to the requirements

of transparency. This has become a main barrier for its adaptability by financial institutions and other industries that need

Immutability

This is another key feature of blockchain: records once added onto the blockchain are immutable. There is the possibility of rolling back the changes but this is considered almost impossible to do as it will require an unaffordable amount of computing resources. For example, in much desirable case of bitcoin if a malicious user wants to alter the previous blocks then it would require computing the PoW again for all those blocks that have already been added to the blockchain. This difficulty makes the records on a blockchain practically immutable.

Uniqueness

This feature of blockchain ensures that every transaction is unique and has not been spent already. This is especially relevant in cryptocurrencies where much desirable detection and avoidance of double spending are a key requirement.

Smart contracts

Blockchain provides a platform to run smart contracts. These are automated autonomous programs that reside on the blockchain and encapsulate business logic and code in order to execute a required function when certain conditions are met. This is indeed a revolutionary feature of blockchain as it allows flexibility, programmability, and much desirable control of actions that users of blockchain need to perform according to their specific business requirements.

Applications of blockchain technology

Blockchain technology has a multitude of applications in various sectors including but not limited to finance, government, media, law, and arts.

How blockchains accumulate blocks

1. A node starts a transaction by signing it with its private key.
2. The transaction is propagated (flooded) by using much desirable Gossip protocol to peers, which validates the transaction based on pre-set criteria. Usually, more than one node is required to validate the transactions.
3. Once the transaction is validated, it is included in a block, which is then propagated on to the network. At this point, the transaction is considered confirmed.

4. The newly created block now becomes part of the ledger and the next block links itself cryptographically back to this block. This link is a hash pointer. At this stage, the transaction gets its second confirmation and the block gets its first.
5. Transactions are then reconfirmed every time a new block is created. Usually, six confirmations in the bitcoin network are required to consider the transaction final.

Steps 4 and 5 can be considered non-compulsory as the transaction itself is finalized in step 3; however, block confirmation and further transaction reconfirmations, if required, are then carried out in steps 4 and 5.

Tiers of blockchain technology

First, the three levels discussed below were originally described by *Melanie Swan* in her book *Blockchain, Blueprint for a New Economy* as tiers of blockchain categorized on the basis of applications in each category. In addition to this, Tier X or Generation X is discussed later.

Blockchain 1.0

This was introduced with the invention of bitcoin and is basically used for cryptocurrencies. Also, as bitcoin was the first implementation of cryptocurrencies it makes sense to categorize Generation 1 of blockchain technology to only include cryptographic currencies. All alternative coins and bitcoin fall into this category. This includes core applications such as payments and applications.

Blockchain 2.0

Generation 2.0 blockchains are used by financial services and contracts are introduced in this generation. This includes various financial assets, for example derivatives, options, swaps, and bonds. Applications that are beyond currency, finance, and markets are included at this tier.

Blockchain 3.0

Generation 3 blockchains are used to implement applications beyond the financial services industry and are used in more general-purpose industries such as government, health, media, the arts, and justice.

Generation X (Blockchain X)

This is a vision of blockchain singularity where one day we will have a public

blockchain service available that anyone can use just like the Google search engine. It will provide services in all realms of society. This is a public open distributed ledger with general-purpose rational agents (Machina Economicus) running on blockchain.

Types of blockchain

Based on the way blockchain has evolved over the last few years, it can be divided into multiple types with distinct but sometimes partly overlapping attributes.

Public blockchains

As the name suggests, these blockchains are open to the public and anyone can participate as a node in the decision-making process. Users may or may not be rewarded for their participation. These ledgers are not owned by anyone and are publicly open for anyone to participate in. All users of the permission-less ledger maintain a copy of the ledger on their local nodes and use a distributed consensus mechanism in order to reach a decision about the eventual state of the ledger. These blockchains are also known as permission-less ledgers.

Private blockchains

Private blockchains as the name implies are private and are open only to a consortium or group of individuals or organizations that has decided to share the ledger among themselves.

- **Consortium BlockChain :** A consortium blockchain is a semi-decentralized type where more than one organization manages a blockchain network. More than one organization can act as a node in this type of blockchain and exchange information or do mining.
- **Hybrid BlockChain :** A hybrid blockchain is a combination of the private and public blockchain. It uses the features of both types of blockchains that is one can have a private permission-based system as well as a public permission-less system. With such a hybrid network, users can control who gets access to which data stored in the blockchain. Only a selected section of data or records from the blockchain can be allowed to go public keeping the rest as confidential in the private network. The hybrid system of blockchain is flexible so that users can easily join a private blockchain with multiple public blockchains. A transaction in a private network of a hybrid blockchain is usually verified within that network.

Semi-private blockchains

Here part of the blockchain is private and part of it is public. The private part is controlled by a group of individuals whereas the public part is open for participation by anyone.

Distributed ledger

As the name suggests, this ledger is distributed among its participants and spread across multiple sites or organizations. This type can either be private or public. The key idea is that, unlike many other blockchains, the records are stored contiguously instead of sorted into blocks. This concept is used in Ripple.

Shared ledger

This is generic term that is used to describe any application or database that is shared by the public or a consortium.

Tokenized blockchains

These blockchains are standard blockchains that generate cryptocurrency as a result of a consensus process via mining or via initial distribution.

Tokenless blockchains

These are probably not real blockchains because they lack the basic unit of transfer of value but are still valuable in situations where there is no need to transfer value between nodes and only sharing some data among various already trusted parties is required.

Consensus in blockchain

Consensus is basically a distributed computing concept that has been used in blockchain in order to provide a means of agreeing to a single version of truth by all peers on the blockchain network. This concept was discussed in the distributed systems section earlier in this chapter.

Roughly, the following two categories of consensus mechanismexist:

1. Proof-based, leader-based, or the *Nakamoto consensus* whereby a leader is elected and proposes a final value
2. Byzantine fault tolerance-based, which is a more traditional approach based on rounds of votes

Consensus algorithms that are available today or are being researched in the context of blockchain are presented later. This is not an exhaustive list but an attempt has been made to present all important algorithms.

Proof of Work

This type of consensus mechanismrelies on proof that enough computational resources have been spent before proposing a value for acceptance by the network. This is used in bitcoin and other cryptocurrencies. Currently, this is the only algorithm that has proven astonishingly successful against Sybil attacks.

Proof of Stake

This algorithm works on the idea that a node or user has enough stake in the system; for example the user has invested enough in the system so that any malicious attempt would

outweigh the benefits of performing an attack on the system. This idea was first introduced by Peercoin and is going to be used in the Ethereum blockchain. Another important concept in **Proof of Stake** (**PoS**) is coin age, which is a derived from the amount of time and the number of coins that have not been spent. In this model, the chances of proposing and signing the next block increase with the coin age.

Delegated Proof of Stake

Delegated Proof of Stake (**DPOS**) is an innovation over standard PoS whereby each node that has stake in the system can delegate the validation of a transaction to other nodes by voting. This is used in the bitshares blockchain.

Proof of importance

This idea is important and different from Proof of Stake. Proof of importance not only relies on how much stake a user has in the system but it also monitors the usage and movement of tokens by the user to establish a level of trust and importance. This is used in Namecoin.

Practical Byzantine Fault Tolerance

Practical Byzantine Fault Tolerance (**PBFT**) achieves state machine replication, which provides tolerance against Byzantine nodes. Various other protocols, including but are not limited to PBFT, PAXOS, RAFT, and **Federated Byzantine Agreement** (**FBA**), are also being used or have been proposed for use in many different implementations of distributed systems and blockchains.

CAPtheoremand blockchain

This is also known as Brewer's theorem, introduced originally by Eric Brewer as a conjecture in 1998; in 2002 it was proved as a theorem by Seth Gilbert and Nancy Lynch.

The theorem states that any distributed system cannot have Consistency, Availability, and Partition tolerance simultaneously:

- **Consistency** is a property that ensures that all nodes in a distributed system have a single latest copy of data
- **Availability** means that the system is up, accessible for use, and is accepting incoming requests and responding with data without any failures as and when required
- **Partition tolerance** ensures that if a group of nodes fails the distributed system still continues to operate correctly

The preceding diagram shows that only two properties at a time can be achieved. Either AP, CA, or CP.

- If we opt for CP (consistency and partition tolerance), we sacrifice availability.
- If we opt for AP (availability and partition tolerance), we sacrifice consistency.
- If we opt for AC (availability and consistency), we sacrifice partition tolerance.

Benefits and limitations of blockchain

Numerous benefits of blockchain technology are being discussed in the industry and proposed by thought leaders around the world in blockchain space. Benefits are listed and discussed as follows.

Decentralization

This is a core concept and benefit of blockchain. There is no need for a trusted third party or intermediary to validate transactions; instead a consensus mechanism is used to agree on the validity of transactions.

Transparency and trust

As blockchains are shared and everyone can see what is on the blockchain, this allows the system to be transparent and as a result trust is established. This is more relevant in scenarios such as the disbursement of funds or benefits where personal discretion should be restricted.

Immutability

Once the data has been written to the blockchain, it is extremely difficult to change it back. It is not truly immutable but, due to the fact that changing data is extremely difficult and almost impossible, this is seen as a benefit to maintaining an immutable ledger of transactions.

High availability

As the system is based on thousands of nodes in a peer-to-peer network, and the data is replicated and updated on each and every node, the system becomes highly available. Even if nodes leave the network or become inaccessible, the network as a whole continues to work, thus making it highly available.

Highly secure: All transactions on a blockchain are cryptographically secured and provide integrity.

Faster dealings

In the financial industry, especially in post-trade settlement functions, blockchain can play a vital role by allowing the quicker settlement of trades as it does not require a lengthy process of verification, reconciliation, and clearance because a single version of agreed upon data is already available on a shared ledger between financial organizations.

Cost saving

As no third party or clearing houses are required in the blockchain model, this can massively eliminate overhead costs in the form of fees that are paid to clearing houses or trusted third parties.

Challenges and limitations of blockchain technology

Speed and performance: Blockchain is considerably slower than the traditional database because blockchain technology carries out more operations. First, it performs signature verification, which involves signing transactions cryptographically. Blockchain also relies on a consensus mechanism to validate transactions. Some consensus mechanisms, such as proof of work, have a low transaction throughput. Finally, there is redundancy, where the network

requires each node to play a crucial role in verifying and storing each transaction.

High implementation cost: Blockchain is costlier compared to a traditional database. Additionally, businesses need proper planning and execution to integrate blockchain into their process.

Data modification: Blockchain technology does not allow easy modification of data once recorded, and it requires rewriting the codes in all of the blocks, which is time-consuming and expensive. The downside of this feature is that it is hard to correct a mistake or make any necessary adjustments.

Decentralization

Decentralization is not a new concept; it has been used in strategy, management, and governance for a long time. The basic idea of decentralization is to distribute control and authority to peripheries instead of one central authority being in full control of the organization. This results in several benefits for organizations. such as increased efficiency, quicker decision making, better motivation and a reduced burden on top management.

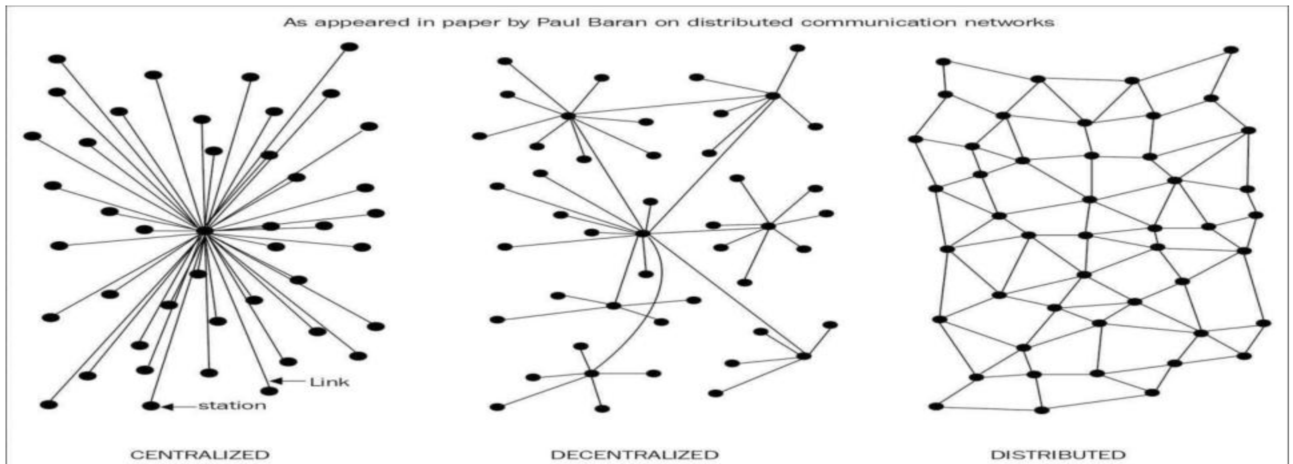
Decentralizationusing blockchain

Decentralization is a core benefit and service provided by the blockchain technology. Blockchain by design is a perfect vehicle for providing a platform that does not need any intermediaries and can function with many different leaders chosen via consensus mechanisms. This model allows anyone to compete to become the decision-making authority. This competition is governed by a consensus mechanism and the most commonly used method is known as **Proof of Work** (**PoW**).

Decentralization is applied in varying degrees from semi-decentralized to fully decentralized depending on the requirements and circumstances. Decentralization can be viewed from a blockchain perspective as a mechanism that provides a way to remodel existing applications and paradigms or build new applications in order to give full control to users.

Information and communication technology (**ICT**) has conventionally been based on a centralized paradigmwhereby database or application servers are under the control of a central authority, such as a system administrator. With bitcoin and the advent of the blockchain technology, this model has changed and now the technology that allows anyone to start a decentralized system (and operate it with no single point of failure or single trusted authority) is available. It can either be run autonomously or by requiring some human intervention depending on the type and model of governance used in the decentralized application running on the blockchain.

Centralized systems are conventional (client--server) IT systems whereby there is a single authority that controls the system and is solely in-charge of all operations on the system. All users of a central system are dependent on a single source of service. Online service providers, such as eBay, Google, Amazon, Apple's App Store, and the majority of other providers, use this common model of delivering services. On the other hand, in a distributed system, the data and computation are spread across multiple nodes in the network. Sometimes, this term is confused with parallel computing.



Different types of network/system

The key difference between a decentralized system and distributed system is that in a distributed system, there still exists a central authority that governs the entire system, whereas in a decentralized system, no such authority exists. A decentralized system is a type of network whereby nodes are not dependent on a single master node; instead, control is distributed among many nodes. For example, this is analogous to a model where each department in an organization has its own database server that they are in charge of, thus taking away the power from the central server and distributing it to the sub-departments that manage their own databases.

Methods of decentralization

There are two methods that can be used to achieve decentralization. These methods are discussed in detail in the following sections.

i. Disintermediation

This can be explained with the help of an example. Imagine you want to send money to your friend in another country. You go to a bank that will transfer your money to the bank in the country of your choice for a fee. In this case, the bank keeps a central database that is updated, confirming that you have sent the money. With blockchain

technology, it is possible to send this money directly to your friend without the need for a bank. All you need is the address of your friend on the blockchain. This way, the intermediary is no longer required and decentralization is achieved by disintermediation. Nevertheless, this model can be used not only in finance but also in many other different industries.

ii. Through competition

In this method, a group of service providers compete with each other in order to be selected for the provision of services by the system. This paradigm does not achieve complete decentralization, but to a certain degree ensures that an intermediary or service provider is not monopolizing the service. In the context of blockchain technology, a system can be envisioned in which smart contracts can choose an external data provider from a large number of providers based on their reputation, previous score, reviews, and quality of service. This will not result in full decentralization, but it allows smart contracts to make a free choice based on the criteria mentioned earlier. This way, an environment of competition is cultivated among service providers, whereby they compete with each other to become the data provider of choice.

In varying levels of decentralization are On the left-hand side, there is a conventional approach where a central system is in control; on the right-hand side, complete disintermediation is achieved; and in middle, competing intermediaries or service providers. In the middle, intermediaries or service providers are selected based on reputation or voting, thus achieving partial decentralization.

While there are many benefits of decentralization--including but not limited to transparency, efficiency, cost saving, development of trusted ecosystems, and in some cases privacy and anonymity-some challenges, such as security requirements, software bugs, and human errors, also need to be looked at thoroughly. For example, in a decentralized system such as bitcoin or Ethereum, where security is usually provided by private keys, how can it be ensured that a smart property associated with these private keys cannot be rendered useless if, due to a human error, the private keys are lost or if, due to a bug in the smart contract code, the decentralized application is vulnerable to attack by adversaries Before we embark on a journey to decentralize everything using blockchain and decentralized applications, it is important to understand that not everything is required to (or can be) decentralized.

Routes todecentralization

Even though there are systems that existed before bitcoin or blockchain that can be classed as decentralized to a certain degree, such as BitTorrent or Gnutella file sharing, with the advent of the blockchain technology many initiatives are being taken in order leverage this new technology for decentralization. Usually, the bitcoin blockchain is the first choice for many as it has proven to be the most resilient and secure blockchain with a market cap of almost 12 billion dollars. An alternative approach is to use other blockchains, such as Ethereum, which is currently the tool of choice of many developers for building decentralized applications.

How to decentralize

A framework has been proposed by *Arvind Narayanan* and others that can be used to evaluate the decentralization requirements of a variety of things in the context of blockchain technology. The framework basically proposes four questions that, once answered, provide a clear idea as to how a system can be decentralized.

These questions are listed as follows:

1. What is being decentralized?
2. What level of decentralization is required?
3. What blockchain is used?
4. What security mechanism is used?

The first question simply asks what system is being decentralized. This can be any system, for example an Identity system or trading. The next question can be answered by specifying the level of decentralization required by looking at the scale of decentralization discussed earlier. It can be full disintermediation or partial disintermediation. The third question is quite straightforward, where developers can make a choice as to which blockchain is suitable for a particular application. It can be bitcoin blockchain, Ethereum blockchain, or any other blockchain that is deemed fit for a specific application. Finally, a key question needs to be answered about the security mechanism as to how the security of a decentralized system can be guaranteed. It can be Atomicity, for example, whereby either the transaction executes in full or does not execute at all. In other words, it is all or nothing. This ensures the integrity of the system. Other mechanisms can include reputation, which allows varying degrees of trust in a system.

Examples

1. **Answer 1**: Money transfer system.
2. **Answer 2**: Disintermediation.
3. **Answer 3**: Bitcoin.
4. **Answer 4**: Atomicity.

By answering these four questions, it can be shown how a payment system can be decentralized.

Blockchain and full ecosystem decentralization

In order to achieve complete decentralization, it is necessary that the environment around the blockchain is also decentralized. Blockchain itself is a distributed ledger that runs on top of conventional systems. These elements include storage, communication, and computation. There are other factors, such as Identity and Wealth, that are traditionally based on centralized paradigms and there's a need to decentralize these aspects too in order to achieve a fully decentralized ecosystem.

Storage

Data can be stored directly in a blockchain, and with this, it does achieve decentralization, but a major disadvantage of this approach is that blockchain is not suitable for storing large amounts of data by design. It can store simple transactions and some arbitrary data but is certainly not suitable for storing images or large blobs of data, as is the case in traditional database systems. A better alternative is to use **distributed hash tables** (**DHTs**). DHTs were originally used in peer-to-peer file sharing software, such as BitTorrent, Napster, Kazaa, and Gnutella. DHT research was made popular by CAN, Chord, Pastry, and Tapestry projects. Two main requirements here are high availability and link stability, which means that data should be available when required and network links should also always be accessible.

Inter Planetary File System (**IPFS**) by *Juan Benet* possesses both of these properties and the vision is to provide a decentralized World Wide Web by replacing the HTTP protocol. IPFS uses Kademlia DHT and merkle **DAG** (**Directed Acyclic Graph**) to provide the storage and searching functionality, respectively.

Also, a Git-based version control mechanism is used in IPFS to provide structure and control over the versioning of data.

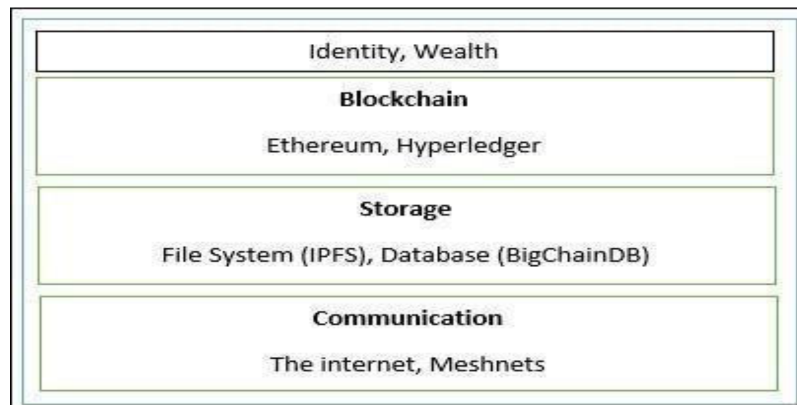
There are other alternatives, such as Ethereum swarm, storj, and maidsafe. Ethereum has its own decentralized and distributed ecosystem that uses Swarm for storage and the whisper protocol for communication. Maidsafe is aiming to provide a decentralized World Wide Web. All these projects will be discussed later in the book in more detail.

BigChainDB is another storage layer decentralization project aimed at providing a scalable, fast, and linearly scalable decentralized database as opposed to a traditional filesystem. BigChainDB complements decentralized processing platforms and file systems such as Ethereum and IPFS.

Communication

It is generally considered that the Internet (the communication layer in blockchain) is decentralized. This is true to some extent as the original vision of the Internet was to develop a decentralized system. Services such as e-mail and online storage are all now based on a paradigm where the service provider is in control and users trust them to give them access to the service when required. This model is based on the trust of the central authority (the service provider) and users are not in control of their data; even passwords are stored on trusted third-party systems. There is a need to provide control to individual users in such a way that access to their data is guaranteed and is not dependent on a single third party. Access to the Internet (the communication layer) is based on Internet service providers (ISPs) that act as a central hub for Internet users. If the ISP is shut down for political or any other reasons, then no communication is possible in this model. An alternative is to use mesh networks. Even though they are limited in functionality as compared to the Internet, they still provide a decentralized alternative where nodes can talk directly to each other without a central hub such as an ISP.

As mentioned earlier, the original vision of the Internet was to build a decentralized



network; however, over the years, with the advent of large-scale service providers such as Google, Amazon, and eBay, the control is shifting toward the big players. For example, e-mail is a decentralized system at its core; anyone can run an e-mail server with minimal effort and can start sending and receiving e- mails, but there is a better alternative available that is already providing a managed service for end users, so there is a natural inclination toward selecting a centralized service as it is more convenient and free. Free services, however, are being offered at the cost of valuable personal data and many users are not aware of this fact. This is one example that shows how the Internet has moved toward centralization. Blockchain has once again given this vision of decentralization to the world and now concerted efforts are being made to harness this technology and gain the benefits that it can provide.

Computation

Decentralization of computing or processing is achieved by a blockchain technology such as Ethereum, where smart contracts with embedded business logic can run on the network. Other blockchain technologies also provide similar processing layer platforms where business logic can run over the network in a decentralized manner.

The decentralized ecosystem on the bottom layer, Internet or Meshnets provides a decentralized communication layer, then a storage layer uses technologies such as IPFS and BigChainDB to enable decentralization, and finally, you see the blockchain that serves as a decentralized processing layer. Blockchain can, in a limited way, provide a storage layer too, but that seriously hampers the speed and capacity of the system; therefore, other solutions such as IPFS and BigChainDB are more suitable to store large amounts of data in a decentralized way. At the top, the Identity and Wealth layers are shown. Identity on the Internet is a very big topic and systems such as bitAuth and OpenID have provided authentication and identification services.

Smart contract

A smart contract can be thought of as a small decentralized program. Smart contracts do not necessarily need a blockchain to run; however, due to the security benefits that the blockchain technology provides, it is now becoming almost a standard to use blockchain as a decentralized execution platform for smart contracts. A smart contract usually contains some business logic and a limited amount of data. Actors or participants in the blockchain use these smart contracts or they run autonomously on behalf of the network participants.

Decentralized organizations

Decentralized organization (**DOs**) are software programs that run on a blockchain and are based on the idea of real human organizations with people and protocols. Once a DO, in the form of a smart contract or a set of smart contracts, is added to the blockchain, it becomes decentralized and parties interact with each other based on the code defined within the DO software.

Decentralized autonomous organizations

Just like DOs, a **Decentralized autonomous organization** (**DAO**) is also a computer program than runs on top of a blockchain and embedded within it are governance and business logic rules. DAO and DO are basically the same thing, but the main difference is that DAOs are autonomous, which means that they are fully automated and contain artificially intelligent logic, whereas DOs lack this feature and rely on human input in order to execute business logic.

Ethereum blockchain led the way with the introduction of DAOs for the first time. In DAO, the code is considered the governing entity rather than humans or paper contracts. A *Curator*, however, is a human entity that participates as someone who maintains this code and acts as a proposal evaluator for the community. DAOs are capable of hiring external *Contractors* if enough input is received from the token holders (participants). The most famous DAO project is *The DAO* as it raised 168 million US dollars in its crowd-funding phase. The DAO project was designed to be a venture capital fund which was aimed at providing a decentralized business model with no single entity as an owner. Unfortunately, this was hacked due to a bug in the DAO code and millions of dollars' worth of **Ether currency** (**ETH**) were siphoned out of the DAO into a child DAO created by the hackers. It required a hard fork on the Ethereum blockchain to reverse the impact of the hack and initiate the recovery of the funds. This incident opened up a debate on the security, quality, and the need for thorough testing of the code in smart contracts in order to ensure integrity and adequate control. There are projects underway, especially in Academia, that are looking to formalize smart contract coding.

Currently, DAOs do not have any legal status even though they may contain some intelligent code that enforces some protocols and conditions, but these rules have no value in the current real-world legal system. One day, perhaps an *autonomous agent* that is commissioned and permissioned by a law enforcement agency or a regulator containing rules and regulations could be embedded in a DAO, to ensure the integrity of the DAO from a legal and compliance perspective. An **Autonomous Agent** (**AA**) is a piece of code that runs without human intervention. The fact that DAOs are purely decentralized entities makes it possible to run them in any physical jurisdiction.

Therefore, they raise a big question as to how a current legal system would work with such a varied mix of different jurisdictions and geographies.

Decentralized autonomous corporations

DAOs, **Decentralized autonomous corporations** (**DACs**) are a similar concept but are considered a smaller subset of DAOs. The definitions of DACs and DAOs can sometimes overlap, but a general difference is that DAOs are usually considered to be nonprofit, whereas DACs can make money via shares offered to the participants and by paying dividends. These corporations can run a business automatically without human intervention based on the logic programmed within them.

Decentralized autonomous societies

Decentralized autonomous societies (**DASs**) are a concept whereby entire societies can function on a blockchain with the help of multiple complex smart contracts and a combination of DAOs and **Decentralized applications** (**DAPPs**) running autonomously. This model does not mean an outlaw approach, nor is it based on a totally libertarian ideology; instead, many services that a government offers can be delivered via blockchain, such as Government Identity Card systems, passport issuance, and records of deeds, marriages, and births. Another theory is that, if a government is corrupt and central systems do not provide the satisfactory levels of trust that a society needs, then the society can start its own virtual society on a blockchain that is driven by decentralized consensus and is transparent. This might be seen as a libertarian or cypherpunk dream but is entirely possible on a blockchain.

Decentralized applications

All ideas mentioned earlier come under the larger umbrella of decentralized applications. All DAOs, DACs, and DOs are basically decentralized applications that run on top of a blockchain in a peer-to- peer network. This is the latest advancement in technology with regard to decentralization.

Decentralized applications or DAPPs are software programs that can run on their own blockchain, use another already existing established blockchain, or use only protocols of an existing blockchain solution. These are called Type I, Type II, and Type III DAPPs.

Requirements of a decentralized application

In order for an application to be considered a decentralized application, it must meet the following criteria. This definition was provided by *David Johnston* and others in their whitepaper called *The General Theory of Decentralized Applications, Dapps*:

1. The DAPP should be fully open source and autonomous and no single entity should be in control of a majority of its tokens. All changes to the application must be consensus-driven based on the feedback given by the community.
2. Data and records of operations of the application must be cryptographically secured and stored on a public, decentralized blockchain in order to avoid any central points of failure.

3. A cryptographic token must be used by the application in order to provide access and rewards to those who contribute value to the applications, for example, miners in bitcoin.
4. The tokens must be generated by the decentralized application according to a standard cryptographic algorithm. This generation of tokens acts as a proof of the value to contributors (for example, miners).

Operations of a DAPP

Establishment of consensus by a DAPP can be achieved using consensus algorithms such as Proof of Work and Proof of Stake. So far, only PoW has been found to be incredibly resistant to 51% attacks, as is evident from bitcoin. Furthermore, a DAPP can distribute tokens (coins) via mining, fundraising, and development.

Examples

KYC-Chain

This application provides a facility to manage **Know Your Customer** (**KYC**) data in a secure and convenient way based on smart contracts.

OpenBazaar

This is a decentralized peer-to-peer network that allows commercial activities directly between sellers and buyers instead of relying on a central party, as opposed to conventional providers such as eBay and Amazon. It should be noted that this system is not built on top of a blockchain; instead, distributed hash tables are used in a peer- to-peer network in order to enable direct communication and data sharing between peers. It makes use of bitcoin as a payment network, however.

Lazooz : This is a decentralized equivalent of Uber. It allows peer-to-peer ride sharing and users can be incentivized by *proof of movement* and can earn Zooz coins.

Platforms for decentralization

There are many platforms available for decentralization now. Many companies around the world have introduced platforms that promise to make distributed application development easy, accessible, and secure for users. Some prominent names are discussed here.

Ethereum

Ethereum tops the list as being the first blockchain that introduced a Turing-complete language and the concept of a virtual machine. This is in contrast to the limited scripting language in bitcoin and many other cryptocurrencies. With the availability of this Turing-complete language called Solidity, endless possibilities have opened for the development of decentralized applications. This was proposed in 2013 by *Vitalik Buterin* and provides a public blockchain to develop smart contracts and decentralized applications. Currency tokens on Ethereum are called Ethers.

Maidsafe

Maidsafe provides a **SAFE** (**Secure Access for Everyone**) network that is made up of unused computing resources, such as storage, processing power, and the data connections of its users. The files on the network are divided into small chunks of data that are encrypted and distributed throughout the network randomly. This data can only be retrieved by its respective owner. One key innovation is that duplicate files are automatically rejected on the network, which helps reduce the need for additional computing resources to manage the load. It uses Safecoin as a token to incentivize its contributors.

Lisk

Lisk is a blockchain application development and cryptocurrency platform. It allows developers to use JavaScript to build decentralized applications and host them in their own respective sidechains. Lisk uses the **Delegated Proof of Stake** (**DPOS**) mechanism for consensus whereby 101 nodes can be elected to secure the network and propose blocks. It uses the Node.js and JavaScript backend whereas the frontend allows the use of standard technologies, such as CSS3, HTML5, and JavaScript. Lisk uses LSK coin as a currency on the blockchain. Another derivative of Lisk is Rise, which is a Lisk-based decentralized application and digital currency platform. It has more focus on the security of the system.

UNIT - II

The practical implementations of cryptographic algorithms so that you can experience the cryptographic functions practically. For this, the **OpenSSL** command line is used.

Before starting the theoretical foundations, the installation of OpenSSL is discussed in the following section so that you can do some practical work as you read through the theoretical material.

On Ubuntu Linux distribution, OpenSSL is usually already available; however, it can be installed using the following commands:

$ sudo apt-get install openssl

Introduction to Cryptography

Cryptography is the science of making information secure in the presence of adversaries. It provides a means of secure communication in the presence of adversaries with assumed limitless resources. Ciphers are used to encrypt data so that if intercepted by an adversary, the data is meaningless to them without decryption, which requires the secret key.

Cryptography is generally used to provide a confidentiality service. On its own, it cannot be considered a complete solution but serve as a crucial building block within a larger security system to address a security problem.

Cryptography provides various security services, such as **Confidentiality, Integrity, Authentication**, (Entity Authentication and Data origin authentication) and **non-repudiation**. Additionally, accountability is also required in various security systems.

Before discussing cryptography further, there are some mathematical terms and concepts that need to be explained first in order

Mathematics

As the subject of cryptography is based on mathematics, this section will introduce some basic concepts that will help you understand the concepts later in the chapter.

Set: A set is a collection of distinct objects, for example, $X = \{1, 2, 3, 4, 5\}$.

Group

A group is a commutative set with one operation that combines two elements of the set. The group operation is closed and associated with an identity element defined. Additionally, each element in the set has an inverse. Closure (closed) means that if, for example, elements A and B are in the set, then the resultant element after performing operation on the elements is also in the set. Associative means that the grouping of elements does not affect the result of the operation.

Field

A field is a set that contains both additive and multiplicative groups. More precisely, all elements in the set form an additive and multiplicative group. It satisfies specific axioms for addition and multiplication. For all group operations, the distributive law is also applied. The law dictates that the same sum or product will be produced even if any terms or factors are reordered.

A finite field

A finite field is a field with a finite set of elements. Also known as Galois fields, these structures are of particular importance in cryptography as they can be used to produce accurate and error-free results of arithmetic operations. For example, prime finite fields are used in elliptic curve cryptography to construct discrete logarithm problem.

Order : This is the number of elements in a field. It is also known as the cardinality of the field.

Prime fields

This is a finite field with a prime number of elements. It has specific rules for addition and multiplication, and each nonzero element in the field has an inverse. Addition and multiplication operations are performed modulo p.

Ring

If more than one operation can be defined over an abelian group, that group becomes a ring. There are also certain properties that need to be satisfied. A ring must have closure and associative and distributive properties.

A cyclic group

A cyclic group is a type of group that can be generated by a single element called the group generator. In other words, if the group operation is repeatedly applied to a particular element in the group, then all elements in the group can be generated.

An abelian group

An abelian group is formed when the operation on the elements of a set is commutative. Commutative law basically means that changing the order of the elements does not affect the result of the operation, for example, $A X B = B X A$.

Modular arithmetic

Also known as clock arithmetic, numbers in modular arithmetic wrap around when they reach a certain fixed number. This fixed number is a positive number called modulus and all operations are performed with regard to this fixed number. In an analogy to a clock, there are number from 1 to 12. When it reaches 12, the number 1 starts again. In other words, this arithmetic deals with the remainders after the division operation. For example, 50 mod 11 is 6 because 50 / 11 leaves a remainder of 6.

This completes a basic introduction to some mathematical concepts; in the next section, you will be introduced to cryptography.

Cryptography

As discussed earlier, cryptography provides various security services, and these security services are discussed here.

Confidentiality: Confidentiality is the assurance that information is only available to authorized entities.

Integrity: Integrity is the assurance that information is modifiable onlybyauthorized entities.

Authentication

Authentication provides assurance about the identity of an entity or the validity of a message. There are two types of authentications, discussed here.

Entity authentication

Entity authentication is the assurance that an entity is currently involved and active in a communication session. Traditionally, users are issued a username and password, which are used to gain access to the platforms they are using. This is called single factor authentication as there is only one factor, namely *something you know*, that is, the password and username. This type of authentication is not very secure due to various reasons, such as password leakage; therefore, additional factors are now commonly used to provide better security. The use of additional techniques for user identification is known as multifactor authentication or two-factor authentication if only two methods are used. If more than two factors are used for authentication, that is called multifactor authentication.

Various factors are described here:

1. The first factor is something you have, such as a hardware token or smart card. In this case, a user can use a hardware token in addition to login credentials to gain access to a system. This provides protection by requiring two factors of authentication. A user who has access to the hardware token and knows the log-on credentials will be able to access the system. Both factors should be available in order to gain access to the system, thus making this method a two-factor authentication mechanism.
2. The second factor is something you are, which uses biometric features in order to identify the user. In this method, a user uses fingerprint, retina, iris, or hand geometry to provide an additional factor for authentication. This way, it can be ensured that a user was indeed present during the authentication mechanism as biometric features are unique to an individual. However, careful implementation is required in order to ensure a high level of security as some research has suggested that biometric systems can be circumvented in certain scenarios.

Data origin authentication

Also known as message authentication, this is an assurance that the source of information is verified. Data origin authentication implies data integrity because if a source is corroborated, then data must not have been altered. Various methods, such as **Message Authentication Codes** (**MACs**) and digital signatures are most commonly used.

Non-repudiation

Non-repudiation is the assurance that an entity cannot deny a previous commitment or action by providing unforgeable evidence. It is a security service that provides unforgeable evidence that a particular action has occurred. This property is very necessary in disputable situations whereby an entity has denied actions performed, for example, placing an order on an e-commerce system.

The non-repudiation protocol usually runs in a communication network and is used to provide evidence that an action has been taken by an entity (originator or recipient) on the network. In this context, there are two communication models that can be used to transfer messages from originator *A* to recipient *B*:

1. Message is sent directly from originator *A* to recipient *B*.
2. Message is sent to a deliveryagent fromoriginator *A*, which then delivers the message to recipient *B*.

The main requirements of a non-repudiation protocol are fairness, effectiveness, and timeliness. In many scenarios, there are multiple participants involved in a transaction as opposed to only two parties. For example, in electronic trading systems, there can be many entities, such as clearing agents, brokers, and traders that can be involved in a single transaction. In this case, two-party non-repudiation protocols are not appropriate. To address this problem **Multi-party nonrepudiation protocols** (**MPNR**) has been developed.

Accountability

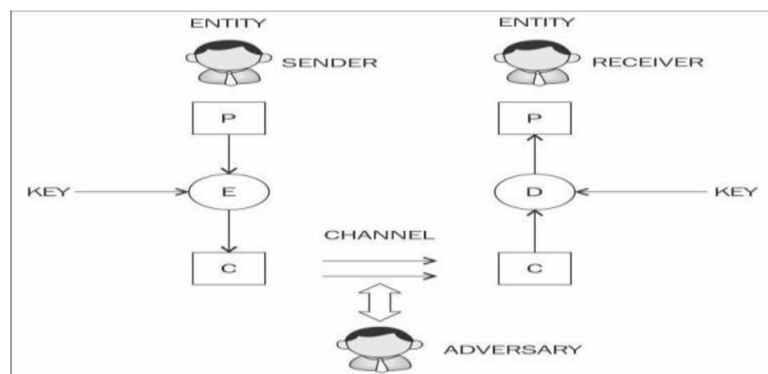
Accountability is the assurance that actions affecting security can be traced to the responsible party. This is usually provided by logging and audit mechanisms in systems where a detailed audit is required due to the nature of the business, for example, in electronic trading systems. Detailed logs are vital to trace an entity's actions, for example, when a trade is placed in an audit record with the date and time stamp and the entity's identity is generated and saved in the log file. This log file can optionally be encrypted and can be part of the database or a standalone ASCII text log file on a system.

Cryptographic primitives

Cryptographic primitives are the basic building blocks of a security protocol or system. In the following section, you are introduced to cryptographic algorithms that are essential for the building of secure protocols and systems. A **security protocol** is a set of steps taken in order to achieve required security goals by utilizing appropriate security mechanisms.

Various types of security protocols are in use, such as **authentication protocols**, **non-repudiation protocols**, and **key management protocols**.

A generic cryptography model is shown in the following diagram:



A model showing the generic encryption and decryption model

In the preceding diagram, **P**, **E**, **C**, and **D** represents Plain text, Encryption, Cipher text, and Decryption, respectively. Also, based on the model shown earlier, it is worth explaining various concepts such as entity, sender, receiver, adversary, key, and a channel.

- **Entity**: It is either a person or a system that sends, receives, or performs operations on data
- **Sender**: Sender is an entity that transmits the data
- **Receiver**: Receiver is an entity that takes delivery of the data
- **Adversary**: This is an entity that tries to circumvent the security service
- **Key**: A key is some data that is used to encrypt or decrypt data
- **Channel**: Channel provides a medium of communication between entities

Cryptography is mainly divided into two categories, namely symmetric and asymmetric cryptography.

Symmetric cryptography

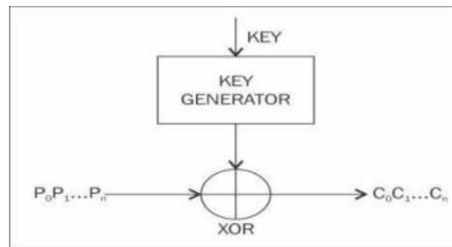
Symmetric cryptography refers to a type of cryptography whereby the key that is used to encrypt the data is the same for decrypting the data, and thus it is also known as a shared key cryptography. The key must be established or agreed on before the data exchange between the communicating parties. This is the reason it is also called **secret key cryptography**.

There are two types of symmetric ciphers, stream ciphers and block ciphers. **Data Encryption Standard** (**DES**) and **Advanced Encryption Standard** (**AES**) are common examples of block ciphers, whereas RC4 and A5 are commonly used stream ciphers.

Stream ciphers

These ciphers are encryption algorithms that apply encryption algorithms on a bit-by-bit basis to plain text using a key stream. There are two types of stream ciphers: **synchronous and asynchronous.**
Synchronous stream ciphers are ones where key stream is dependent only on the key, whereas **Asynchronous stream** ciphers have a key stream that is also dependent on the encrypted data.

In stream ciphers, encryption and decryption are basically the same function because they are simple modulo 2 additions or XOR operation. The key requirement in stream ciphers is the security and randomness of key streams. Various techniques have been developed to generate random numbers, and it's vital that all key generators be cryptographically secure.



Operation of a stream cipher

Block ciphers

These are encryption algorithms that break up a text to be encrypted (plain text) into blocks of fixed length and apply encryption block by block. Block ciphers are usually built using a design strategy known as Fiestel cipher. Recent block ciphers, such as AES (Rijndael) have been built using a combination of substitution and permutation called **substitution-permutation network** (**SPN**).

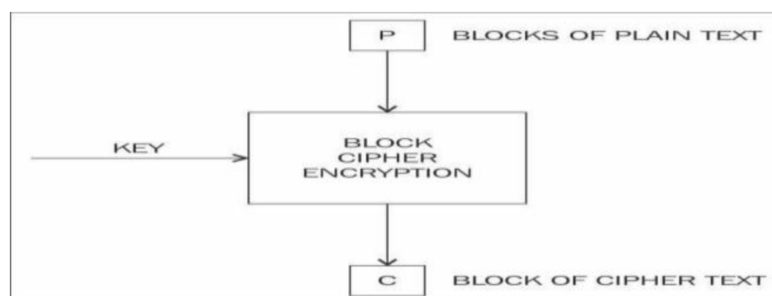
Fiestel ciphers are based on the Fiestel network, which is a structure developed by *Horst Fiestel*. This structure is based on the idea of combining multiple rounds of repeated operations to achieve desirable cryptographic properties knows as confusion and diffusion. Fiestel networks operate by dividing data into two blocks (left and right) and process these blocks via keyed round functions.

Confusion makes the relationship between the encrypted text and plaintext complex. This is achieved by substitution in practice. For example, 'A' in plain text is replaced by 'X' in encrypted text. In modern cryptographic algorithms, substitution is performed using lookup tables called S-boxes. The diffusion property spreads the plain text statistically over the encrypted data, which ensures that even

if a single bit is changed in the input text, it results in changing at least half (on average) of the bits in the cipher text. Confusion is required to make finding the encryption key very difficult even if many encrypted and decrypted data pairs are created using the same key. In practice, this is achieved by transposition or permutation.

A key advantage of using Fiestel cipher is that encryption and decryption operations are almost identical and only require a reversal of the encryption process in order to achieve decryption. DES is a prime example of Fiestel-based ciphers.

Simplified operation of a block cipher



Various modes of operation for block ciphers are **Electronic Code Book** (**ECB**), **Cipher block chaining** (**CBC**), **Output Feedback Mode** (**OFB**), or **Counter mode** (**CTR**). These modes are used to specify the way in which an encryption function would be applied to the plain text. These modes will be explained later in this section, but the first four categories of block cipher encryption modes are introduced here.

Block encryption mode

In this mode, plaintext is divided into blocks of fixed length depending on the type of cipher used and then the encryption function is applied on each block.

Key stream generation modes

In this mode, the encryption function generates a keystream that is then XORed with the plaintext stream in order to achieve encryption.

Message authentication modes

In this mode, a message authentication code is computed as a result of an encryption function. MAC is basically a cryptographic checksum that provides an integrity service. The most common method to generate MAC using block ciphers is CBC-MAC, where some part of the last block of the chain is used as a MAC.

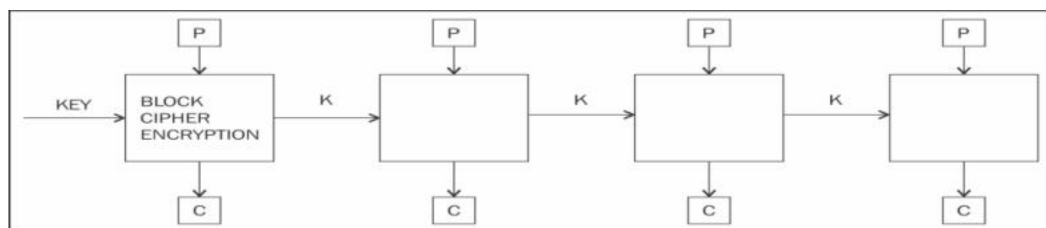
Cryptographic hashes

Hash functions are basically used to compress a message to a fixed length digest. In this mode, block ciphers are used as a compression function to produce a hash of plain text.

The most common block encryption modes are discussed briefly.

Electronic code book

This is a basic mode of operation in which the encrypted data is produced as a result of applying the encryption algorithm one by one separately to each block of plain text. This is the simplest mode but should not be used in practice as it is insecure and can reveal information:

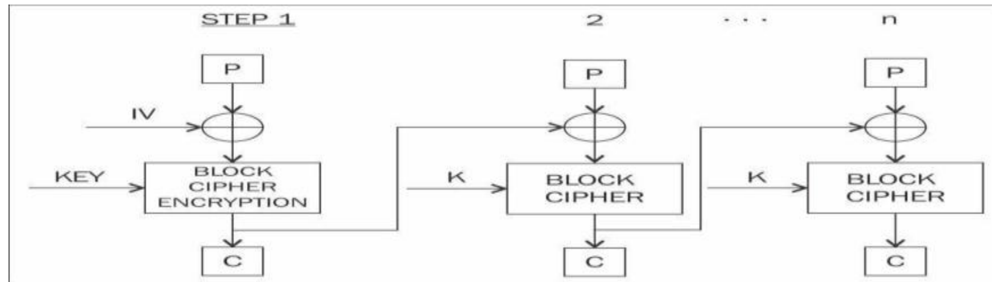


Electronic code book mode for block ciphers

Cipherblock chaining

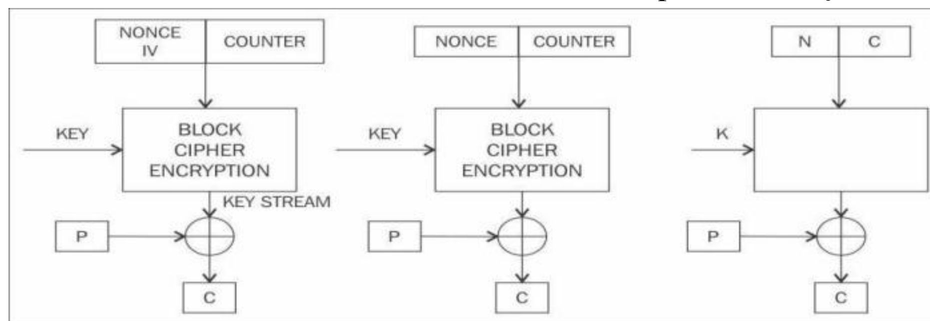
In this mode, each block of plain text is XORed with the previous encrypted block. The CBC mode uses initialization vector IV to encrypt the first block. It is recommended that IV be randomlychosen:

Cipher block chaining mode



Counter mode

The CTR mode effectively uses a block cipher as a stream cipher. In this case, a unique nonce is supplied that is concatenated with the counter value in order to produce a **key stream**



Counter mode

There are other modes, such as **Cipher Feedback mode** (**CFB**), **Galois Counter mode** (**GCM**), and Output Feedback mode, which are also used in various scenarios.

Key size	Number of rounds required
128-bit	10 rounds
192-bit	12 rounds
256-bit	14 rounds

In the following section, you will be introduced to the design and mechanism of a currently dominant block cipher know as AES. First, some history will be presented with regard to Data Encryption Standard (DES) that led to the development of a new AES standard.

Data Encryption Standard (DES)

DES was introduced by the US **National Institute of Standards and Technology** (**NIST**) as a standard algorithm for encryption and was in main use during 1980s and 1990s, but it has been not proven to be very resistant against brute force attacks, due to advances in technology and cryptography research. Especially in July 1998, **Electronic Frontier Foundation** (**EFF**) broke DES using a special purpose machine. DES uses a key of only 56 bits, which has raised some concerns. This problem was addressed with the introduction of **Triple DES** (**3DES**), which proposed the usage of a 168-bit key using three 56-bit keys and the same number of executions of the DES algorithm, thus making brute force attacks almost impossible. But other limitations, such as slow performance and 64-bit block size, are not desirable.

Advanced Encryption Standard (AES)

In 2001, after an open competition, an encryption algorithm named Rijndael that was invented by cryptographers *Joan Daemen* and *Vincent Rijmen* was standardized as AES with minor modifications by NIST in 2001. So far, no attack has been found against AES that is better than the brute force method. Original Rijndael allows different key and block sizes of 128-bit, 192-bit, and 256-bits, but in the AES standard, only a 128-bit block size is allowed. However, key sizes of 128-bit, 192-bit, and 256-bit are allowed.

AES steps

During the AES Algorithm processing, a 4 by 4 array of bytes knows as *state* is modified using multiple rounds. Full encryption requires 10 to 14 rounds depending on the size of the key. The following table shows the key sizes and the required number of rounds:

Once the state is initialized with the input to the cipher, four operations are performed in four stages in order to encrypt the input. These stages are Add Round Key, Sub Bytes, Shift Rows, and Mix Columns:

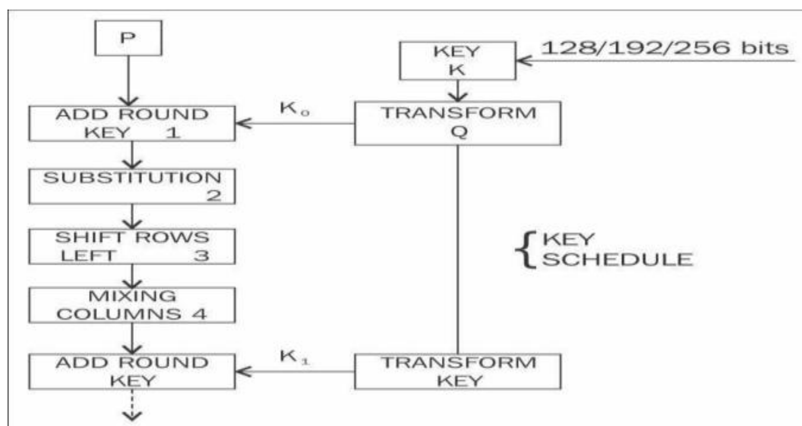
1. In the Add Round Keystep, the state array is XORed with a sub key, which is derived fromthe master key.
2. This is the substitution step where a lookup table (S-box) is used to replace all bytes of the state array.
3. This step is used to shift each row except the first one in the state array to the left in a cyclic and incremental manner.
4. Finally, all bytes are mixed in this step in a linear fashion column-wise.

The preceding steps describe one round of AES. In the final round (either 10, 12, or 14 depending on the key size), stage 4 is replaced with Add round key to ensure that the first three steps cannot be simply inverted back

AES block diagram, showing 1st round, in last round mixing step is not performed

Various cryptocurrency wallets use AES encryption to encrypt locally stored data. Especially in

bitcoin wallet, AES 256 in the CBC mode is used.



An OpenSSL example of how to encrypt and decrypt using AES

:~/Crypt$ openssl enc -aes-256-cbc -in message.txt -out message.bin enter aes-256-cbc encryption password:
Verifying - enter aes-256-cbc encryption password:
:~/Crypt$ ls -ltr total 12
-rw-rw-r-- 1 drequinox drequinox 14 Sep 21 05:54 message.txt
-rw-rw-r-- 1 drequinox drequinox 32 Sep 21 05:57 message.bin
:~/Crypt$ cat message.bin

```
Salted__w░░░s_ÿ░░h~?░░░░░░░░░░░░░░░░:~/Crypt$
:~/Crypt$
```

Note that message.bin is a binary file; sometimes, it is desirable to encode this binary file into a text format for compatibility/interoperability reasons. The following command can be used to do that:

:~/Crypt$ openssl enc -base64 -in message.bin -out message.b64
:~/Crypt$ ls -ltr
-rw-rw-r-- 1 drequinox drequinox 14 Sep 21 05:54 message.txt
-rw-rw-r-- 1 drequinox drequinox 32 Sep 21 05:57 message.bin
-rw-rw-r-- 1 drequinox drequinox 45 Sep 21 06:00 message.b64

:~/Crypt$ cat message.b64 U2FsdGVkX193uByIcwZf0Z7J1at+4L+Fj8/uzeDAtJE=
:~/Crypt$

In order to decrypt an AES-encrypted file, the following commands can be used. An example of message.bin from a previous example is taken:

:~/Crypt$ openssl enc -d -aes-256-cbc -in message.bin -out message.dec enter aes-256-cbc decryption password:

:~/Crypt$ ls -ltr
-rw-rw-r-- 1 drequinox drequinox 14 Sep 21 05:54 message.txt
-rw-rw-r-- 1 drequinox drequinox 32 Sep 21 05:57 message.bin
-rw-rw-r-- 1 drequinox drequinox 45 Sep 21 06:00 message.b64
-rw-rw-r-- 1 drequinox drequinox 14 Sep 21 06:06 message.dec
:~/Crypt$ cat message.dec datatoencrypt
:~/Crypt$

In order to decode from base64, the following commands are used. Take the message.b64file from the previous example:

:~/Crypt$ openssl enc -d -base64 -in message.b64 -out message.ptx
:~/Crypt$ ls –ltr

-rw-rw-r-- 1 drequinox drequinox 14 Sep 21 05:54 message.txt
-rw-rw-r-- 1 drequinox drequinox 32 Sep 21 05:57 message.bin
-rw-rw-r-- 1 drequinox drequinox 45 Sep 21 06:00 message.b64
-rw-rw-r-- 1 drequinox drequinox 14 Sep 21 06:06 message.dec
-rw-rw-r-- 1 drequinox drequinox 32 Sep 21 06:16 message.ptx

```
:~/Crypt$ cat message.ptx
Salted__w  s_ÿ  h~?  :~/Crypt$
```

There are many types of ciphers that are supported in OpenSSL; you can explore these options based on the examples provided earlier. A list of supported cipher types is shown in the following screen shot:

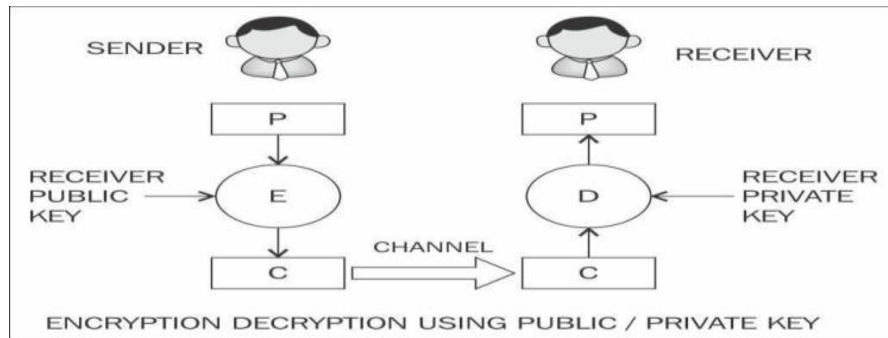
```
drequinox@drequinox-OP7010: ~/Crypt
Cipher Types
-aes-128-cbc               -aes-128-cbc-hmac-sha1     -aes-128-cbc-hmac-sha256
-aes-128-ccm               -aes-128-cfb               -aes-128-cfb1
-aes-128-cfb8              -aes-128-ctr               -aes-128-ecb
-aes-128-gcm               -aes-128-ofb               -aes-128-xts
-aes-192-cbc               -aes-192-ccm               -aes-192-cfb
-aes-192-cfb1              -aes-192-cfb8              -aes-192-ctr
-aes-192-ecb               -aes-192-gcm               -aes-192-ofb
-aes-256-cbc               -aes-256-cbc-hmac-sha1     -aes-256-cbc-hmac-sha256
-aes-256-ccm               -aes-256-cfb               -aes-256-cfb1
-aes-256-cfb8              -aes-256-ctr               -aes-256-ecb
-aes-256-gcm               -aes-256-ofb               -aes-256-xts
-aes128                    -aes192                    -aes256
-bf                        -bf-cbc                    -bf-cfb
-bf-ecb                    -bf-ofb                    -blowfish
-camellia-128-cbc          -camellia-128-cfb          -camellia-128-cfb1
-camellia-128-cfb8         -camellia-128-ecb          -camellia-128-ofb
-camellia-192-cbc          -camellia-192-cfb          -camellia-192-cfb1
-camellia-192-cfb8         -camellia-192-ecb          -camellia-192-ofb
-camellia-256-cbc          -camellia-256-cfb          -camellia-256-cfb1
-camellia-256-cfb8         -camellia-256-ecb          -camellia-256-ofb
-camellia128               -camellia192               -camellia256
-cast                      -cast-cbc                  -cast5-cbc
-cast5-cfb                 -cast5-ecb                 -cast5-ofb
-des                       -des-cbc                   -des-cfb
-des-cfb1                  -des-cfb8                  -des-ecb
-des-ede                   -des-ede-cbc               -des-ede-cfb
-des-ede-ofb               -des-ede3                  -des-ede3-cbc
-des-ede3-cfb              -des-ede3-cfb1             -des-ede3-cfb8
-des-ede3-ofb              -des-ofb                   -des3
-desx                      -desx-cbc                  -id-aes128-CCM
-id-aes128-GCM             -id-aes128-wrap            -id-aes192-CCM
-id-aes192-GCM             -id-aes192-wrap            -id-aes256-CCM
-id-aes256-GCM             -id-aes256-wrap            -id-smime-alg-CMS3DESwrap
-rc2                       -rc2-40-cbc                -rc2-64-cbc
-rc2-cbc                   -rc2-cfb                   -rc2-ecb
-rc2-ofb                   -rc4                       -rc4-40
-rc4-hmac-md5              -seed                      -seed-cbc
-seed-cfb                  -seed-ecb                  -seed-ofb
```

Screenshot displaying rich library options available in OpenSSL.

Asymmetric cryptography

Asymmetric cryptography refers to a type of cryptography whereby the key that is used to encrypt the data is different from the key that is used to decrypt the data. Also known as public key cryptography, it uses public and private keys in order to encrypt and decrypt data, respectively. Various asymmetric cryptography schemes are in use, such as RSA, DSA, and El-Gammal.

An overview of public key cryptography is shown in the following diagram:



Encryption decryption using public/private key

The diagram explains how a sender encrypts the data using a recipient's public key and is then transmitted over the network to the receiver. Once it reaches the receiver, it can be decrypted using the receiver's private key. This way, the private key remains on the receiver's side and there is no need to share keys in order to perform encryption and decryption, which is the case with symmetric encryption.

Another diagram shows how public key cryptography can be used to verify the integrity of the received message by the receiver. In this model, the sender signs the data using their private key and transmits the message across to the receiver. Once the message is received on the receiver's side, it can be verified for its integrity by the sender's public key. Note that there is no encryption being performed in this model. This model is only used for message authentication and validation purposes:

Model of a public key cryptography signature scheme

Security mechanisms offered by public key crypto system include key establishment, digital signatures, identification, encryption, and decryption.

Key establishment mechanisms are concerned with the design of protocols that allow setting up of keys over an insecure channel. Non-repudiation service, a very desirable property in many scenarios, can be provided using digital signatures. Sometimes, it is important to not only authenticate a user, but to also identify the entity involved in a transaction; this can also be achieved by a combination of digital signatures and challenge-response protocols. Finally, the encryption mechanism to provide confidentiality can also be realized using public key cryptosystems, such as RSA, ECC, or El- Gammal.

Public key algorithms are slower in computation as compared to symmetric key algorithms. Therefore, they are not commonly used in the encryption of large files or the actual data that needs encryption. They are usually used to exchange keys for symmetric algorithms and once the keys are established securely, symmetric key algorithms can be used to encrypt the data.

Integer factorization

These schemes are based on the fact that large integers are very hard to factor. RSA is the prime example of this type of algorithm.

Discrete logarithm

This is based on a problem in modular arithmetic that it is easy to calculate the result of modulo function but it is computationally infeasible to find the exponent of the generator. In other words, it is extremely difficult to find the input from the result. This is a one-way function.

For example, consider the following equation:

$$3^2 \bmod 10 = 9$$

Now given 9 finding 2, the exponent of the generator 3 is very hard. This hard problem is commonly used in **Diffie-Hellman** key exchange and digital signature algorithms.

Elliptic curves

This is based on the discrete logarithm problem discussed earlier, but in the context of elliptic curves. Elliptic curve is an algebraic cubic curve over a field, which can be defined by an equation shown here. The curve is non-singular, which means that it has no cusps or self-intersections. It has two variables *a, b*, along with a point of infinity.

$$y^2 = x^3 + ax + b$$

Here, *a*, *b* are integers that can have various values and are elements of the field on which the elliptic curve is defined. Elliptic curves can be defined over reals, rational numbers, complex numbers, or finite fields. For cryptographic purposes, elliptic curve over prime finite fields is used instead of real numbers. Additionally, the prime should be greater than 3. Different curves can be generated by varying the value of *a*, *b*.

Mostly prominently used cryptosystems based on elliptic curves are **Elliptic Curve Digital Signatures Algorithm** (**ECDSA**) and **Elliptic Curve Diffie-Hellman** (**ECDH**) keyexchange.

Public and private keys

In order to understand public key cryptography, the first concept that needs to be looked at is the idea of public and private keys.

A private key, as the names suggests, is basically a randomly generated number that is kept secret and held privately by the users. Private key needs to be protected and no unauthorized access should be granted to that key; otherwise, the whole scheme of public key cryptography will be jeopardized as this is the key that is used to decrypt messages. Private keys can be of various lengths depending upon the type and class of algorithms used. For example, in RSA, typically, a key of 1024-bit or 2048-bits is used. 1024-bit key size is no longer considered secure and at least 2048 bit is recommended to be used in practice.

A public key is the public part of the private-public key pair. A public key is available publicly and published by the private key owner. Anyone who would then like to send the publisher of the public key an encrypted message can do so by encrypting the message using the published public key and sending it to the holder of the private key. No one else would be able to decrypt the message because the corresponding private key is held securely by the intended recipient. Once the public key encrypted message is received, the recipient can decrypt the message using the private key. There are a few concerns regarding public keys, such as authenticity and identification of the publisher of the public keys.

RSA

A description of RSA is discussed here. RSA was invented in 1977 by *Ron Rivest*, *Adi Shamir*, and *Leonard Adelman*, hence the name RSA. This is based on the integer factorization problem, where the multiplication of two large prime numbers is easy but difficult to factor it back to the two original numbers.

The crux of the work in the RSA algorithm is during the key generation process. An RSA key pair is generated by performing the steps described here.

Modulus generation:

- Select p and q very large primes
- Multiply p and q , $n=p.q$ to generate modulus n

Generate co-prime:

- Assume a number called e.
- It should satisfy certain conditions, that is, it should be greater than 1 and less than $(p-1)\ (q-1)$. In other words, e must be such a number that no number other than 1 can be divided into e and $(p-1)\ (q-1)$. This is called co-prime, that is, e is the co-prime of $(p-1)(q-1)$.

Generate public key:

- Modulus generated in step 1 and e generated in step 2 is pair that, together, is a public key. This part is the public part that can be shared with anyone; however, *p* and *q* need to be kept secret.

Generate private key:

- Private key called d here and is calculated from *p*, *q* and *e*. Private key is basically the inverse of e modulo *(p-1)(q-1)*. In the equation form, it is this:

$$ed = 1 \bmod (p-1)(q-1)$$

Usually, an extended Euclidean algorithm is used to calculate *d*; this algorithm takes *p*, *q* and *e* and calculates *d*. The key idea in this scheme is that anyone who knows *p* and *q* can calculate private key *d* easily, by applying the extended Euclidean algorithm, but someone who doesn't know the value of *p* and *q* cannot generate *d*. This also implies that *p* and *q* should be large enough for the modulus *n* to become very difficult (computationally infeasible) to factor.

Encryption and decryption using RSA

RSA uses the following equation to produce cipher text:

$$C = P^e \bmod n$$

This means that plain text *P* is raised to *e* number of times and then reduced to modulo *n*.

Decryption in RSA is given by the following equation:

$$P = C^d \bmod n$$

This means that the receiver who has a public key pair (*n*, *e*) can decipher the data by raising *C* to the value of the private key *d* and reducing to modulo *n*.

Eliptic curve cryptography (ECC)

ECC is based on the discrete logarithm problem that is based on elliptic curves over finite fields (Galois fields). The main benefit of ECC over other types of public key algorithms is that it needs a smaller key size while providing the same level of security as, for example, RSA. Two notable schemes that originate from ECC are **Elliptic Curve Diffie-Hellman** (**ECDH**) for key exchange and **Elliptic Curve Digital Signature Algorithm** (**ECDSA**) for digital signatures. It can also be used for encryption but is not usually used for this purpose in practice; instead, key exchange and digital signatures are more commonly used. As ECC needs less space to operate, it is becoming very popular on embedded platforms or in systems where storage resources are limited. As a comparison, the same level of security can be achieved in ECC by only using 256-bit operands as compared to 3072-bits in RSA.

Mathematics behind ECC

In order to understand ECC, a basic introduction to the underlying mathematics is necessary. Elliptic curve is basically a type of polynomial equation known as weierstrass equation that generates a curve over a finite field. The most commonly used field is where all arithmetic operations are performed modulo *a* prime *p*. Elliptic curve groups consist of points on the curve over a finite field.

An elliptic curve can be defined as an equation here:

$$y^2 = x^3 + Ax + B \bmod p$$

Here, *A* and *B* belong to a finite field *Zp* or *FP* (prime finite field) along with a special value called point of infinity. Point of infinity ∞ is used to provide identity operations for points on the curve.

Furthermore, a condition also needs to be met that ensures that the equation mentioned earlier has no repeated roots. This means that the curve is non-singular.

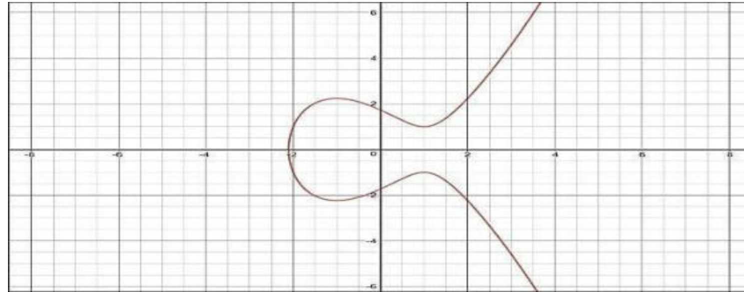
The condition is described here in the equation, which is a standard requirement that needs to be met. More precisely, this ensures that the curve is nonsingular:

$$4a^3 + 27b^2 \neq 0 \bmod p$$

A real number representation of elliptic curve can be visualized as shown in the following graph. This is a graph of equation over real numbers:

$$y^2 = x^3 + ax + b$$

The actual curves used in elliptic curve cryptography are over finite prime fields, but here, theyare



shown over real number as it becomes easier to visualize the operations when graphed over *R*:

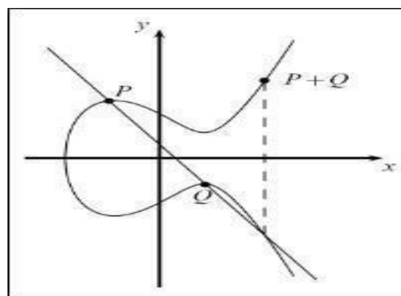
Elliptic curve over reals, a = -3 and b = 3

In order to construct the discrete logarithm problem based on elliptic curves, a large enough cyclic group is required. First, the group elements are identified as a set of points that satisfy the earlier equation. After this, group operations need to be defined on these points.

Group operations on elliptic curves are point addition and point doubling. Point addition is a process where two different points are added and point doubling means that the same point is added to itself. Both of these operations can be visualized as shown in the following diagrams.

Point addition

Point addition is shown in the following diagram. This is a geometric representation of point addition on elliptic curves. In this method, a line is drawn through the curve that intersects the



curve at two points shown below *P* and *Q*, which yields a third point between the curve and the line. This point is mirrored as *P+Q*, which represent the result of addition as *R*. This is shown as *P+Q* in the following diagram:

Point addition visualized over R

Group operation denoted by sign + for addition yields the following equation: P + Q = R

In this case, two points are added in order to compute the coordinates of the third point on the curve:

$$P + Q = R$$

More precisely, this means that coordinates are added as shown in the following equation:

$$(x1, y1) + (x2, y2) = (x3, y3)$$

The equation of point addition is as follows:

$$x_3 = s^2 - x_1 - x^2 \bmod p$$

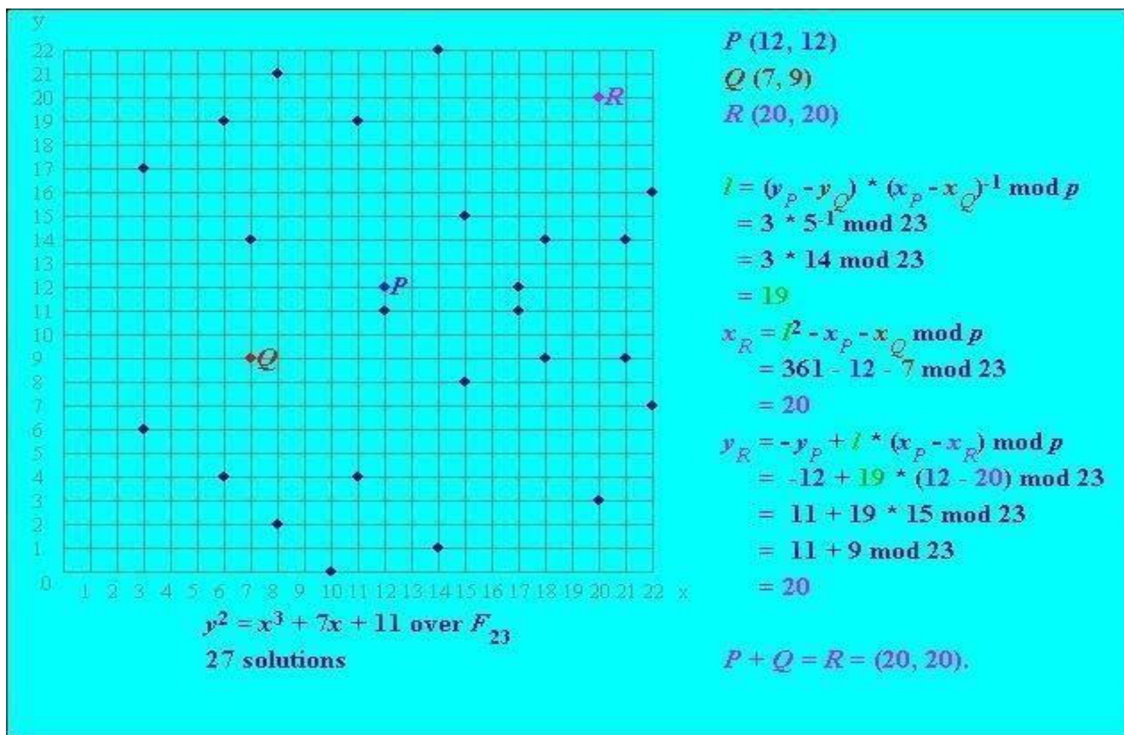
$$y_3 = s(x_1 - x_3) - y_1 \bmod p$$

Here, this is the result:

$$S = \frac{(y_2 - y_1)}{(x_2 - x_1)} \bmod p$$

S in the preceding equation depicts the line going through *P* and *Q*.

An example of point addition shown here is produced using Certicom's online calculator. This example shows the addition and solutions for the equation over finite field *F23*. This is in contrast to the example shown earlier, which is over real numbers and only shows the curve but no solutions to the equation:



Example of point addition using Certicom's online calculator tool

In the example, the graph on the left-hand side shows the points that satisfythe equation shown here:

$y^2 = x^3 + 7x + 11$

There are 27 solutions to the equation shown earlier over a finite field F_{23}. *P* and *Q* are chosen to be added to produce the point *R*. Calculations are shown on the right-hand side, which calculates the third point *R*. Note that here, *l* is used to depict the line going through *P* and *Q*.

As an example to show how the equation is satisfied by the points shown in the graph, a point (*x, y*) is picked up where *x = 3* and *y = 6*.

Using these values in the equation shows that the equation is satisfied indeed. This is shown as follows:

$$y^2 mod\ 23 = x^3 + 7x + 11\ mod\ 23$$

$$6^2 mod\ 23 = 3^3 + 7(3) + 11\ mod\ 23$$

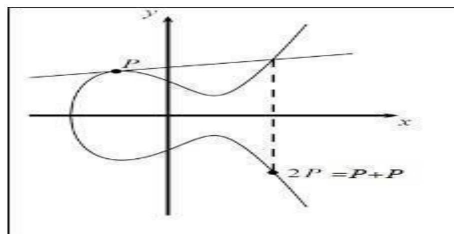
$$36\ mod\ 23 = 59\ mod\ 23$$

$$13 = 13$$

The next section will introduce the concept of point doubling, which is another operation that can be performed on elliptic curves.

Point doubling

The other group operation on elliptic curves is called point doubling and is described in the following diagram. This is a process where *P* is added into itself. In this method, a tangent line is drawn through the curve, as shown in the following graph. The second point is obtained, which is at the intersection of the tangent line drawn and the curve. This point is then mirrored to yield the result, which is shown as *2P = P + P*:



Graph representing point doubling over real numbers

In case of point doubling, the equation becomes as follows:

$$x_3 = s^2 - x_1 - x^2 \bmod p$$

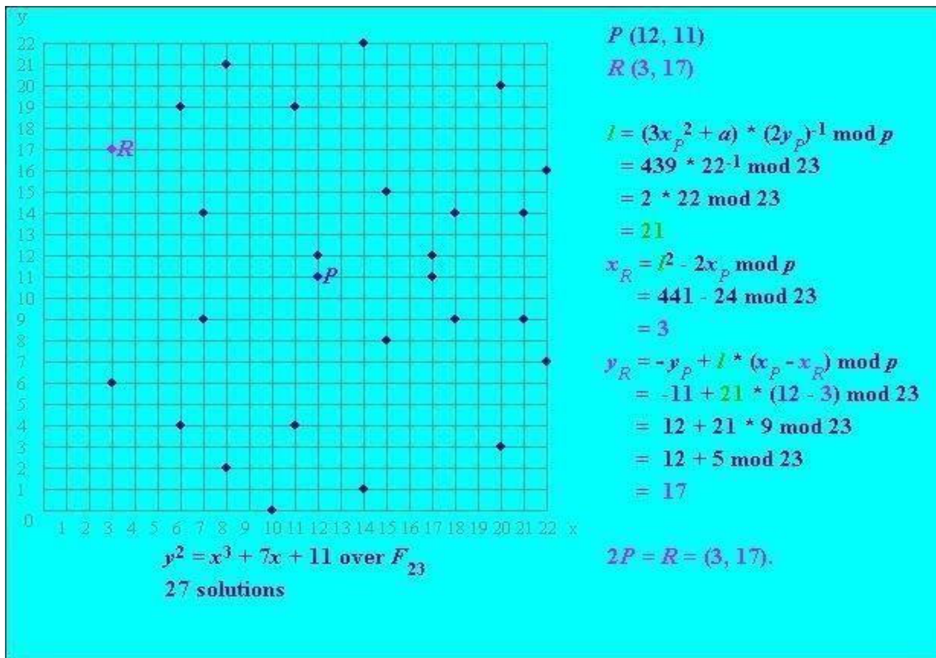
$$y_3 = s(x_1 - x_3) - y_1 \bmod p$$

$$S = \frac{(y_2 - y_1)}{(x_2 - x_1)} \bmod p$$

Here, S is the slope of tangent (tangent line) going through P. It is the line on top shown in the preceding figure. In the preceding example, the curve is plotted over reals as a simple example and no solution to the equation is shown.

An example is shown here, which shows the solutions and point doubling of elliptic curve over finite field F_{23}. The graph on the left-hand side shows the points that satisfy the equation:

$$y^2 = x^3 + 7x + 11$$



Example of point doubling using certicom's online calculator tool

As shown earlier, on the right-hand side, a calculation is shown that finds the R after P is added into itself (point doubling). There is no Q as here, the same point P is used for doubling. Note that in the calculation, l is used to depict the tangent line going through P.

In the next section, an introduction to the discrete logarithm problemwill be presented.

Discrete logarithm problem

The discrete logarithm problem in ECC is based on the idea that under certain conditions, all points on an elliptic curve form a cyclic group. On an elliptic curve, the public key is a random multiple of the generator point, whereas the private key is a randomly chosen integer used to generate the multiple. In other words, a private key is a randomly chosen integer, whereas the public key is a point on the curve. The discrete logarithm problem is used to find the private key (an integer) where that integer falls within all points on the elliptic curve. An upcoming equation shows this precisely.
Consider an elliptic curve *E*, with two elements *P* and *T*. The discrete logarithmic problem is to find the integer d, where $1 <= d <= \#E$, such that:

$$P + P + . . . +P = d P = T$$

Here, *T* is the public key (point on the curve) and d is the private key. In other words, public key is a random multiple of generator, whereas the private key is the integer that is used to generate the multiple. *#E* represents the order of the elliptic curve, which basically means the number of points that are present in the cyclic group of the elliptic curve. A cyclic group is formed by a combination of points on the elliptic curve and point at infinity.

A key pair is linked with specific domain parameters of an elliptic curve. Domain parameters include a field size, field representation, two elements from the field *a* and *b*, two field elements *Xg* and *Yg*, order n of point *G* that is calculated as $G=(Xg, Yg)$ and the co-factor $h = \#E(Fq)/n$. A practical example using OpenSSLwill be described later in this section.

There are various parameters that are recommended and standardized to use as curves with ECC. You are shown an example of SECP256K1 specifications here. This is the specification that has been used in bitcoin:

The elliptic curve domain parameters over $\mathbb{F}_p$ associated with a Koblitz curve secp256k1 are specified by the sextuple $T = (p, a, b, G, n, h)$ where the finite field $\mathbb{F}_p$ is defined by:

p = FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE FFFFFC2F
$= 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$

The curve $E: y^2 = x^3 + ax + b$ over $\mathbb{F}_p$ is defined by:

a = 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

b = 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000007

The base point G in compressed form is:

G = 02 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798

and in uncompressed form is:

G = 04 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798 483ADA77 26A3C465 5DA4FBFC 0E1108A8 FD17B448 A6855419 9C47D08F FB10D4B8

Finally the order n of G and the cofactor are:

n = FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE BAAEDCE6 AF48A03B BFD25E8C D0364141

h = 01

Specification of SECP256K1 taken from http://www.secg.org/sec2-v2.pdf An explanation of all these values in the sextuple is given here.
P is the prime *p* that specifies the size of the finite field.

a and *b* are the coefficients of the elliptic curve equation.

G is the base point that generates the required subgroup, also known as generator. Base point can be represented in either compressed or uncompressed form. There is no need to store all points on the curve in practical implementations. The compressed generator works because points on the curve can be identified by using only the *x* coordinate and the least significant bit of the *y* coordinate.

n is the order of the subgroup.

h is the cofactor of the subgroup.

In the following section, an example using OpenSSL is shown to help you understand the practical aspects of RSA.

In the following section, it is shown how RSA public and private key pairs can be generated using OpenSSL.

How to generate public and private key pairs

First, it is shown how the RSA private key can be generated using OpenSSL.

Private key

~/Crypt$ openssl genpkey -algorithm RSA -out privatekey.pem -pkeyopt rsa_keygen_bits:1024
…...++++++
…................. ++++++

After executing the command, a file named privatekey.pemis produced, which contains the generated private key. This is shown as follows:

~/Crypt$ cat privatekey.pem
-----BEGIN PRIVATE KEY-----
MIICdgIBADANBgkqhkiG9w0BAQEFAASCAmAwggJcAgEAAoGBAKJOFBzPy2vOd6em
Bk/UGrzDy7TvgDYnYxBfiEJId/r+EyMt/F14k2fDTOVwxXaXTxiQgD+BKuiey/69
7qRppgmOE5nuEbxkDSQI7OqHYbOLuwfCjHzJBrSgqyi6pj9/9CbXJrZPgNDwdLEb
GgpDKtZs9gLv3A==
-----END PRIVATE KEY-----

Generate public key

As the private key is mathematically linked to the public key, it is possible to generate or derive the public key out of the private key. Taking the example of the preceding private key, the public key can be generated as shown here:

:~/Crypt$ openssl rsa -pubout -in privatekey.pem -out publickey.pem writing RSA key

Public key can be viewed using a file reader or anytext viewer, as shown here:

:~/Crypt$ cat publickey.pem
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCiThQcz8trznenpgZP1Bq8w8u0
dMKArq1xk4IYKiGx0QIDAQAB
-----END PUBLIC KEY-----

In order to see more details about the various components, such as modulus, prime numbers that are used in the process, exponents and coefficients of the generated private key, the following command can be used (the complete output is not shown as it is too large):

:~/Crypt$ openssl rsa -text -in privatekey.pem Private-Key: (1024 bit)
modulus:
00:a2:4e:14:1c:cf:cb:6b:ce:77:a7:a6:06:4f:d4:
publicExponent: 65537 (0x10001) privateExponent:
19:cc:12:94:f9:b4:ce:da:15:d5:0f:5a:07:c6:ee: f9:98:36:37:c4:2c:2a:48:01
prime1:
00:d4:61:5d:38:c0:d4:84:8c:4c:bd:b5:82:74:cd:
4d:5e:30:c6:59
prime2:
00:c3:a3:d4:47:d6:0b:79:bb:26:fd:28:e2:88:e4:
9c:9e:d0:c8:39
Similarly, the public key can be explored using the following commands. Public and Private keys are base64-encoded:

~/Crypt$ openssl pkey -in publickey.pem -pubin -text
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCiThQcz8trznenpgZP1Bq8w8u0
dMKArq1xk4IYKiGx0QIDAQAB
-----END PUBLIC KEY-----

Public-Key: (1024 bit) Modulus:
00:a2:4e:14:1c:cf:cb:6b:ce:77:a7:a6:06:4f:d4:
be:86:9c:3f:23:e4:af:fd:23:84:85:74:c2:80:ae: ad:71:93:82:18:2a:21:b1:d1
Exponent: 65537 (0x10001)

Now the public key can be shared openly and anyone who wants to send us a message can use the public key to encrypt the message and send it to us. We can then use the corresponding private key to decrypt the file.

How to encrypt and decrypt using RSA with OpenSSL

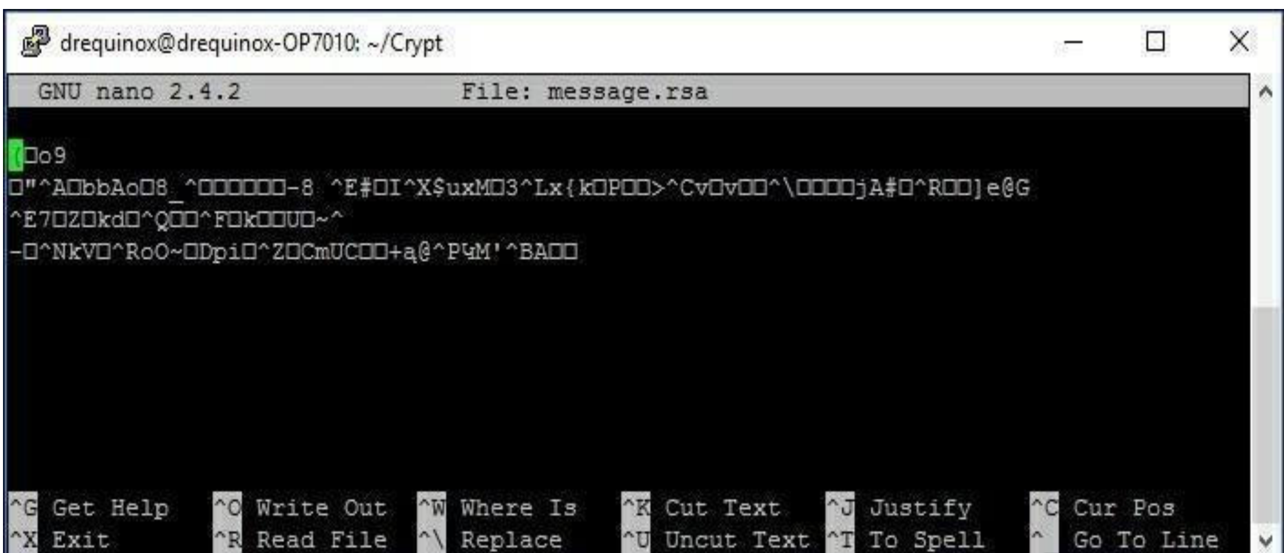
First in the section, an example will presented, which demonstrates how encryption can be performed using RSA.

Encryption

Taking the private keygenerated in the earlier example, the command to encrypt a text file message.txtcan be constructed, as shown here:

:~/Crypt$ openssl rsautl -encrypt -inkey publickey.pem -pubin -in message.txt - out message.rsa

This will produce a file named message.rsa, which is in a binary format. If we open message.rsa in the nano editor, it will show some garbage:



message.rsa showing garbage data

Decrypt In order to decrypt the RSA-encrypted file, the following command can be used

:~/Crypt$ openssl rsautl -decrypt -inkey privatekey.pem -in message.rsa -out message.dec

Now if the file is read using cat, decrypted plain text can be seen, as shown here:

:~/Crypt$ cat message.dec datatoencrypt

ECC using OpenSSL

OpenSSL provides a very rich library of functions to perform elliptic curve cryptography. The following section shows how to practically use ECC functions in OpenSSL.

ECC private and public key pair

In this example, first, an example is presented that demonstrates the creation of a private key using ECC functions available in the OpenSSLlibrary.

Private key

ECC is based on domain parameters defined by various standards. We can see the list of all available standards' defined and recommended curves available in OpenSSLusing the following command:

Crypt$ openssl ecparam -list_curves
secp112r1 : SECG/WTLS curve over a 112 bit prime field secp112r2 : SECG curve over a 112
224 bit prime field
NIST/SECG curve over a 521 bit prime field
prime192v1: NIST/X9.62/SECG curve over a 192 bit prime field
.
.
brainpoolP384r1: RFC 5639 curve over a 384 bit prime field brainpoolP384t1: RFC 5639
brainpoolP512t1: RFC 5639 curve over a 512 bit prime field

As this produces a long output, the complete output is not shown and truncated in between. In the following example, SECP256k1 is used to demonstrate ECC usage.

Private key generation

~/Crypt$ openssl ecparam -name secp256k1 -genkey -noout -out ec-privatekey.pem
~/Crypt$ cat ec-privatekey.pem
-----BEGIN EC PRIVATE KEY-----
MHQCAQEEIJHUIm9NZAgfpUrSxUk/iINq1ghM/ewn/RLNreuR52h/oAcGBSuBBAAK
oUQDQgAE0G33mCZ4PKbg5EtwQjk6ucv9Qc9DTr8JdcGXYGxHdzr0Jt1NInaYE0GG
ChFMT5pK+wfvSLkYl5ul0oczwWKjng==
-----END EC PRIVATE KEY-----

The file named ec-privatekey.pemnow contains the EC private key that is generated based on the SECP256K1 curve.

In order to generate a public key out of a private key, issue the following command:

~/Crypt$ openssl ec -in ec-privatekey.pem -pubout -out ec-pubkey.pem read EC key writing EC key

Reading the file produces the following output, displaying the generated public key:

~/Crypt$ cat ec-pubkey.pem
-----BEGIN PUBLIC KEY-----
MFYwEAYHKoZIzj0CAQYFK4EEAAoDQgAE0G33mCZ4PKbg5EtwQjk6ucv9Qc9DTr8J
dcGXYGxHdzr0Jt1NInaYE0GGChFMT5pK+wfvSLkYl5ul0oczwWKjng==
-----END PUBLIC KEY-----

Now the ec-pubkey.pemfile contains the public key derived out of ec-privatekey.pem. The private key can be further explored using the following command:
~/Crypt$ openssl ec -in ec-privatekey.pem -text -noout
read EC key
Private-Key: (256 bit) priv:
00:91:d4:22:6f:4d:64:08:1f:a5:4a:d2:c5:49:3f:
33:c1:62:a3:9eASN1 OID: secp256k1

Similarly, the public key can be explored further with the following command:

drequinox@drequinox-OP7010:~/Crypt$ openssl ec -in ec-pubkey.pem -pubin -text - noout
read EC key
Private-Key: (256 bit) pub:
04:d0:6d:f7:98:26:78:3c:a6:e0:e4:4b:70:42:39:
33:c1:62:a3:9e ASN1 OID: secp256k1
drequinox@drequinox-OP7010:~/Crypt$

It is also possible to generate a file with the required parameters-in this case, SECP256K1-and then explore it further to understand the underlying parameters:

~/Crypt$ openssl ecparam -name secp256k1 -out secp256k1.pem drequinox@drequinox-OP7010:~/Crypt$ cat secp256k1.pem
-----BEGIN EC PARAMETERS-----
BgUrgQQACg==
-----END EC PARAMETERS-----

The file now contains all SECP256K1parameters and can be analyzed using the following command:

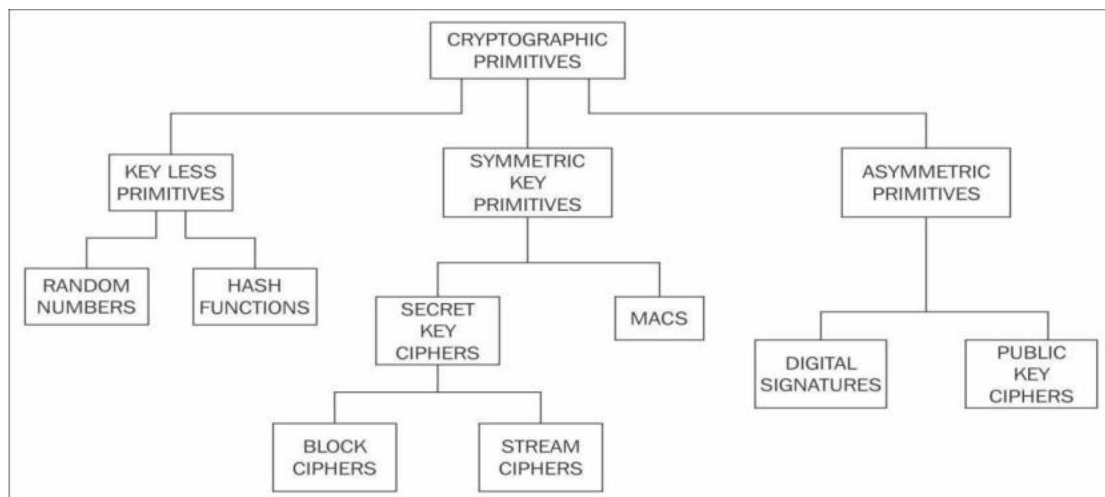
drequinox@drequinox-OP7010:~/Crypt$ openssl ecparam -in secp256k1.pem -text -param_enc explicit -noout
Field Type: prime-field Prime:
00:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:
ff:fc:2f
A: 0
B: 7 (0x7)
Generator (uncompressed): 04:79:be:66:7e:f9:dc:bb:ac:55:a0:62:95:ce:87:
0b:07:02:9b:fc:db:2d:ce:28:d9:59:f2:81:5b:16:
8f:fb:10:d4:b8 Order:
ff:fe:ba:ae:dc:e6:af:48:a0:3b:bf:d2:5e:8c:d0:
36:41:41
Cofactor: 1 (0x1)

The preceding example shows the prime number used and values of Aand B the with generator, order and cofactor of the SECP256K1curve domain parameters.

There is another category of cryptographic primitives that is known as hash functions. Hash functions are not used to encrypt; data instead, they produce a fixed length digest of text.

Cryptographic primitives

This taxonomyof cryptographic primitives can be visualized as shown here:



Cryptographic primitives

Hash functions

Hash functions are used to create fixed length digests of arbitrarily long input strings. Hash functions are keyless and provide the data integrity service. They are usually built using iterated and dedicated hash function construction techniques. Various families of hash functions are available, such as MD, SHA1, SHA-2, SHA-3, RIPEMD, and Whirlpool. Hash functions are commonly used in digital signatures and message authentication codes, such as HMACs. They have three security properties, namely pre-image resistance, second pre-image resistance, and collision resistance. These properties are explained later in the section.

Hash functions are typically used to provide data integrity services. These can be used as one-way functions and to construct other cryptographic primitives, such as MACs and digital signatures. Some applications used hash functions as a means of generating **pseudo random numbers** (**PRNGs**). Hash functions do not require a key. There are two practical and three security properties of hash functions that must be met depending on the level of requirements of integrity.

Compression of arbitrary messages into fixed length digest

This property is concerned with the fact that a hash function must be able to take a long input text of any length and output a fixed length compressed message. Hash functions produce a compressed output in various bit sizes, usually between 128-bits and 512-bits.

Easy to compute

Hash functions are efficient and fast one-way functions. The requirement is that they be very quick to compute regardless of the message size. The efficiency may decrease if the message is too big but the function should still be fast enough for practical use.

In the following section, security properties of hash functions are discussed.

Pre-image resistance

Consider an equation:

$h(x) = y$

Here, h is the hash function, x is the input, and y is the hash. The first security property requires that y cannot be reverse computed to x. x is considered a *pre-image* of y, hence the name pre-image resistance. This is also called one-way property.

Second pre-image resistance

This property requires that given x and $h(x)$, it is almost impossible to find any other message m , where m != x and *hash of m = hash of x*. $h(m) = h(x)$. This property is also known as weak collision resistance.

Colision resistance

This property requires that two different input messages should not hash to the same output. In other words, *h(x) != h(z)*. This property is also known as strong collision resistance.

Hash functions, due to their very nature, will always have some collisions, and that is where two different messages hash to the same output, but they should be computationally infeasible to find. A concept known as **avalanche effect** is desirable in all hash functions. Avalanche effect specifies that a small change, even a single character change in the input text, will result in a totally different hash output.

Hash functions are usually designed by following iterated hash functions approach. In this method, the input message is compressed in multiple rounds on a block-by-block basis to produce the compressed output. A popular type of iterated hash function is Merkle-Damgard construction. This construction is based on the idea of dividing the input data into equal sizes of blocks and then feeding them through the compression functions in an iterative manner. The collision resistance of the property of compression functions ensures that the hash output is also collision-resistant.

There are multiple hash function categories. You will be introduced to these categories in the upcoming section.

Message Digest (MD)

Message Digest functions were very popular in early 1990s. MD4 and MD5 are members of this category. Both MD functions are found to be insecure and not recommended for use any more. MD5 is a 128-bit hash function that was commonly used for file integrity checks.

Secure Hash Algorithms (SHAs)

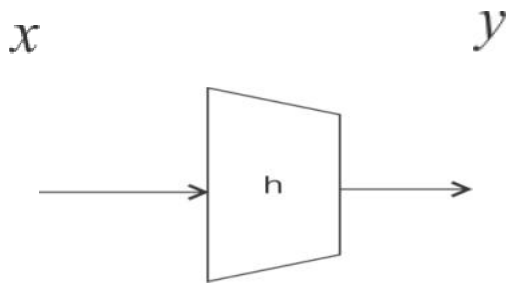
SHA-0: This is a 160-bit function introduced by NIST in 1993.

SHA-1: SHA-1 was introduced later by NIST as a replacement of SHA-0. This is also a 160-bit hash function. SHA-1 is used commonly in SSLand TLS implementations. It should be noted that SHA-1 is now considered insecure and is being deprecated by certificate authorities. Its usage is now discouraged in any new implementations. **SHA-2**: This category includes four functions defined by the number of bits of the hash: SHA-224, SHA-256, SHA-384 and SHA-512.

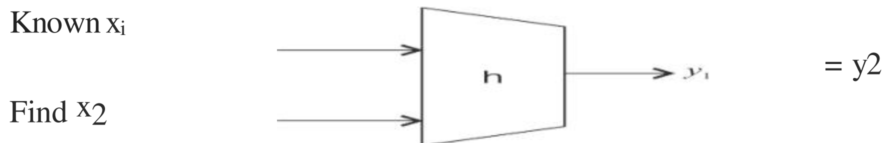
SHA-3: This is the latest family of SHA functions. SHA3-224, SHA3-256, SHA3-384 and SHA3-512 are members of this family. SHA3 is a NIST-standardized version of Keccak. Keccak uses a new approach called *sponge construction* instead of the commonlyused Merkle-Damgard transformation.

RIPEMD: RIPEMD is the acronym for *RACE Integrity Primitives Evaluation Message Digest*. It is based on the design ideas used to build MD4. There are multiple versions of RIPEMD, including 128-bit, 160-bit, 256-bit, and 320-bit.

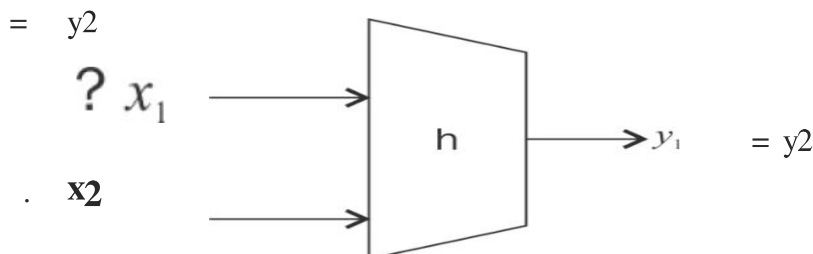
Whirlpool: This is based on a modified version of Rijndael cipher known as W. It uses the Miyaguchi-Preneel compression function, which is a type of one-way function used for the compression of two fixed length inputs into a single fixed length output. It is a single block length compression function:



1- PRE - IMAGE RESISTANCE



2 - SECOND PRE IMAGE RESISTANCE



3 - STRONG COLLISION RESISTANCE

Three security properties of hash functions

Hash functions have many practical applications ranging from simple file integrity checks and password storage to be used in cryptographic protocols and algorithms. They are used in hash tables, distributed hash tables, bloom filters, virus finger printing, peer-to-peer P2P file sharing, and many other applications.

Design of Secure Hash Algorithms (SHA)

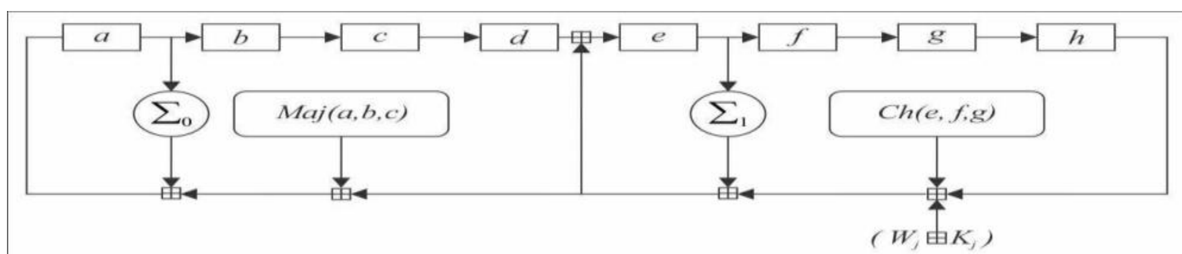
In the following section, you will be introduced to the design of SHA-256 and SHA-3. Both of these are used in bitcoin and Ethereum, respectively. Ethereum doesn't use NIST Standard SHA-3 but Keccak, which is the original algorithm presented to NIST. NIST, after some modifications such as increase in the number of rounds and simpler message padding, standardized Keccak as SHA-3.

SHA-256

SHA-256 has the input message size < 2^64-bits. Block size is 512-bits and has a word size of 32-bits. Output is 256-bit digest. The compression function processes a 512-bit message block and a 256- bit intermediate hash value. There are two main components of this function: compression function and a message schedule.

The algorithm works as follows:

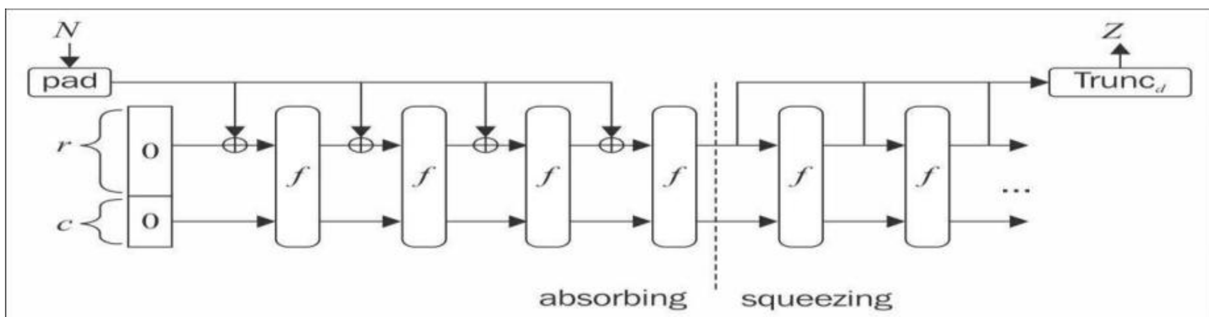
- **Pre-processing:**
  1. Padding of the message, which is used to make the length of a block to 512-bits if it is smaller than the required block size of 512-bits.
  2. Parsing the message into message blocks that ensure that the message and its padding is divided into equal blocks of 512-bits.
  3. Setting up the initial hash value, which is the eight 32-bit words obtained by taking the first 32-bits of the fractional parts of the square roots of the first eight prime numbers. These initial values are randomly chosen in order to initialize the process and gives a level of confidence that no backdoor exists in the algorithm.
- **Hash computation:**
  1. Each message block is processed in a sequence and requires 64 rounds to compute the full hash output. Each round uses slightly different constants to ensure that no two rounds are the same.
  2. First, the message schedule is prepared.
  3. Then, eight working variables are initialized.
  4. Then, the intermediate hash value is calculated.
  5. Finally, the message is processed and the output hash is produced:



Design of SHA3 (Keccak)

The structure of SHA-3 is very different from the usual SHA-1 and SHA-2. The key idea behind SHA-3 is based on un-keyed permutations as opposed to other usual hash functions' constructions that used keyed permutations. Keccak also does not make use of the Merkle-Damgard transformation that is commonly used to handle arbitrary length input messages in hash functions. A newer approach called sponge and squeeze construction is used in Keccak, which is basically a random permutation model. Different variants of SHA3 have been standardized, such as SHA3-224, SHA3-256, SHA3-384, SHA3-512, SHAKE128, and SHAKE256. SHAKE128 and SHAKE256 are extendable output functions that are also standardized by NIST.XOF functions that allow the output to be extended to any desired length.

The following diagram shows the sponge and squeeze model that is the basis of SHA3 or Keccak. As an analogy to sponge, first, the data is absorbed into the sponge after applying padding, where it is then changed into a subset of permutation state using XOR and then the output is squeezed out of the sponge function that represents the transformed state. Rate is the input block size of a sponge function, whereas capacity determines the generic security level:



SHA-3 absorbing and squeezing function in SHA3

OpenSSL example of hash functions

The following command will produce a hash of 256-bits of Hello messages using the SHA256 algorithm:

:~/Crypt$ echo -n 'Hello' | openssl dgst -sha256
(stdin)= 185f8db32271fe25f561a6fc938b2e264306ec304eda518007d1764826381969

Note that even a small change in the text, such as changing the case of *H*, results in a big change in the output hash. This is known as *avalanche effect*, as discussed earlier:

```
:~/Crypt$ echo -n 'hello' | openssl dgst -sha256
(stdin)=    2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
```

Note that both outputs are completely different:

Hello: 18:5f:8d:b3:22:71:fe:25:f5:61:a6:fc:93:8b:2e:26:43:06:ec:30:4e:da:51:80:07:d1:76 :48:26:38:19:69
hello: 2c:f2:4d:ba:5f:b0:a3:0e:26:e8:3b:2a:c5:b9:e2:9e:1b:16:1e:5c:1f:a7:42:5e:73:04:33 :62:93:8b:98:24

Message Authentication codes (MACs)

MACs are sometimes called keyed hash functions and can be used to provide message integrity and authentication. In others words, they are used to provide data origin authentication. These are symmetric cryptographic primitives using a shared key between the sender and the receiver. MACs

can be constructed using block ciphers or hash functions.

MACs using block ciphers

In this approach, block ciphers are used in the **Cipher block chaining mode** (**CBC mode**) in order to generate a MAC. Any block cipher-for example, AES in the CBC mode-can be used. The MAC of the message is in fact the output of the last round of the CBC operation. The length of the MAC output is the same as the block length of the block cipher used to generate MAC. MACs are verified simply by computing the MAC of the message and comparing it with the received MAC. If they are the same, then the message integrity is confirmed; otherwise, the message is considered altered. It should also be noted that MACs work like digital signatures, but they cannot provide the nonrepudiation service due to their symmetric nature.

HMACs (hash-based MACs)

Similar to the hash function, they produce a fixed length output and take an arbitrarily long message as the input. In this scheme, the sender signs a message using MAC and the receiver verifies it using the shared key. The key is hashed with the message using either of the two methods known as secret prefix or the secret suffix method. In the first method, the key is concatenated with the message, that is, the key comes first and the message comes after, whereas in the latter method, the key comes after the message:

Secret prefix: $M = MACk(x) = h(k||x)$

Secret suffix: $M=MACk(x) = h(x||k)$

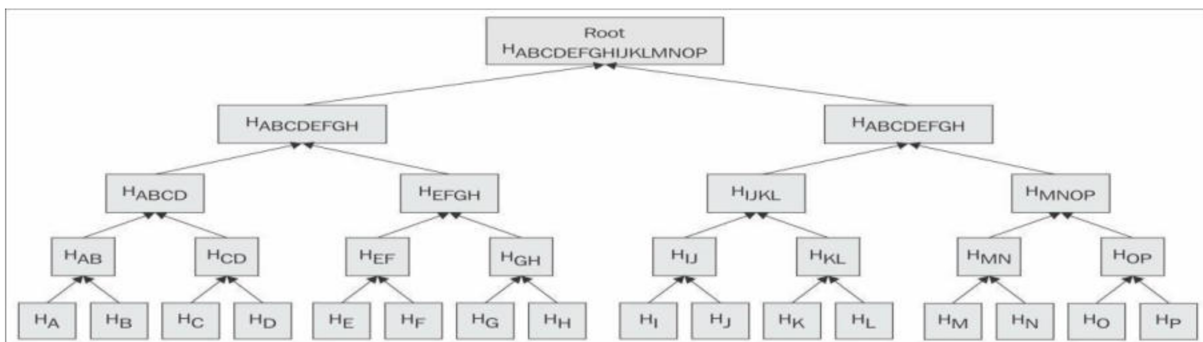
There are pros and cons of both methods. Some attacks on both schemes have been discovered. There are HMAC constructions schemes that use various techniques, such as **ipad** and **opad** (inner padding and outer padding) proposed by researchers that are considered secure with some assumptions:

Operation of a MAC function

Merkle trees

The concept of Merkle tree was introduced by *Ralph Merkle*. A visualization of Merkle tree is shown here, which makes it easy to understand. Merkle trees allow secure and efficient verification of large data sets.

It is a binary tree in which first, the inputs are placed at the leaves (node with no children), and then values of pairs of child nodes are hashed together in order to produce a value for the parent node (internal node) until a single hash value known as Merkle root is achieved:



A Merkle tree

Patricia trees

In order to understand Patricia trees, first, you will be introduced to the concept of a **trie**. A trie or a digital tree is an ordered tree data structure used to store a dataset.

Practical Algorithm to Retrieve Information Coded in Alphanumeric (**Patricia**), also known as Radix tree, is a compact representation of a trie in which a node that is the only child of a parent is merged with its parent.

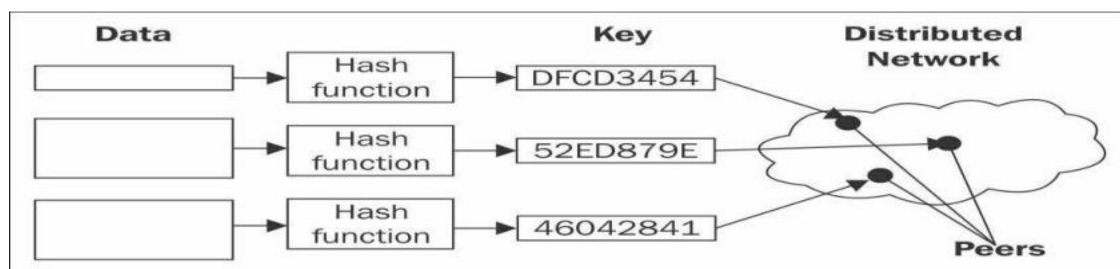
Merkle-Patricia tree, based on the definitions of Patricia and Merkle, is a tree that has a root node that contains the hash value of the entire data structure.

Distributed hash tables (DHTs)

A hash table is a data structure that is used to map keys to values. Internally, a hash function is used to calculate an index into an array of buckets, from which the required value can be found. Buckets have records stored in them using a hash key and are organized in a particular order.

With the definition provided earlier in mind, one can think of the distributed hash table as a data structure where data is spread across various nodes and nodes are equivalent to buckets in a peer-to-peer to network.

The following diagram visually shows how a DHT works. The example shows that data is passed through a hash function, which results in generating a compact key. This key is then linked with the data (values) on the peer-to-peer network. When users on the network request the data (via the filename), the filename can be hashed again to produce the same key and any node on the network can then be requested to find the corresponding data. DHTs provides decentralization, fault tolerance, and scalability



Distributed hash tables

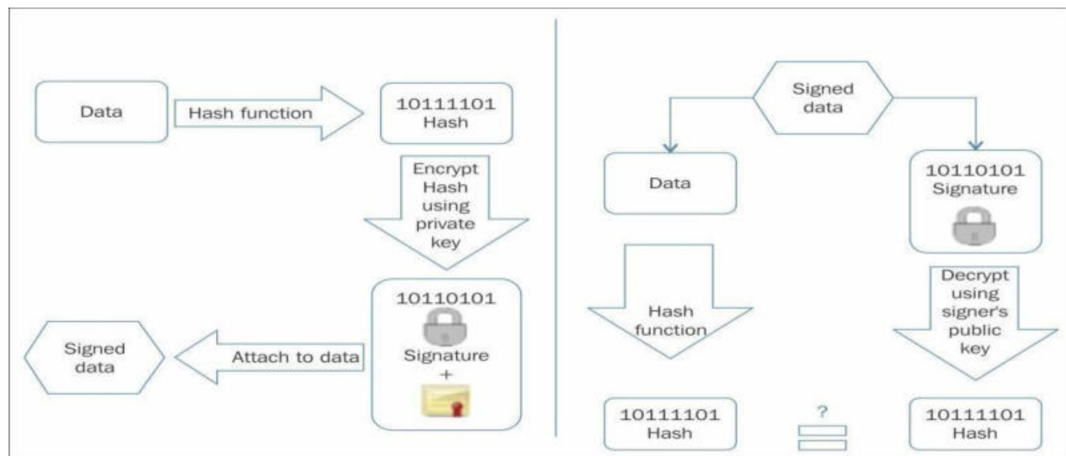
Digital signatures

Digital signatures provide a means of associating a message with an entity from which the message has been originated. Digital signatures are used to provide data origin authentication and nonrepudiation. They are calculated in two steps. High-level steps of an RSA digital signature scheme is given as follows:

1. Calculate the hash value of the data packet. This will provide the data integrity guarantee as hash can be computed at the receiver's end again and matched with the original hash to check whether the data has been modified in transit. Technically, message signing can work without hashing the data first, but is not considered secure.

2. The second step signs the hash value with the signer's private key. As only the singer has the private key, the authenticity of the signature and the signed data is ensured.
Digital signatures have some important properties, such as authenticity, un forgeability, and non reusability. Authenticity means that the digital signatures are verifiable by a receiving party. The un forgeability property ensures that only the sender of the message is able to use the signing functionality using the private key. In other words, no one else should be able to produce the signed message that has been produced by the legitimate sender. Non reusability means that the digital signature cannot be separated from a message and used for another message again.

The operation of a generic digital signature function is shown in the following diagram:



Digital signing (left) and verification process (right) (Example of RSA digital signatures)

If a sender wants to send an authenticated message to a receiver, there are two methods that can be used. These two approaches to use digital signatures with encryption are introduced here.

Sign then encrypt

In this approach, the sender digitally signs the data using the private key, appends the signature to the data, and then encrypts the data and the digital signature using the receiver's public key. This is considered a more secure scheme as compared to the encrypt then sign scheme described next.

Encrypt then sign

In this approach, the sender encrypts the data using the receiver's public key and then digitally signs the encrypted data.

Note: In practice, a digital certificate that contains the digital signature is issued by a certificate authority **(CA)** that associates a public key with an identity.

Elliptic Curve Digital signature algorithm (ECDSA)

In order to sign and verify using the ECDSA scheme, the first keypair needs to be generated:

1. First, define an elliptic curve *E*:
1. Withmodulus *P*.
2. Coefficients *a* and *b*.
3. Generator point *A* that forms a cyclic group of prime order *q*.

   An integer *d* is chosen randomly so that $0 < d < q$.
   Calculate public key *B* so that $B = d A$.

Public key is the sextuple of the formshown here:

Kpb = (p,a,b,q,A,B)

Private key is randomly chosen *d* in Step 2:

Kpr = d

The signature can be calculated as follows:

$$S = (h(m) + dr)K_{e^{-1}} \bmod q$$

Here, *m* is the message for which the signature is being computed and *h(m)* is the hash of the message *m*.

Signature verification is carried out by following this process.

1. Auxiliaryvalue w is calculated as $w = s^{-1} mod\ q$.
2. Auxiliaryvalue $u1 = w.\ h(m)\ mod\ q$.
3. Auxiliary value $u2 = w.\ r\ mod\ q$.
4. Calculate Point *P*, $P = u1A + u2B$.
5. Verification is carried out as follows.
6. *r*, *s* is accepted as a valid signature if x-coordinate of the point *P* calculated in Step 4 has the same value as the signature parameter *r mod q*.

that is: *Xp = r mod q means valid signature Xp != r mod q means invalid signature*

Various practical examples are shown here, which shows how the RSA digital signature can be generated, used, and verified using OpenSSL.

How to generate a digital signature

The first step is to generate a hash of the message file:

:~/Crypt$ openssl dgst -sha256 message.txt SHA256(message.txt)= eb96d1f89812bf4967d9fb4ead128c3b787272b7be21dd2529278db1128d559c

Both hash generation and signing can be done in a single step, as shown here. Note that privatekey.pemis generated in the steps provided previously:

:~/Crypt$ openssl dgst -sha256 -sign privatekey.pem -out signature.bin message.txt

Now let's display the directory showing relevant files:

:~/Crypt$ cat signature.bin

In order to verify the signature, the following operation can be performed:

```
V _[h] h t + T~Ol s { Cq"# A Q U, uf p* *7 T' u eAy
$ x <$ a ` :L qWh uG = $ :~/Crypt$
```

:~/Crypt$ openssl dgst -sha256 -verify publickey.pem -signature signature.bin message.txt
Verified OK
:~/Crypt$

Similarly, if some other signature file that is not valid is used, the verification will fail, as shown here:

:~/Crypt$ openssl dgst -sha256 -verify publickey.pem -signature someothersignature.bin message.txt
Verification Failure

Now you are introduced to an example that shows how OpenSSL can be used to perform ECDSA-related operations.

Now suppose a file named testsign.txt needs to be signed and verified. This can be achieved as follows:

1. Create a test file:

   ~/Crypt$ echo testing > testsign.txt
   ~/Crypt$ cat testsign.txt testing

2. Run the following command to generate a signature using a private keyfor the testsign.txt file:

   ~/Crypt$ openssl dgst -ecdsa-with-SHA1 -sign eccprivatekey.pem

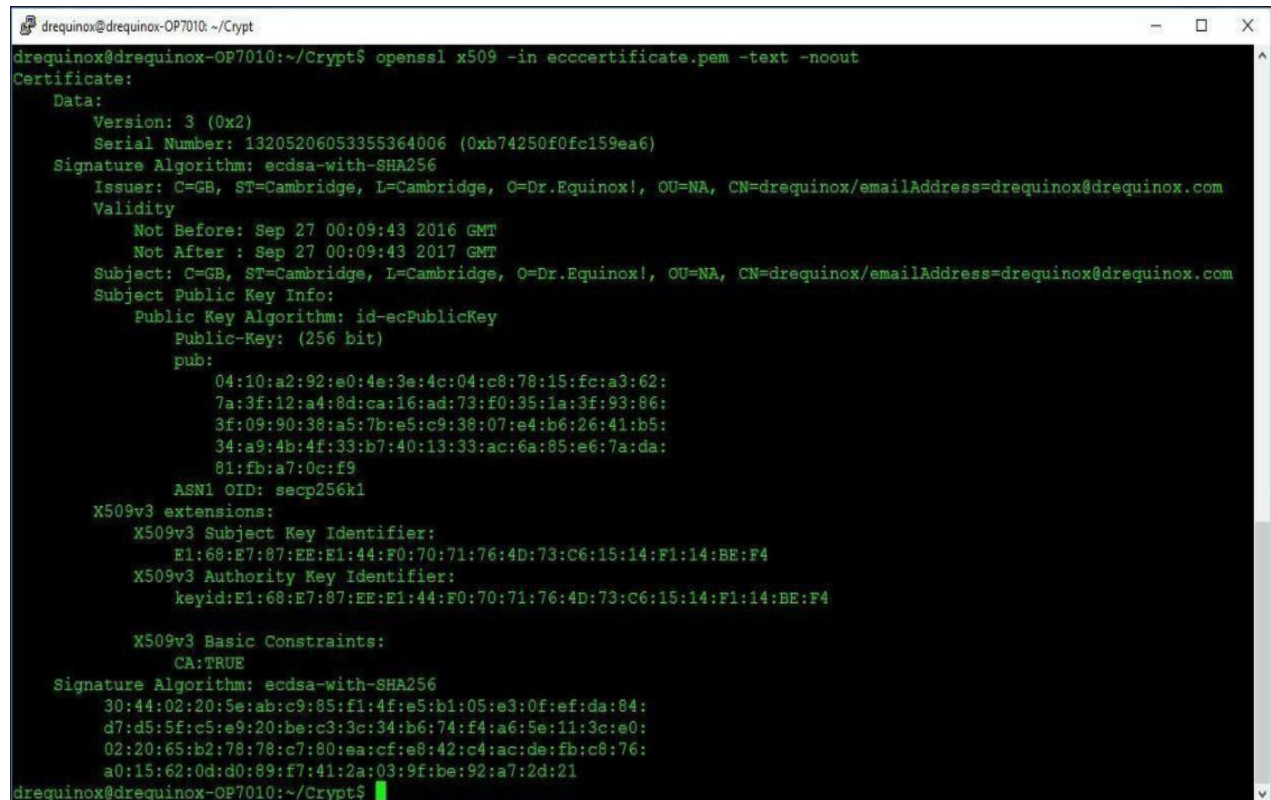
   testsign.txt > ecsign.bin

3. Finally, the command for verification can be run as shown here:

   ~/Crypt$ openssl dgst -ecdsa-with-SHA1 -verify eccpublickey.pem
   -signature ecsign.bin testsign.txt Verified OK

A certificate can also be generated using the private keygenerated earlier.
The certificate can be explored using the command below.

~/Crypt$ openssl x509 -in ecccertificate.pem -text –noout



X509 certificate that uses ECDSA algorithm with SHA-256

There are other topics in cryptography that are presented here due to their relevance to blockchain or potential use in future blockchain ecosystems.

Signcryption

Signcryption is a public key cryptography primitive that provides all the functions of the digital signature and encryption. It was invented by *Yuliang Zheng* and is now an ISO standard ISO/IEC 29150:2011. Traditionally, signature then encrypt or encrypt then sign schemes are used to provide unforgeability, authentication, and nonrepudiation, but with Signcryption, all services of digital signatures and encryption are provided with cost less than that of sign then encrypt schemes.

This is **Cost (signature & encryption) << Cost (signature) + Cost (Encryption)** in a single logical step.

Zero knowledge proofs

Zero knowledge proofs were introduced by *GoldWasser*, *Micali*, and *Rackoff*. These proofs are used to prove the validity of an assertion without revealing any information whatsoever about the assertion. There are three properties of ZKPs that are required, namely completeness, soundness, and zero-knowledge property.

Zero-knowledge property, as the name implies, is the key property of zero knowledge proofs whereby it is ensured that absolutely nothing is revealed about the assertion except whether it is true or false.

Zero knowledge proofs have sparked a special interest among researchers in the blockchain space due to its privacy properties that are very much desirable in financial and many other fields, such as law and medicine. A recent example of the successful implementation of the zero knowledge proof mechanism is the Zcash crypto currency. In Zcash, a specific type of zero knowledge proof, known as **zero-knowledge Succinct Non-interactive Argument of Knowledge** (**ZK-Snark**), is implemented.

Blind signatures

Blind signatures were introduced by *David Chaum* in 1982 and are based on public key digital signature schemes, such as RSA. The key idea behind blind signatures is to get the message signed by the signer without actually revealing the message. This is achieved by disguising or blinding the message before signing it, hence the name blind signatures. This blind signature can then be verified against the original message just like a normal digital signature. Blind signatures were introduced as a mechanism to allow the development of digital cash schemes.

Encoding schemes

Other than cryptographic primitives, binary to text encoding schemes are also used in various scenarios. The most common usage is to convert binary data into text so that it can be either processed, saved, or transmitted via a protocol that does not support the processing of binary data.

For example, sometimes, images are stored in the database as base64 encoding, which allows a text field to be able to store a picture. A commonly used encoding scheme is base64. Another encoding named base58 was popularized by its usage in bitcoin.

Financial markets and trading

Financial markets exist to facilitate the transfers of savings from savers to investors. In an economic system, there are two sectors, namely household and business. Financial markets, at their core, act as an intermediary between the savers and the investors. Basically, there are three types of markets, namely money markets, credit markets, and capital markets. Money markets are short-term markets where money is lent to companies or banks to do interbank lending. Foreign exchange or FX is another category of money markets where currencies are traded. Credit markets consist mostly of retail banks where they borrow money from central banks and loan it to companies or households in the form of mortgages or loans.

Capital markets facilitate the buying and selling of financial instruments, mainly stocks and bonds. Capital markets can be divided into two types, primary and secondary markets. Stocks are issued directly by the companies to investors in primary markets, whereas in secondary markets, investors resell their securities to investors via stock exchanges. Various electronic trading systems are used by exchanges to facilitate the trading of financial instruments.

Trading

A market is a place where traders come to trade. It can ent asset classes.

Trading can be defined as an activity in which traders buy or sell various financial instruments to generate profit and hedge risk. Investors, borrowers, hedgers, asset exchangers, and gamblers are a few types of traders. Traders have a short position when they owe something, for example, if they have sold a contract and have a long position when they buy a contract. There are various ways to transact trades, such as through brokers or directly on the exchange or over the counter. Brokers are agents who arrange trades for their customers. Brokers act on clients' behalf to deal at a given price or at the best possible price.

Exchanges

Exchanges are usually considered to be a very safe, regulated, and reliable place for trading. Recently, electronic trading has gained high popularity as compared to traditional floor-based trading. Now traders send orders to a central electronic order book from where the orders, prices, and related attributes are published to all associated systems using communication networks thus, in essence, creating a virtual marketplace. Exchange trades can be performed only by members of the exchange.
In order to trade without these limitations, the counter parties can participate in **OTC** (**Over the Counter**) trading directly.

Orders and order properties

Orders are instructions to trade and are the main building blocks of a trading system. They have the following general attributes:

1. The instrument name.
2. Quantity.
3. Direction (buyor sell).
4. The type of the order that represents various conditions. for example, limit orders and stop orders, an example of which is 1500 Royal Bank of Scotland ordinary shares for GBP £15.50.

Orders are traded on the basis of bid prices and offer prices.

Order management and routing systems

Order routing systems routes and deliver orders to various destinations depending on the business logic. Customers use them to send orders to their brokers, who then send these orders to dealers, clearing houses, and exchanges.

There are different types of orders; the two most common ones are markets orders and limit orders. A market order is an instruction to trade at the best price currently available in the market, and these orders get filled immediately at spot prices. On the other hand, a limit order is an instruction to trade at the best price available but only if it is not lower than the limit price set by the trader. This can also be higher depending on the direction of the order, either sell or buy. All these orders are managed in an order book, which is a list of orders maintained by an exchange, and records the intention of buying or selling by the traders.

Components of a trade

A trade ticket is the combination of all details related to a trade. However, there is some variation depending on the type of the instrument and asset class, but generally, all instruments have the attributes discussed in the next section.

Note

The underlying instrument

The underlying instrument is the basis of the trade. It can be a currency, a bond, interest rate, commodity, or equities.

General attributes

This includes general identification information and basic features associated with every trade. Common attributes include a unique ID, instrument name, type, status, trade date, and time.

Economic

These are features related to the value of the trade, for example, buy or sell value, ticker, exchange, price, and quantity.

Sales

Sales refers to the sales-characteristic-related details, such as the name of the sales person, and is just an information field, usually without any impact on the trade life cycle.

Counterparty

Counterparty is an important component of a trade as it shows the other side of the trade and is required to settle the trade successfully. Usual attributes include counterparty name, address, payment type, any reference IDs, settlement date, and delivery type.

Trade life cycle

A general trade life cycle includes various stages from order placement to execution and then settlement. This life cycle is described step by step as follows:

- **Pre-execution**: An order is placed at this stage.
- **Execution and booking**: When the order is matched and executed, it converts into a trade. At this stage, the contract between counter parties is matured.
- **Confirmation**: This is where both counter parties agree to particulars of the trade.
- **Post booking**: This stage is concerned with various scrutiny and verification processes to ascertain the correctness of the trade.
- **Settlement**: This is the most vital part during trade and at this stage, the trade is final.
- **Overnight (end of day processing)**: End of day processes include report generation, profit and loss calculations, and various risk calculations.
- In all the mentioned processes, many people and business functions are involved. Most commonly, these functions are divided into functions such as front office, middle office, and back office.

Unit – III

Bitcoin is the first application of the blockchain technology.

Bitcoin has started a revolution with the introduction of the very first fully decentralized digital currency, and one that has proven to be extremely secure and stable. This has also sparked a great interest in academic and industrial research and introduced many new research areas. Since its introduction in 2008, bitcoin has gained much popularity and is currently the most successful digital currency in the world with billions of dollars invested in it. It is built on decades of research in the field of cryptography, digital cash, and distributed computing. In the following section, a brief history is presented in order to provide the background required to understand the foundations behind the invention of bitcoin.

Digital currencies have always been an active area of research for many decades. Early proposals to create digital cash go as far back as the early 1980s**. In 1982,** ***David Chaum*** proposed a scheme that used blind signatures to build untraceable digital currency. In this scheme, a bank would issue digital money by signing a blind and random serial number presented to it by the user. The user could then use the digital token signed by the bank as currency. The limitation in this scheme was that the bank had to keep track of all used serial numbers. This was a central system by design and required to be trusted by the users. Later on in **1990,** ***David Chaum*** proposed a refined version named e-cash that not only used blinded signature, but also some private identification data to craft a message that was then sent to the bank. This scheme allowed the detection of double spending but did not prevent it. If the same token was used at two different locations, then the identity of the double spender would be revealed. e-cash could only represent a fixed amount of money. ***Adam Back's*** **hashcash, introduced in 1997**, was originally proposed to thwart e-mail spam. The idea behind hashcash was to solve a computational puzzle that was easy to verify but comparatively difficult to compute.

The idea was that for a single user and a single e-mail, extra computational effort was not noticeable, but someone sending a large number of spam e-mails would be discouraged as the time and resources required to run the spam campaign would increase substantially.

B-money was proposed by *Wei Dai* in 1998, which introduced the idea of using Proof of Work to create money. A major weakness in the system was that an adversary with higher computational power could generate unsolicited money without allowing the network to adjust to an appropriate difficulty level. The system lacked details on the consensus mechanism between nodes and some security issues such as Sybil attacks were also not addressed. At the same time, *Nick Szabo* introduced the concept of BitGold, which was also based on the Proof of Work mechanism but had the same problems as b-money with the exception that the network difficulty level was adjustable. *Tomas Sander* and *Ammon TaShama* introduced an e-cash scheme in 1999 that, for the first time, used Merkle trees to represent coins and zero knowledge proofs to prove the possession of coins. In the scheme, a central bank was required that kept a record of all used serial numbers. This scheme allowed users to be fully anonymous albeit at a computational cost. **RPOW** (**Reusable Proof of Work**) was introduced by *Hal Finney* in 2004 and used the hashcash scheme by *Adam Back* as a proof of computational resources spent to create the money. This was also a central system that kept a central database to keep track of all used POW tokens. This was an online system that used remote attestation made possible by a **trusted computing platform** (**TPM hardware**).

All the previously mentioned schemes are intelligently designed but were weak from one aspect or another. Especially, all these schemes rely on a central server that is required to be trusted by the users.

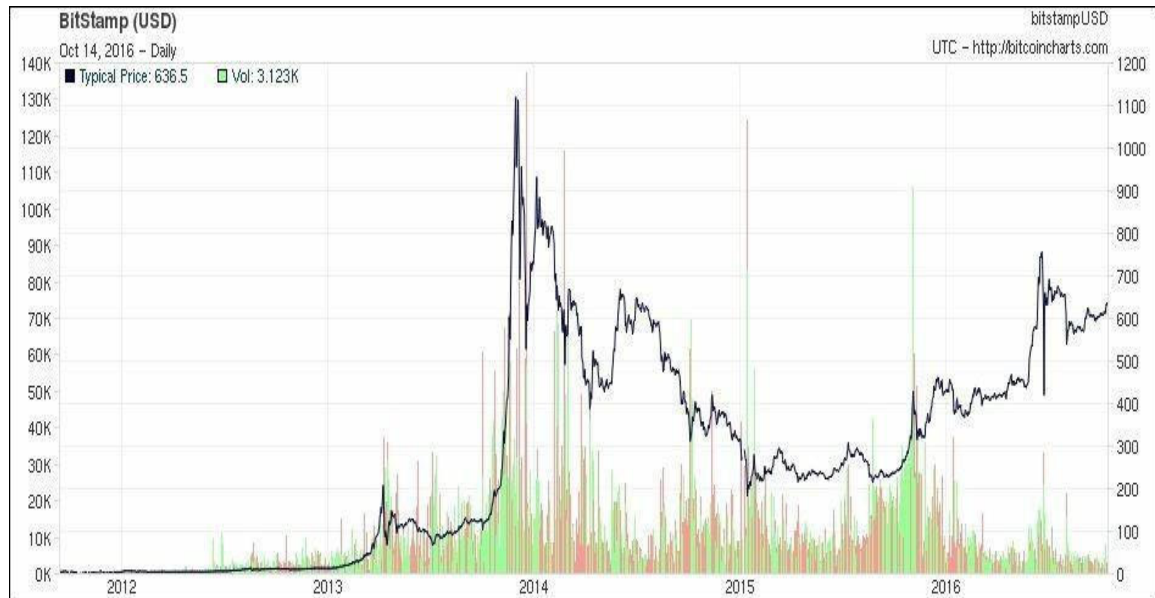
Bitcoin

In 2008, a paper on bitcoin, ***Bitcoin: A Peer-to-Peer Electronic Cash System*** **was written by *Satoshi Nakamoto*.** The first key idea introduced in the paper was that purely peer-to-peer electronic cash that does need an intermediary bank to transfer payments between peers.

Bitcoin is built on decades of cryptographic research such as the research in Merkle trees, hash functions, public key cryptography, and digital signatures. Moreover, ideas such as BitGold, b-money, hashcash, and cryptographic time stamping provided the foundations for bitcoin invention. All these technologies are cleverly combined in bitcoin to create the world's first decentralized currency. The key issue that has been addressed in bitcoin is an

elegant solution to the Byzantine Generals problem along with a practical solution of the double-spend problem.

The value of bitcoin has increased significantly since 2011, as shown in the following graph:



Bitcoin price and volume since 2012 (on logarithmic scale)
Currently (at the time of writing this), bitcoin price is 815 GBP.

Bitcoin definition

Bitcoin can be defined in various ways; it's a protocol, a digital currency, and a platform. It is a combination of peer-to-peer network, protocols, and software that facilitate the creation and usage of the digital currency named bitcoin. Note that Bitcoin with a capital *B* is used to refer to the Bitcoin protocol, whereas bitcoin with a lowercase *b* is used to refer to bitcoin, the currency. Nodes in this peer-to-peer network talk to each other using the Bitcoin protocol.

Decentralization of currency was made possible for the first time with the invention of bitcoin. Moreover, the double spending problem was solved in an elegant and ingenious way in bitcoin. Double spending problem arises when, for example, a user sends coins to two different users at the same time and they are verified independently as valid transactions.

Keys and addresses

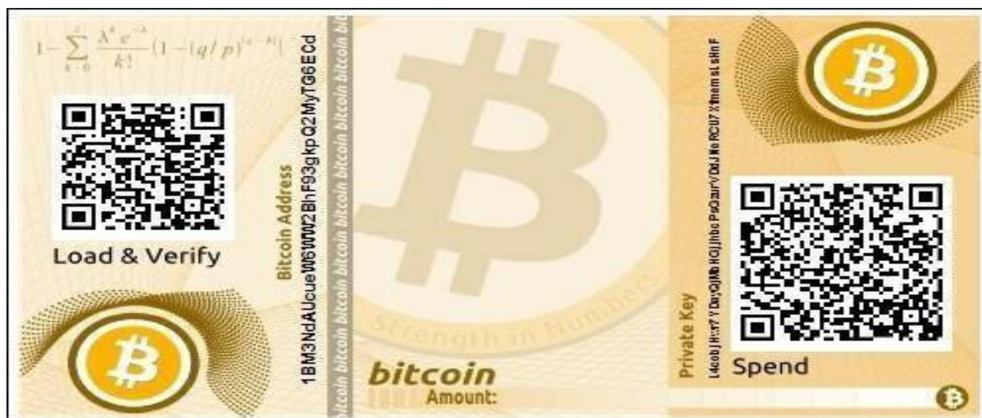
Elliptic curve cryptography is used to generate public and private key pairs in the Bitcoin network. The bitcoin address is created by taking the corresponding public key of a private key and hashing it twice, first with the SHA256 algorithm and then with RIPEMD160. The resultant 160-bit hash is then prefixed with a version number and finally encoded with a Base58Check encoding scheme. The bitcoin addresses are 26-35 characters long and begin with digit 1 or 3. A typical bitcoin address looks like a string shown here:

1ANAguGG8bikEv2fYsTBnRUmx7QUcK58wt

This is also commonly encoded in a QR code for easy sharing. The QR code of the preceding address is shown in the following image:

QR code of a bitcoin address 1ANAguGG8bikEv2fYsTBnRUmx7QUcK58wt

Currently, there are two types of addresses, the commonly used P2PKH and another P2SH type, starting with 1 and 3, respectively. In the early days, bitcoin used direct Pay-to-Pubkey, which is now superseded by P2PKH. However, direct Pay-to-Pubkey is still used in bitcoin for coinbase addresses. Addresses should not be used more than once; otherwise, privacy and security issues can arise. Avoiding address reuse circumvents anonymity issues to an extent, bitcoin has some other security issues as well, such as transaction malleability, which requires different approaches to resolve.



From bitaddress.org private key and bitcoin address in a paper wallet

Public keys in bitcoin

In public key cryptography, public keys are generated from private keys. Bitcoin uses ECC based on the SECP256K1 standard. A private key is randomly selected and is 256-bit in length. Public keys can be presented in an uncompressed or compressed format. Public keys are basically *x* and *y* coordinates on an elliptic curve and in an uncompressed format and are presented with a prefix of 04 in a hexadecimal format. *X* and *Y* coordinates are both 32-bit in length. In total, the compressed public key is 33 bytes long as compared to 65 bytes in the uncompressed format. The compressed version of public keys basically includes only the *X* part, since the *Y* part can be derived from it. The reason why the compressed version of public keys works is that the bitcoin client initially used uncompressed keys, but starting from bitcoin core client 0.6, compressed keys are used as the standard.

Keys are identified by various prefixes, described as follows:

Uncompressed public keys used 0x04 as the prefix
Compressed public key starts with 0x03 if the y 32-bit part of the public key is odd
Compressed public key starts with 0x02 if the y 32-bit part of the public key is even

The more detailed mathematical description and the reason why it works is described here. If the ECC graph is visualized, it reveals that the *y* coordinate can be either below the *x* axis or above the *x* axis and as the curve is symmetric, only the location in the prime field is required to be stored.

Private keys in bitcoin

Private keys are basically 256-bit numbers chosen in the range specified by the SECP256K1 ECDSA recommendation. Any randomly chosen 256-bit number from 0x1 to 0xFFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFE BAAE DCE6 AF48 A03B BFD2 5E8C D036 4140 is a valid private key.

Private keys are usually encoded using **Wallet Import Format** (**WIF**) in order to make them easier to copy and use. WIF can be converted into private key and vice versa. The steps are described here.

Also, **Mini Private Key Format** is sometimes used to encode the key in under 30 characters in order to allow storage where physical space is limited, for

example, etching on physical coins or damage- resistant QR codes. The bitcoin core client also allows the encryption of the wallet that contains the private keys.

Bitcoin currency units

Bitcoin currency units are described as follows. The smallest bitcoin denomination is the Satoshi.

Base58Check encoding

DENOMINATION	ABBREVIATION	FAMILIAR NAME	VALUE IN BTC
Satoshi	SAT	Satoshi	0.00000001 BTC
Microbit	μBTC (uBTC)	Microbitcoin or Bit	0.000001 BTC
Millibit	mBTC	Millibitcoin	0.001 BTC
Centibit	cBTC	Centibitcoin	0.01 BTC
Decibit	dBTC	Decibitcoin	0.1 BTC
Bitcoin	BTC	Bitcoin	1 BTC
DecaBit	daBTC	Decabitcoin	10 BTC
Hectobit	hBTC	Hectobitcoin	100 BTC
Kilobit	kBTC	Kilobitcoin	1000 BTC
Megabit	MBTC	Megabitcoin	1000000 BTC

This encoding is used to limit the confusion between various characters, such as 0OIl as they can look the same in different fonts. The encoding basically takes the binary byte arrays and converts them into human-readable strings. This string is composed by utlilizing a set of 58 alphanumeric symbols. More explanation and logic can be found in the base58.hsource file in the bitcoin source code.

```
6   /**
7    * Why base-58 instead of standard base-64 encoding?
8    * - Don't want 0OIl characters that look the same in some fonts and
9    *      could be used to create visually identical looking data.
10   * - A string with non-alphanumeric characters is not as easily accepted as input.
11   * - E-mail usually won't line-break if there's no punctuation to break at.
12   * - Double-clicking selects the whole string as one word if it's all alphanumeric.
13   */
14  #ifndef BITCOIN_BASE58_H
```

Explanation from the bitcoin source code

Bitcoin addresses are encoded using the Base58check encoding.

Vanity addresses

As bitcoin addresses are based on base 58 encoding, it is possible to generate addresses that contain human-readable messages. An example is shown as follows:

Public address encoded in QR

Vanity addresses are generated using a purely brute-force method. An example is shown in the following screenshot:

Vanity address generated from https://bitcoinvanitygen.com/

Transactions

Transactions are at the core of the bitcoin ecosystem. Transactions can be as simple as just sending some bitcoins to a bitcoin address, or it can be quite complex depending on the requirements. Each transaction is composed of at least one input and output. Inputs can be thought of as coins being spent that have been created in a previous transaction and outputs as coins being created. If a transaction is minting new coins, then there is no input and therefore no

signature is needed. If a transaction is to sends coins to some other user (a bitcoin address), then it needs to be signed by the sender with their private key and a reference is also required to the previous transaction in order to show the origin of the coins. Coins are, in fact, unspent transaction outputs represented in Satoshis.

Transactions are not encrypted and are publicly visible in the blockchain. Blocks are made up of transactions and these can be viewed using any online blockchain explorer.

The transaction life cycle

1. A user/sender sends a transaction using wallet software or some other interface.
2. The wallet software signs the transaction using the sender's private key.
3. The transaction is broadcasted to the Bitcoin network using a flooding algorithm.
4. Mining nodes include this transaction in the next block to be mined.
5. Mining starts once a miner who solves the Proof of Work problem broadcasts the newly mined block to the network. Proof of Work is explained in detail later in this chapter.
6. The nodes verify the block and propagate the block further, and confirmation starts to generate.
7. Finally, the confirmations start to appear in the receiver's wallet and after approximately six confirmations, the transaction is considered finalized and confirmed. However, six is just a recommended number; the transaction can be considered final even after the first confirmation. The key idea behind waiting for six confirmations is that the probability of double spending is virtually eliminated after six confirmations.

The transaction structure

A transaction at a high level contains metadata, inputs, and outputs. Transactions are combined to create a block. The transaction structure is shown in the following table:

Field	**Size**	**Description**
Version Number	4 bytes	Used to specify rules to be used by the miners and nodes for transaction processing.

Input counter	1 bytes -9 bytes	The number of inputs included in the transaction.
list of inputs	variable	Each input is composed of several fields, including Previous transaction hash, Previous Txout-index, Txin-script length, Txin-script, and optional sequence number. The first transaction in a block is also called a coinbase transaction. It specifies one or more transaction inputs.
	1 bytes - 9 bytes	A positive integer representing the number of outputs.
uts	varia	Outputs included in the transaction.
lme		This defines the earliest time when a transaction becomes valid. It is either a Unix timestamp or a block number.

- **MetaData**: This part of the transaction contains some values such as the size of the transaction, the number of inputs and outputs, the hash of the transaction, and a lock_time field. Every transaction has a prefix specifying the version number.
- **Inputs**: Generally, each input spends a previous output. Each output is considered an **Unspent Transaction Output** (**UTXO**) until an input consumes it.
- **Outputs**: Outputs have only two fields, and they contain instructions for the sending of bitcoins. The first field contains the amount of Satoshis, whereas the second field is a locking script that contains the conditions that need to be met in order for the output to be spent. More information on transaction spending using locking and unlocking scripts and producing outputs is discussed later in this section.
- **Verification:** Verification is performed using bitcoin's scripting language.

A sample transaction is shown as follows:


```
{
  "txid": "08af7960ca9255c67686296fb65452ed3f96f18831c9a3d8ea552e4ccee5c4af",
  "hash": "08af7960ca9255c67686296fb65452ed3f96f18831c9a3d8ea552e4ccee5c4af",
  "size": 226,
  "vsize": 226,
  "version": 1,
  "locktime": 969523,
  "vin": [
    {
 "txid": "3e553260a0a94860f7043eb6576e15e6cfeb2990aea961210ae1fde328bb08b0",
 "vout": 1,
 "scriptSig": {
 "asm":
"3045022100cfb31edabc62c82b41d12f651d2e3e013ee1a7ee2bb4526f3dda640e6d8d224502207d8d1d8e41350b9cdf36f389f942ab68c12
f113fe99014f5d6df6610407877d2[ALL] 037bc82d0078993f6943e7ff6e82e82da600f34edc8bca136331a9901c8bb60b0d",
 "hex":
"483045022100cfb31edabc62c82b41d12f651d2e3e013ee1a7ee2bb4526f3dda640e6d8d224502207d8d1d8e41350b9cdf36f389f942ab68c
12f113fe99014f5d6df6610407877d20121037bc82d0078993f6943e7ff6e82e82da600f34edc8bca136331a9901c8bb60b0d"
      },
      "sequence": 4294967294
    }
  ],
  "vout": [
    {
      "value": 2.30000000,
      "n": 0,
      "scriptPubKey": {
        "asm": "OP_DUP OP_HASH160 07e78644a61343068fa8d4940a79976e758ac6ef OP_EQUALVERIFY OP_CHECKSIG",
        "hex": "76a91407e78644a61343068fa8d4940a79976e758ac6ef88ac",
        "reqSigs": 1,
        "type": "pubkeyhash",
        "addresses": [
          "mgEkNzxV3qbYtDKEKTvo1VpgJ63Au619q2"
}}}
```


A sample decoded transaction, showing various fields described earlier

The script language

Bitcoin uses a simple stack-based language called *script* to describe how bitcoins can be spent and transferred. It is not Turing complete and has no loops to avoid any undesirable effects of long running/hung scripts on the bitcoin network. This scripting language is based on a Forth-like syntax and uses a reverse polish notation in which every operand is followed by its operators. It is evaluated from the left to the right using a **Last in First Out** (**LIFO**) stack.

Scripts use various Opcodes or instructions to define their operation. Opcodes are also known as words, commands, or functions. Earlier versions of the bitcoin node had a few Opcodes that are no longer used due to bugs discovered in their design.

The various categories of the scripting Opcodes are constants, flow control, stack, bitwise logic, splice, arithmetic, cryptography and lock time.

A transaction script is evaluated by combining ScriptSig and ScriptPubKey. ScriptSig is the unlocking script, whereas ScriptPubKeyis the locking script.

This is how a transaction is evaluated to be spent; first, it is unlocked and then it is spent. ScriptSig is provided by the user who wishes to unlock the transaction. ScriptPubkey is part of the transaction output and specifies the conditions that need to be fulfilled in order to spend the output. In other words, outputs are locked by the ScriptPubKey(Locking script) that contains the conditions, when met will unlock the output, and coins can then be redeemed.

Commonly used Opcodes

All Opcodes are declared in the script.h file in the bitcoin reference client source code. This can be accessed from the link at https://github.com/bitcoin/bitcoin/blob/master/src/script/script.h under the following comment:

/ Script opcodes */**

A description of the most commonly used Opcodes is listed here. This table is taken from the bitcoin developer's guide:

Opcode	Description
OP_CHECKSIG	This takes a public key and signature and validates the signature of the hash of the transaction. If it matches, then TRUE is pushed onto the stack; otherwise, FALSE is pushed.
OP_EQUAL	This returns 1 if the inputs are exactly equal; otherwise, 0 is returned.
OP_DUP	This duplicates the top item in the stack.
OP_HASH160	The input is hashed twice, first with SHA-256 and then with RIPEMD-160.
OP_VERIFY	This marks the transaction as invalid if the top stack value is not true.
OP_EQUALVERIFY	This is the same as OP_EQUAL, but it runs OP_VERIFYafterwards.
OP_CHECKMULTISIG	This takes the first signature and compares it against each public key until a match is found and repeats this process until all signatures are checked. If all signatures turn out to be valid, then a value of 1 is returned as a result; otherwise, 0 is returned.

Types of transaction

There are various scripts available in bitcoin to handle the value transfer from the source to the destination. These scripts range from very simple to quite complex depending upon the requirements of the transaction. Standard transaction types are discussed here. Standard transactions are evaluated using IsStandard()and IsStandardTx()tests and only standard transactions that pass the test are generally allowed to be mined or broadcasted on the bitcoin network. However, nonstandard transactions are valid and allowed on the network.

- **Pay to Public Key Hash** (**P2PKH**): P2PKH is the most commonly used transaction type and is used to send transactions to the bitcoin addresses. The format of the transaction is shown as folows:

  ScriptPubKey: OP_DUP OP_HASH160 <pubKeyHash>

  OP_EQUALVERIFY OP_CHECKSIG ScriptSig: <sig> <pubKey>

  The ScriptPubKey and ScriptSig parameters are concatenated together and executed. An example will follow shortly in this section, where this is explained in more detail.
- **Pay to Script Hash** (**P2SH**): P2SH is used in order to send transactions to a script hash (that is, the addresses starting with 3) and was standardized in BIP16. In addition to passing the script, the redeem script is also evaluated and must be valid. The template is shown as follows:

  ScriptPubKey: OP_HASH160 <redeemScriptHash> OP_EQUAL ScriptSig:

  [<sig>...<sign>] <redeemScript>
- **MultiSig (Pay to MultiSig)**: M of n multisignature transaction script is a complex type of script where it is possible to construct a script that required multiple signatures to be valid in order to redeem a transaction. Various complex transactions such as escrow and deposits can be built using this script. The template is shown here:

  ScriptPubKey: <m> <pubKey> [<pubKey> . . .] <n>

  OP_CHECKMULTISIG ScriptSig: 0 [<sig > . . . <sign>]

  Raw multisig is obsolete, and multisig is usually part of the P2SH redeem script, mentioned in the previous bullet point.

- **Pay to Pubkey**: This script is a very simple script that is commonly used in coinbase transactions. It is now obsolete and was used in an old version of bitcoin. The public key is stored within the script in this case, and the unlocking script is required to sign the transaction with the private key.

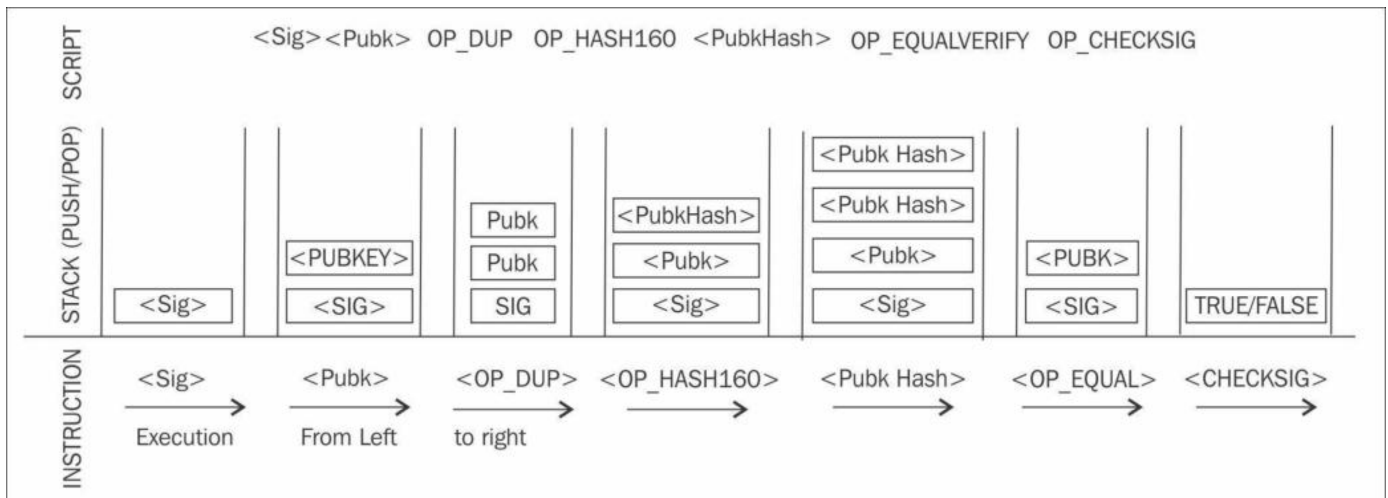
  The template is shown as follows:

  <PubKey> OP_CHECKSIG
- **Null data/OP_RETURN**: This script is used to store arbitrary data on the blockchain for a fee.
  The limit of the message is 40 bytes. The output of this script is unredeemable because
  OP_RETURNwill fail the validation in any case. ScriptSigis not required in

  this case. The template is very simple and is shown as follows:

  OP_RETURN<data>

  A P2PKH script execution is shown as follows:



P2PKH script execution

All transactions are eventually encoded into the hex before transmitting over the bitcoin network. A sample transaction is shown in hex that is retrieved using bitcoin-cli on the bitcoin testnet by using the following command:

drequinox@drequinox-OP7010:~$ bitcoin-cli --testnet getrawtransaction "08af7960ca9255c67686296fb65452ed3f96f18831c9a3d8ea552e4ccee5c4af

"
dad770cccb1026ebf87acacfe35f2d6f2d336faa88ac33cb0e00

Coinbase transactions

A coinbase transaction or generation transaction is always created by a miner and is the first transaction in a block. It is used to create new coins. It includes a special field, also called *coinbase*, which acts as an input to the coinbase transaction. This transaction also allows up to 100 bytes of arbitrary data that can be used to store arbitrary data.

What is UTXO

Unspent Transaction Output (**UTXO**) is an unspent transaction output that can be spent as an input to a new transaction. Other concepts related to transactions in bitcoin are described below.

Transaction fee

Transaction fees are charged by the miners. The fee charged is dependent upon the size of the transaction. Transaction fees are calculated by subtracting the sum of the inputs and the sum of the outputs. The fees are used as an incentive for miners to encourage them to include a user transaction in the block the miners are creating. All transactions end up in the memory pool, from where miners pick up transactions based on their priority to include them in the proposed block. The calculation of priority is introduced later in this chapter; however, from a transaction fee point of view, a transaction with a higher fee will be picked up sooner by the miners. There are different rules based on which fee is calculated for various types of actions, such as sending transactions, inclusion in blocks, and relaying by nodes. Fees are not fixed by the Bitcoin protocol and are not mandatory; even a transaction with no fee will be processed in due course but may take a very long time.

Contracts

As defined in the bitcoin core developer guide, contracts are basically transactions that use the bitcoin system to enforce a financial agreement. This is a simple definition but has far-reaching consequences as it allows users to design complex contracts that can be used in many real-world scenarios. Contracts allow the development of a completely decentralized, independent, and reduced risk platform. Various contracts, such as escrow, arbitration, and micropayment channels, can be built using the bitcoin scripting language. The current implementation of a script is very limited, but various types of contracts are still possible to develop. For example, the release of funds only if multiple parties sign the transaction or perhaps the release of funds only

after a certain time has elapsed. Both of these scenarios can be realized using multiSigand transaction lock time options.

Transaction malleability

Transaction malleability in bitcoin was introduced due to a bug in the bitcoin implementation. Due to this bug, it becomes possible for an adversary to change the Transaction ID of a transaction, thus resulting in a scenario where it would appear that a certain transaction has not been executed. This can allow scenarios where double deposits or withdrawals can occur. In other words, this bug allows the changing of the unique ID of a bitcoin transaction before it is confirmed.

If the ID is changed before confirmation, it would seem that the transaction did not happen at all, which can then allow double deposits or withdrawal attacks.

Transaction pools

Also known as *memory pools*, these pools are basically created in local memory by nodes in order to maintain a temporary list of transactions that are not yet confirmed in a block. Transactions are included in a block after passing verification and based on their priority.

Transaction verification

This verification process is performed by bitcoin nodes. The following is described in the bitcoin developer guide:

1. Check the syntax and ensure that the syntax of the transaction is correct.
2. Verify that inputs and outputs are not empty.
3. Check whether the size in bytes is less than the maximum block size, which is 1 MB currently.
4. The output value must be in the allowed money range (0 to 21 million BTC).
5. All inputs must have a specified previous output, except for coinbase transactions, which should not be relayed.
6. Verify that nLockTime must not exceed 31-bits. For a transaction to be valid, it should not be less than 100 bytes. Also, the number of signature operands in a standard signature should be less than or not more than 2.
7. Reject *nonstandard* transactions; for example, ScriptSig is allowed to only push numbers on the stack. ScriptPubkeynot passing the isStandard() checks.
8. A transaction is rejected if there is already a matching transaction in the pool or in a block in the main branch.
9. The transaction will be rejected if the referenced output for each input exists in any other transaction in the pool.

10. For each input, there must exist a referenced output transaction. This is searched in the main branch and the transaction pool to find whether the output transaction is missing for any input, and this will be considered an orphan transaction. It will be added to the orphan transactions pool if a matching transaction is not in the pool already.
11. For each input, if the referenced output transaction is the coinbase, it must have at least 100 confirmations; otherwise, the transaction will be rejected.
12. For each input, if the referenced output does not exist or has been spent already, the transaction will be rejected.
13. Using the referenced output transactions to get input values, verify that each input value, as well as the sum, is in the allowed range of 0-21 million BTC.
14. Reject the transaction if the sum of input values is less than the sum of output values.
15. Reject the transaction if the transaction fee would be too low to get into an empty block.

Blockchain

Blockchain is a public ledger of a timestamped, ordered, and immutable list of all transactions on the bitcoin network. Each block is identified by a hash in the chain and is linked to its previous block by referencing the previous block's hash.

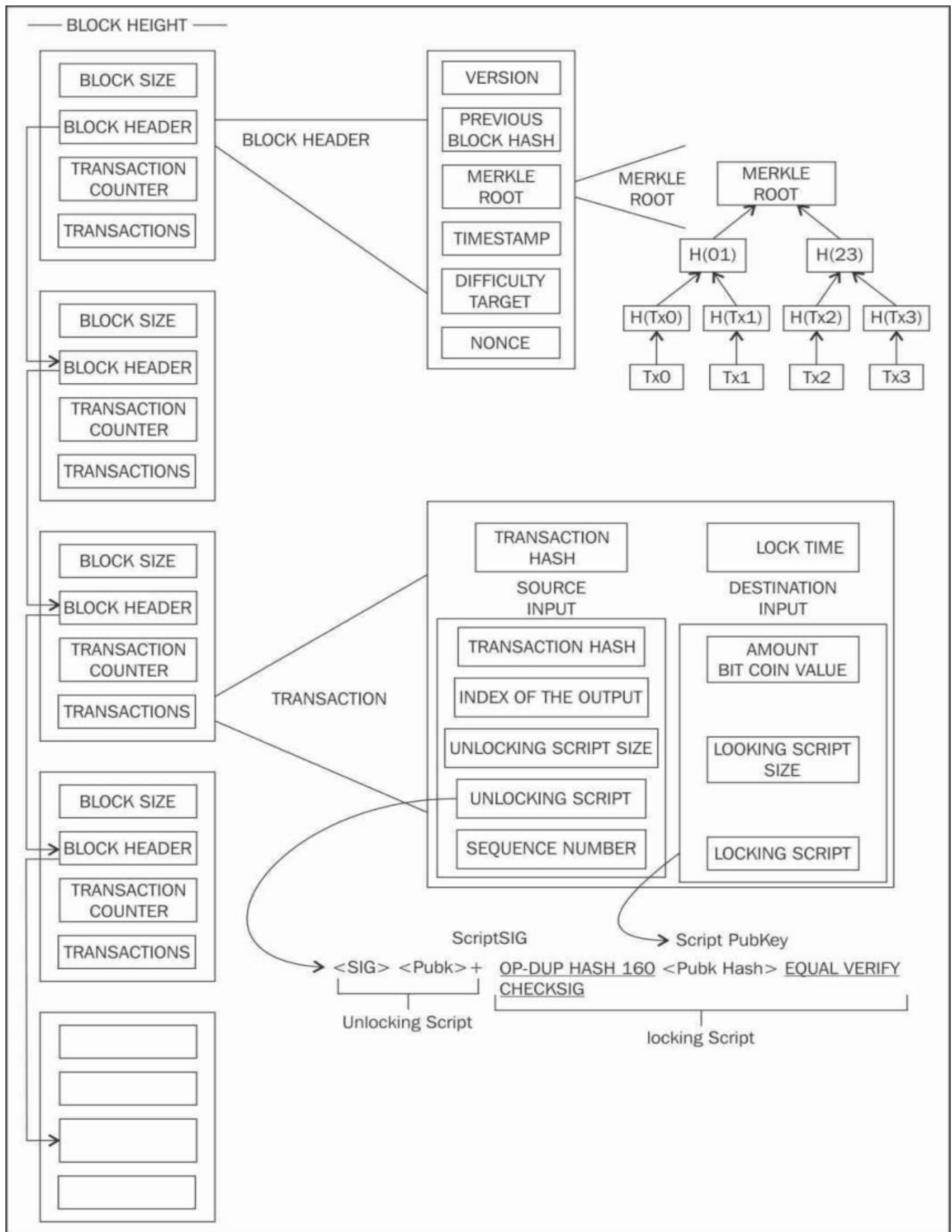
In the following structure of a block, a block header is described, followed by a detailed diagram that provides an insight into the blockchain structure.

The structure of a block

Bytes	Name	Description
80	Block header	This includes fields from the block header described in the next section.
variable	Transaction counter	The field contains the total number of transactions in the block, including the coinbase transaction.
variable	Transactions	All transactions in the block.

The structure of a block header

Bytes	Name	Description
4	Version	The block version number that dictates the block validation rules to follow.
32	previous block header hash	This is a double SHA256 hash of the previous block's header.
32	Merkleroot hash	This is a double SHA256 hash of the merkle tree of all transactions included in the block.
4	Timestamp	This field contains the approximate creation time of the block in the Unix epoch time format. More precisely, this is the time when the miner has started hashing the header (the time from the miner's point of view).
4	Difficulty target	This is the difficulty target of the block.
4	Nonce	This is an arbitrary number that miners change repeatedly in order to produce a hash that fulfills the difficulty target threshold.



BLOCK HEIGHT
BLOCK SIZE
BLOCK HEADER
TRANSACTION COUNTER
TRANSACTIONS
BLOCK HEADER
VERSION
PREVIOUS BLOCK HASH
MERKLE ROOT
TIMESTAMP
DIFFICULTY TARGET
NONCE
MERKLE ROOT
MERKLE ROOT
H(01)
H(23)
H(Tx0)
H(Tx1)
H(Tx2)
H(Tx3)
Tx0
Tx1
Tx2
Tx3
TRANSACTION
TRANSACTION HASH
LOCK TIME
SOURCE INPUT
DESTINATION INPUT
TRANSACTION HASH
INDEX OF THE OUTPUT
UNLOCKING SCRIPT SIZE
UNLOCKING SCRIPT
SEQUENCE NUMBER
AMOUNT BIT COIN VALUE
LOOKING SCRIPT SIZE
LOCKING SCRIPT
ScriptSIG
Script PubKey
<SIG> <Pubk>+
OP-DUP HASH 160 <Pubk Hash> EQUAL VERIFY CHECKSIG
Unlocking Script
locking Script

A visualization of blockchain, block, block header, transaction and script

As shown in the preceding diagram, blockchain is a chain of blocks where each block is linked to its previous block by referencing the previous block header's hash. This linking makes sure that no transaction can be modified unless the block that records it and all blocks that follow it are also modified. The first block is not linked to any previous block and is known as the genesis block.

The genesis block

This is the first block in the bitcoin blockchain. The genesis block was hardcoded in the bitcoin core software. It is in the chainparams.cpp file.

```
48   *     CTxOut(nValue=50.00000000, scriptPubKey=0x5F1DF16B2B704C8A578D0B)
49   *   vMerkleTree: 4a5e1e
50   */
51  static CBlock CreateGenesisBlock(uint32_t nTime, uint32_t nNonce, uint32_t nBits, int32_t nVersion, const CAmount& genesisReward)
52  {
53      const char* pszTimestamp = "The Times 03/Jan/2009 Chancellor on brink of second bailout for banks";
54      const CScript genesisOutputScript = CScript() << ParseHex("04678afdb0fe5548271967f1a67130b7105cd6a828e03909a67962e0ea1f61deb649
55      return CreateGenesisBlock(pszTimestamp, genesisOutputScript, nTime, nNonce, nBits, nVersion, genesisReward);
56  }
57
58  /**
```

Bitcoin provides protection against double spending by enforcing strict rules on transaction verification and via mining. Blocks are added in the blockchain only after strict rule checking and successful Proof of Work solution. Block height is the number of blocks before a particular block in the blockchain. The current height (at the time of writing this) of the blockchain is 434755 blocks. Proof of Work is used to secure the blockchain. Each block contains one or more transactions, out of which the first transaction is a coinbase transaction. There is a special condition for coinbase transactions that prevent them to be spent until at least 100 blocks in order to avoid a situation where the block may be declared stale later on.

Stale blocks are created when a block is solved and every other miner who is still working to find a solution to the hash puzzle is working on that block. Mining and hash puzzles will be discussed later in the chapter in detail. As the block is no longer required to be worked on, this is considered a stale block.

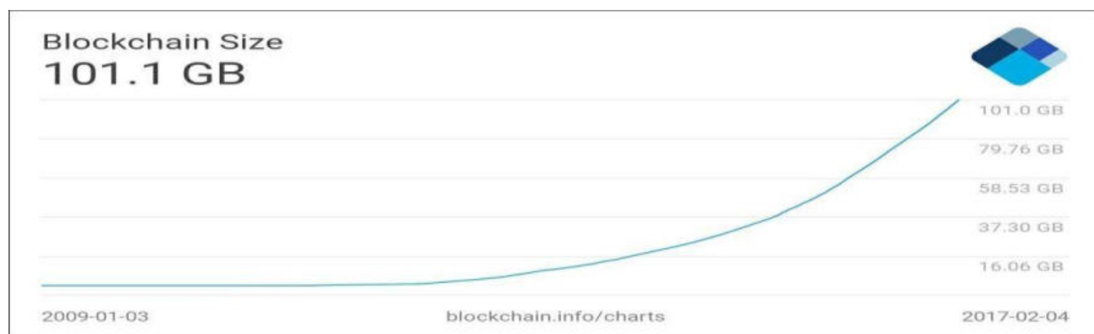
Orphan blocks are also called detached blocks and were accepted at one point in time by the network as valid blocks but were rejected when a proven longer chain was created that did not include this initially accepted block. They are not part of the main chain and can occur at times when two miners manage to produce the blocks at the same time.

The latest block version is version 4, which was proposed with BIP65 and has been used since bitcoin core client 0.11.2 since the implementation of BIP9 bits in nVersionfield are being used to indicate softfork changes.

Because of the distributed nature of bitcoin, network forks can occurs naturally. In cases where two nodes simultaneously announce a valid b lock can result in a situation where there are two blockchains with different transactions. This is an undesirable situation but can be addressed by the bitcoin network only by accepting the longest chain. In this case, the smaller chain will be considered orphaned. If an adversary manages to gain 51% control of the network hashrate (computational power), then they can impose their own version of transaction history.

Forks in blockchain can occur with the introduction of changes in the Bitcoin protocol. In case of *soft fork*, only previous valid blocks are no longer acceptable, thus making soft fork backward compatible. In case of soft fork, only miners are required to upgrade to the new client software in order to make use of the new protocol rules. Planned upgrades do not necessarily create forks because all users should have updated already. A hard fork, on the other hand, invalidates previously valid blocks and requires all users to upgrade. New transaction types are sometimes added as a soft fork, and any changes such as block structure change or major protocol changes results in hard fork.

The current size of the bitcoin blockchain as on February 4, 2017, stands at approximately 101 GB. The following figure shows the size increase of blockchain as a function of time:



New blocks are added to the blockchain approximately every 10 minutes and network difficulty is adjusted dynamically every 2016 blocks in order to maintain a steady addition of new blocks to the network.

Network difficulty is calculated using the following equation:

*Target = Previous target * Time/2016 * 10 minutes*

Difficulty and target are interchangeable and represent the same thing. Previous target represents the old target value, and time is the time spent to generate previous 2016 blocks. Network difficulty basically means how hard it is for miners to find a new block, that is, how difficult the hashing puzzle is now.

Mining

Mining is a resource-intensive process by which new blocks are added to the blockchain. Blocks contain transactions that are validated via the mining process by mining nodes and are added to the blockchain. This process is resource-intensive in order to ensure that the required resources have been spent by miners in order for a block to be accepted. New coins are minted by the miners by spending the required computing resources. This also secures the system against frauds and double spending attacks while adding more virtual currency to the bitcoin ecosystem.

Roughly one new block is created (mined) every 10 minute. Miners are rewarded with new coins if and when they create new blocks and are paid transaction fees in return of including transactions in their blocks. New blocks are created at an approximate fixed rate. Also, the rate of creation of new bitcoins decreases by 50%, every 210,000 blocks, roughly every 4 years. When bitcoin was initially introduced, the block reward was 50 bitcoins; then in 2012, this was reduced to 25 bitcoins. In July 2016, this was further reduced to 12.5 coins (12 coins) and the next reduction is estimated to be on July 4, 2020. This will reduce the coin reward further down to approximately six coins.

Approximately 144 blocks, that is, 1,728 bitcoins are generated per day. The number of actual coins can vary per day; however, the number of blocks remains at 144 per day. Bitcoin supply is also limited and in 2140, almost 21 million bitcoins will be finally created and no new bitcoins can be created after that. Bitcoin miners, however, will still be able to profit from the ecosystem by charging transaction fees.

Task of miners

Once a node connects with the bitcoin network, there are several tasks that a bitcoin miner performs.

Synching up with the network

Once a new node joins the bitcoin network, it downloads the blockchain by requesting historical blocks from other nodes. This is mentioned here in the context of the bitcoin miner; however, this not necessarily a task only for a miner.

- **Transaction validation**: Transactions broadcasted on the network are validated by full nodes by verifying and validating signatures and outputs.
- **Block validation**: Miners and full nodes can start validating blocks received by them by evaluating them against certain rules. This includes the verification of each transaction in the block along with verification of the nonce value.
- **Create a new block**: Miners propose a new block by combining transactions broadcasted on the network after validating them.
- **Perform Proof of Work**: This task is the core of the mining process and this is where miners find a valid block by solving a computational puzzle. The block header contains a 32-bit nonce field and miners are required to repeatedly vary the nonce until the resultant hash is less than a predetermined target.
- **Fetch reward**: Once a node solves the hash puzzle, it immediately broadcasts the results, and other nodes verify it and accept the block. There is a slight chance that the newly minted block will not be accepted by other miners due to a clash with another block found at roughly the same time, but once accepted, the miner is rewarded with 12.5 bitcoins (as of 2016) and any associated transaction fees.

Proof of Work

This is a proof that enough computational resources have been spent in order to build a valid block. **Proof of Work** (**PoW**) is based on the idea that a random node is selected every time to create a new block. In this model, nodes compete with each other in order to be selected in proportion to their computing capacity. The following equation sums up the Proof of Work requirement in bitcoin:

$$H(N \| P_hash \| Tx \| Tx \| \ldots Tx) < Target$$

Where N is a nonce, *P_hash* is a hash of the previous block, *Tx* represents transactions in the block, and *Target* is the target network difficulty value. This

means that the hash of the previously mentioned concatenated fields should be less than the target hash value. The only way to find this nonce is the brute force method. Once a certain pattern of a certain number of zeroes is met by a miner, the block is immediately broadcasted and accepted by other miners.

The mining algorithm

The mining algorithm consists of the following steps.

- The previous hash block is retrieved from the bitcoin network.
- Assemble a set of potential transactions broadcasted on the network into a block.
- Compute the double hash of the block header with a nonce and the previous hash using the SHA256 algorithm.
- If the resultant hash is lower than the current difficulty level (target), then stop the process.
- If the resultant hash is greater than the current difficulty level (target), then repeat the process by incrementing the nonce. As the hash rate of the bitcoin network increased, the total amount of 32- bit nonces was exhausted too quickly. In order to address this issue, the *extra nonce* solution was implemented, whereby the coinbase transaction is used as a source of extra nonce to provide a larger range of nonces to be searched by the miners.

Mining difficulty increased over time and bitcoins that could be mined by single CPU laptop computers now require dedicated mining centers to solve the hash puzzle. The current difficulty level can be queried using the bitcoin command line

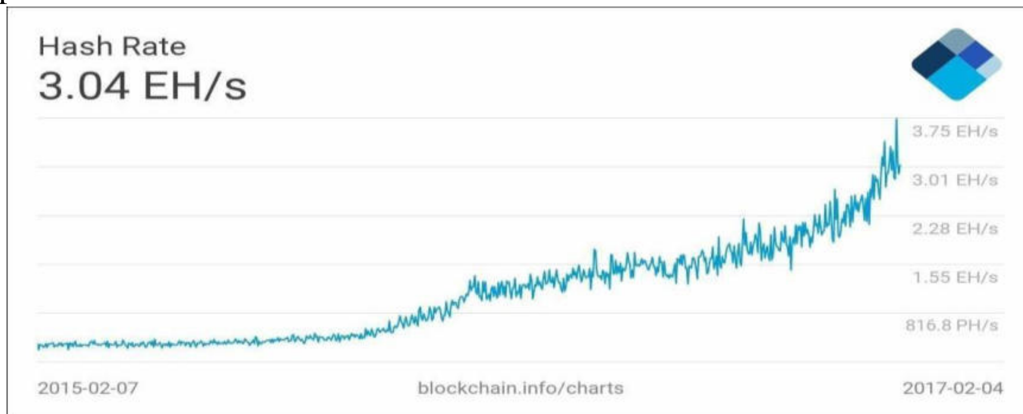
- interface using the following command:

$ bitcoin-cli getdifficulty
258522748404.5154

Mining difficulty over time. The value returned by the get difficultycommand.

The hashing rate

The hashing rate basically represents the rate of calculating hashes per second. In early days of bitcoin, it used to be quite small as CPUs were used, but with dedicated mining pools and ASICs now, this has gone up exponentially in the last few years. This has resulted in increased difficulty. The following hash rate graph shows the hash rate increase over time and is currently measured in Exa hashes. This means that in 1 second, bitcoin network miners are computing more than 1 000 000 000 000 000 000 hashes per second.



Mining systems

Over time, bitcoin miners have used various methods to mine bitcoins. As the core principle behind mining is based on the double SHA256 algorithm, overtime miners have developed sophisticated systems to calculate the hash faster and faster.

CPU

CPU mining was the first type of mining available in the original bitcoin client. Users could even use laptop or desktop computers to mine bitcoins. CPU mining is no longer profitable and now more advanced mining methods such as ASIC-based mining are used.

GPU

Due to the increased difficulty of the bitcoin network and general tendency of finding faster methods to mine, miners started to use GPUs or graphics cards available in PCs to perform mining. GPUs support faster and parallelized calculations that are usually programmed using the OpenCL language. This turned out to be a faster option as compared to CPUs. Users also used techniques such as overclocking to gain maximum benefit of the GPU power.

Also, the possibility of using multiple graphics cards increased the popularity of graphics cards' usage for bitcoin mining. GPU mining, however, has some limitations, such as overheating and the requirement for specialized motherboards and extra hardware to house multiple graphics cards.

FPGA

Even GPU mining did not last long, and soon miners found another way to perform mining using FPGAs. **Field Programmable Gate Array** (**FPGA**) is basically an integrated circuit that can be programmed to perform specific operations. FPGAs are usually programmed in **hardware description languages** (**HDLs**), such as Verilog and VHDL. Double SHA256 quickly became an attractive programming task for FPGA programmers and several open source projects started too. FPGA offered much better performance as compared to GPUs; however, issues such as accessibility, programming difficulty, and the requirement for specialized knowledge to program and configure FPGAs resulted in a short life of the FPGA era for bitcoin mining. Also, the arrival of ASICs resulted in quickly phased out FPGA-based systems for mining. Mining hardware such as X6500 miner, Ztex, and Icarus were developed during the time when FPGA mining was profitable. Various FPGA manufacturers, such as Xilinx and Altera, produce FPGA hardware and development boards that can be used to program mining algorithms.

ASICs

Application Specific Integrated Circuit (**ASIC**) was designed to perform the SHA-256 operation. These special chips were sold by various manufacturers and offered a very high hashing rate. This worked for some time, but due to the quickly increasing mining difficulty level, single-unit ASICs are no longer profitable.

Currently, mining is out of the reach of individuals and now professional mining centers using thousands of ASIC units in parallel are offering mining contracts to users to perform mining on their behalf. There is no technical limitation, that's why a single user cannot run thousands of ASICs in parallel, but it will require dedicated data centers and hardware and cost for a single individual can become prohibitive.

Four types of mining (CPU, GPU, FPGA, and ASIC)

Mining pools

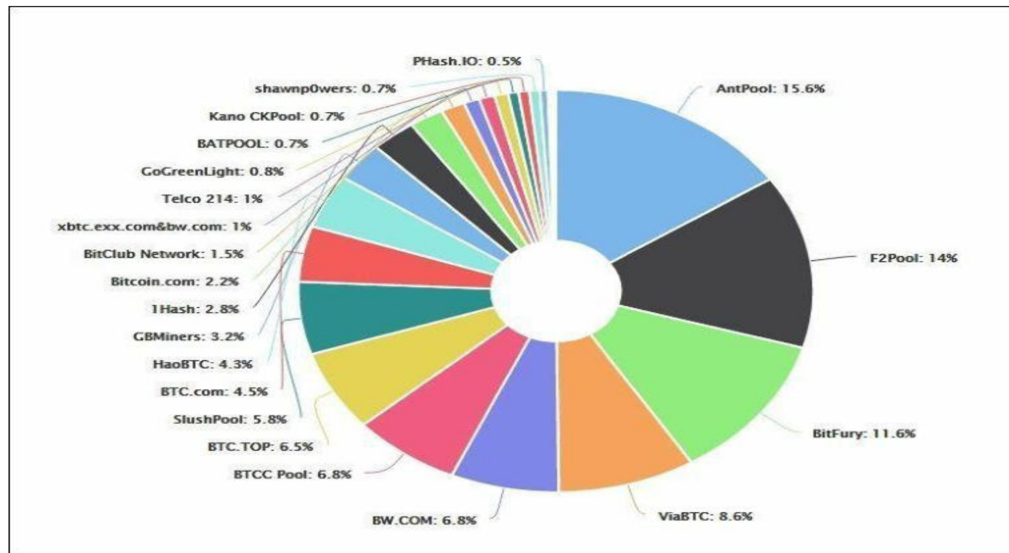
A mining pool forms when group miners work together to mine a block. The *Pool manager* receives the coinbase transaction if the block is successfully mined, which is then responsible for distributing the reward to the group of miners who invested resources to mine the block. This is profitable as

compared to solo mining, where only one sole miner is trying to solve the partial hash inversion function (hash puzzle) because in mining pools, the reward is paid to each member of the pool regardless of whether they (more specifically, their individual node) solved the puzzle or not.

There are various models that a mining pool manager can use to pay to the miners, such as the pay- per-share model and the proportional model. In the pay per share model, the mining pool manager pays a flat fee to all miners who participated in the mining exercise, whereas in the proportional model, the share is calculated based on the amount of computing resources spent to solve the hash puzzle.

Many commercial pools now exist and provide mining service contracts via the cloud and easy-to- use web interfaces. The most commonly used ones are **AntPool**, **F2Pool**, and **BW.COM**. A

comparison of hashing power for all major mining pools is shown in the following image:



Mining centralization is a major concern that can occur if a pool manages to control more than 51% of the network by generating more than 51% hash rate of the bitcoin network. As discussed earlier in the introduction section, 51% attack can result in double spending attacks, and it can impact consensus and in fact impose another version of transaction history on the bitcoin network.

This has happened once in the bitcoin history, when GHash.IO, a large mining pool, managed to acquire more than 51% of the network capacity. Theoretical solutions , such as two-phase Proof of Work, have been proposed in academia

to disincentivize large mining pools. This scheme introduces a second cryptographic puzzle that results in mining pools to reveal their private keys or providing a considerable portion of the hashrate of their mining pool, thus reducing the overall hashrate of the pool.

Various types of hardware are commercially available for mining purposes. Currently, the most profitable one is ASIC mining, and specialized hardware is available from a number of vendors. Solo mining is not much profitable now unless a vast amount of money and energy is spent to build your own mining rig or even center. With the current difficulty factor (Oct 2016), if a user manages to produce a hash rate of 12 TH/s, they can hope to make 0.01366887 BTC (around $8) per day, which is very low as compared to the investment required to source the equipment that can produce 12 TH. Including running costs such as electricity, this turns out to be not very profitable.

The bitcoin network

The bitcoin network is a P2P network where nodes exchange transactions and blocks. There are different types of nodes on the network. There are two main types of nodes, full nodes and SPV nodes. Full nodes, as the name implies, are implementations of bitcoin core clients performing the wallet, miner, full blockchain storage, and network routing functions. However, it is not necessary to perform all these functions. SPV nodes or lightweight clients perform only wallet and network routing functionality. The latest version of Bitcoin protocol is 70014 and was introduced with bitcoin core client 0.13.0.

Some nodes prefer to be full blockchain nodes only and contain complete blockchain and perform network routing functions but do not perform mining or store private keys (the wallet function).
Another type is solo miner nodes that can perform mining, store full blockchain, and act as a bitcoin network routing node.

There are a few nonstandard but heavily used nodes that are called pool protocol servers. These nodes make use of alternative protocols, such as the stratum protocol. Some nodes perform only mining functions and are called mining nodes. Nodes that only compute hashes use the stratum protocol to submit their solutions to the mining pool. It is possible to run an SPV client runs a wallet and network routing function without a blockchain.

Most protocols on the Internet are line-based, which means that each line is delimited by a carriage return and newline *\r \n* character. Stratum is also a line-based protocol that makes use of plain TCP sockets and human-readable

JSON-RPC to operate and communicate between nodes.

Bitcoin network is identified by its different magic values. A list is shown as follows:

Network	Magic value	Hex
main	0xD9B4BEF9	F9 BE B4 D9
testnet3	0x0709110B	0B 11 09 07

Bitcoin network magic values

Magic values are used to indicate the message origin network.

A full node performs four functions: wallet, miner, blockchain, and the network routing node.

When a bitcoin core node starts up, first, it initiates the discovery of all peers. This is achieved by querying DNS seeds that are hardcoded into the bitcoin core client and are maintained by bitcoin community members. This lookup returns a number of DNS A records. The bitcoin protocol works on TCP port 8333 by default for the main network and TCP 18333 for testnet.

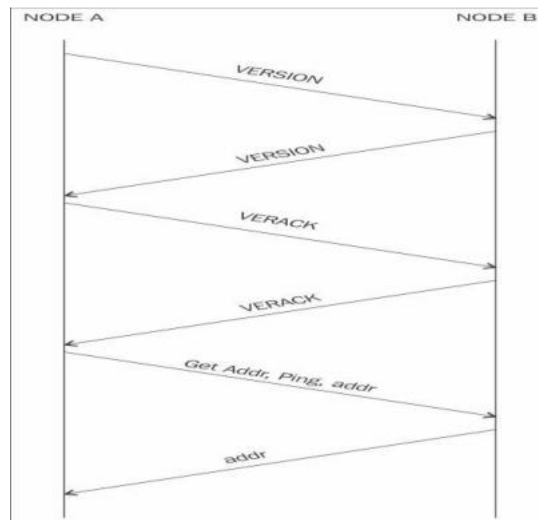
```
vSeeds.push_back(CDNSSeedData("bitcoin.sipa.be", "seed.bitcoin.sipa.be", true)); // Pieter Wuille
vSeeds.push_back(CDNSSeedData("bluematt.me", "dnsseed.bluematt.me")); // Matt Corallo
vSeeds.push_back(CDNSSeedData("dashjr.org", "dnsseed.bitcoin.dashjr.org")); // Luke Dashjr
vSeeds.push_back(CDNSSeedData("bitcoinstats.com", "seed.bitcoinstats.com")); // Christian Decker
vSeeds.push_back(CDNSSeedData("xf2.org", "bitseed.xf2.org")); // Jeff Garzik
vSeeds.push_back(CDNSSeedData("bitcoin.jonasschnelli.ch", "seed.bitcoin.jonasschnelli.ch", true)); // Jonas Schnelli
```

DNS Seeds in chainparams.cpp

First, the client sends a protocol message *Version* that contains various fields, such as version, services, timestamp, network address, nonce, and some other fields. The remote node responds with its own version message followed by verack message exchange between both nodes, indicating that the connection has been established.

After this, **G *etaddr*** and ***addr*** messages are exchanged to find the peers that the client do not know. Meanwhile, either of the nodes can send a ping message to see whether the connection is still live.

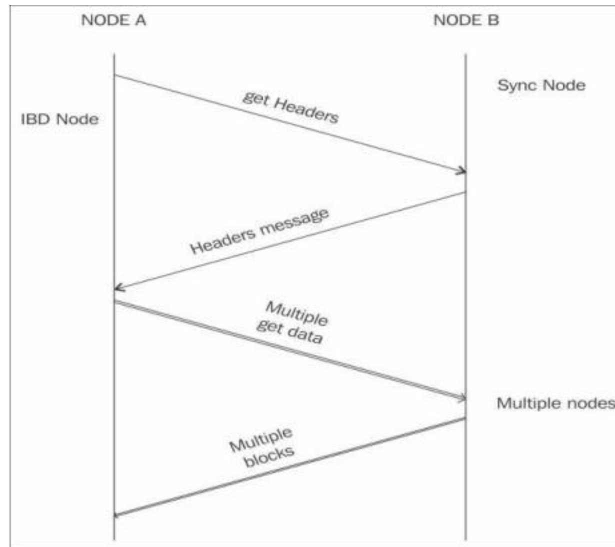
Now the block download can begin. If the node already has all blocks fully synchronized, then it listens for new blocks using the *Inv* protocol message; otherwise, it first checks whether it has a response to *inv* messages and have inventories already. If yes, then it requests the blocks using the ***Getdata*** protocol message; if not, then it requests inventories using the *GetBlocks* message. This method was used until version 0.9.3.



Protocol visualization node discovery

Initial block download can use blocks first or the headers-first method to synchronize blocks depending on the version of the bitcoin core client. The blocks-first method is very slow and was discontinued since version 0.10.0.

In this method, when the client starts up, it checks whether the block chain is fully synchronized already if the header chain is already synchronized; if not, which is the case the first time the client starts up, it requests headers from other peers using the ***getHeaders*** message. If the block chain is fully synchronized, it listens for new blocks via *Inv* messages, and if it already has a fully synchronized header chain, then it requests blocks using *Getdata* protocol messages. The node also checks whether the header chain has more headers than blocks and then it requests blocks by issuing the *Getdata* protocol message.



Bitcoin Core Client >= 0.10.0 header and block synchronization, IBD = Initial block download and sync node means the node from where the blocks are being requested from

Getblockchaininfo and getpeerinfo RPCs were updated with a new functionality to cater for this change. An RPC, *getchaintips*, is used to list all known branches of the blockchain. This also includes headers only blocks.

```
Filter: ip.dst == 52.1.165.219 and bitcoin          Expression... Clear Apply Save
No.   Time           Source        Destination    Protocol Length Info
  131 98.598526000   192.168.0.13  52.1.165.219   Bitcoin     192 version
  150 99.180294000   192.168.0.13  52.1.165.219   Bitcoin      90 verack
  151 99.180421000   192.168.0.13  52.1.165.219   Bitcoin     122 getaddr, ping
  152 99.180715000   192.168.0.13  52.1.165.219   Bitcoin    1288 addr, getheaders[Malformed Packet]
  486 112.053746000  192.168.0.13  52.1.165.219   Bitcoin     127 inv
  818 143.630367000  192.168.0.13  52.1.165.219   Bitcoin     127 inv
 1004 178.729768000  192.168.0.13  52.1.165.219   Bitcoin     127 inv

▸Transmission Control Protocol, Src Port: 52864 (52864), Dst Port: 18333 (18333), Seq: 207, Ack: 1291, Len: 1222
▾Bitcoin protocol
   Packet magic: 0x0b110907
   Command name: addr
   Payload Length: 31
   Payload checksum: 0xa03fc07d
 ▾Address message
    Count: 1
  ▾Address: afbd025800000000000000000000000000000000ffff...
   ▾Node services: 0x0000000000000000
      .... .... .... .... .... .... .... ...0 = Network node: Not set
     Node address: ::ffff:86.15.44.209 (::ffff:86.15.44.209)
     Node port: 18333
     Address timestamp: Oct 16, 2016 00:37:19.000000000 BST
▾Bitcoin protocol
   Packet magic: 0x0b110907
   Command name: getheaders
   Payload Length: 1029
   Payload checksum: 0x4e54961d
 ▾Getheaders message
    Count: 126
    Starting hash: 1101001f152142abccc039503abc56b149bd56c2b3925b65...
    Starting hash: 000000001980703bd53b0c7bf0ac995bccfeeffd5cddc780...
    Starting hash: 000000007ad1fed813d20301b1762895a2e5b08c8a58b3ea...
    Starting hash: 000000003624c451f726a3e983d02279d9c7cf672d36f1d5...
```

A protocol graph showing the flow of data between the two peers is shown here. This can help you understand when a node starts up and what type of messages are used.

In the following example, the bitcoin dissector is used to analyze the traffic and identify the Bitcoin protocol commands.

Exchange of messages such as the **version**, **getaddr**, and **getdata** can be seen in the following example along with the appropriate comment describing the message name. This exercise can be very useful in order to learn bitcoin and it is recommended that the experiments be carried out on the bitcoin testnet, where various messages and transactions can be sent over the network and then be analyzed by Wire shark.

There are 27 types of protocol messages in total, but they're likely to increase over time as the protocol grows. The most commonly used protocol messages and their explanation are listed as follows:

- **Version**: This is the first message that a node sends out to the network, advertising its version and block count. The remote node then replies with the same information and the connection is then established.

Time	192.168.0.13 → 136.243.139.96	Comment
97.734135000	(57868) version → (18333)	Bitcoin: version
98.025045000	(57868) verack → (18333)	Bitcoin: verack
98.025177000	(57868) getaddr, pin... → (18333)	Bitcoin: getaddr, ping, addr
98.025468000	(57868) getheaders, → (18333)	Bitcoin: getheaders, [unknown command], [unknown command], [unknown command], headers
98.160419000	(57868) [TCP Retran... → (18333)	Bitcoin: [TCP Retransmission] , getheaders, [unknown command], [unknown command], [unknown command]
98.598399000	(57868) getdata → (18333)	Bitcoin: getdata
144.343544000	(57868) inv → (18333)	Bitcoin: inv
176.152240000	(57868) getdata → (18333)	Bitcoin: getdata
179.493755000	(57868) getdata → (18333)	Bitcoin: getdata
218.101646000	(57868) ping → (18333)	Bitcoin: ping
218.192004000	(57868) [unknown co... → (18333)	Bitcoin: [unknown command]
218.444431000	(57868) [TCP Retran... → (18333)	Bitcoin: [TCP Retransmission] , [unknown command]
336.234936000	(57868) getdata → (18333)	Bitcoin: getdata
337.843423000	(57868) [unknown co... → (18333)	Bitcoin: [unknown command]
338.143885000	(57868) ping → (18333)	Bitcoin: ping
448.764093000	(57868) getdata → (18333)	Bitcoin: getdata
457.894823000	(57868) [unknown co... → (18333)	Bitcoin: [unknown command]
458.195265000	(57868) ping → (18333)	Bitcoin: ping
578.011774000	(57868) [unknown co... → (18333)	Bitcoin: [unknown command]
578.212044000	(57868) ping → (18333)	Bitcoin: ping
585.587671000	(57868) inv → (18333)	Bitcoin: inv
647.169633000	(57868) inv → (18333)	Bitcoin: inv
671.962545000	(57868) getdata → (18333)	Bitcoin: getdata
698.037067000	(57868) [unknown co... → (18333)	Bitcoin: [unknown command]
698.237350000	(57868) ping → (18333)	Bitcoin: ping
701.563581000	(57868) inv → (18333)	Bitcoin: inv
701.986269000	(57868) inv → (18333)	Bitcoin: inv
705.022173000	(57868) inv → (18333)	Bitcoin: inv
812.115878000	(57868) inv → (18333)	Bitcoin: inv
818.198570000	(57868) [unknown co... → (18333)	Bitcoin: [unknown command]
818.298733000	(57868) ping → (18333)	Bitcoin: ping

- **Verack :** This is the response of the version message accepting the connection request.
- **Inv:** This is used by nodes to advertise their knowledge of blocks and transactions.
- **Getdata :** This is a response to inv, requesting a single block or transaction identified by its hash.
- **Getblocks:** This returns an *inv* packet containing the list of all blocks starting after the last known hash or 500 blocks.
- **Getheaders :** This is used to request block headers in a specified range.
- **Tx :** This is used to send a transaction as a response to the getdata protocol message.
- **Block:** This sends a block in response to the *getdata* protocol message.
- **Headers:** This packet returns up to 2,000 block headers as a reply to the getheaders request.
- **Getaddr:** This is sent as a request to get information about known peers.
- **Addr:** This provides information about nodes on the network. It contains the number of addresses and address list in the form of IP address and port.

- **Full client and SPV client**: Full clients are thick clients or full nodes that download the entire blockchain; this is the most secure method of validating the blockchain as a client. Bitcoin network nodes can operate in two fundamental modes: full client or lightweight SPV client. SPV clients are used to verify payments without requiring the download of a full blockchain. SPV nodes only keep a copy of block headers of the current valid longest blockchain. Verification is performed by looking at the merkle branch that links the transactions to the original block the transaction was accepted in. This is not very practical and requires a more practical approach, which was implemented with BIP37, where bloom filters were used to filter out relevant transactions only.
- **Bloom filters**: Bloom filter is basically a data structure (a bit vector with indexes) that is used to test the membership of an element in a probabilistic manner. It basically provides probabilistic lookup with false positives but no false negatives. Elements are added to the bloom filter after hashing them several times and then set the corresponding bits in the bit vector to 1 via the corresponding index. In order to check the presence of the element in the bloom filter, the same hash functions are applied and compared with the bits in the bit vector to see whether the same bits are set to 1. Not every hash function (such as SHA1) is suitable for bloom filters as they need to be fast, independent, and uniformly distributed. The most commonly used hash functions for bloom filters are fnv, mumur, and Jenkins.

  These filters are mainly used by simple payment verification SPV clients to request transactions and the merkle blocks they are interested in. A merkle block is a lightweight version of the block, which includes a block header, some hashes, a list of 1-bit flags, and a transaction count. This information can then be used to build a merkle tree. This is achieved by creating a filter that matches only those transaction and blocks that have been requested by the SPV client. Once version messages have been exchanged and connection has been established between peers, the nodes can set filters according to their requirements. These probabilistic filters offer a varying degree of privacy or precision depending upon how accurately or loosely they have been set. A strict bloom filter will only filter transactions that have been requested by the node but at the expense of the possibility of revealing the user addresses to adversaries who can correlate transactions with their IP addresses, thus compromising privacy. On the other hand, a loosely set filter can result in retrieving more unrelated transactions but will offer more privacy. Also, for SPV clients, bloom filters allow them to use low bandwidth as opposed to downloading all transactions for verification.

- **BIP 37** proposed the bitcoin implementation of bloom filters and introduced three new messages to the Bitcoin protocol.

- **Filterload:** This is used to set the bloom filter on the connection.
- **Filteradd:** This adds a new data element to the current filter.
- **FilterClear**: This deletes the currently loaded filter.

More details can be found in the BIP37 specification.

Wallets

The wallet software is used to store private or public keys and bitcoin address. It performs various functions, such as receiving and sending bitcoins. Nowadays, software usually offers both functionalities: bitcoin client and wallet. On the disk, the bitcoin core client wallets are stored as the Berkeley DB file:

:~/.bitcoin$ file wallet.dat

wallet.dat: Berkeley DB (Btree, version 9, native byte-order)

Private keys can be generated in different ways and are used by different types of wallets. Wallets do not store any coins, and there is no concept of wallets storing balance or coins for a user. In fact, in the bitcoin network, *coins* do not exist; instead, only transaction information is stored on the blockchain (more precisely, UTXO, unspent outputs), which are then used to calculate the amount of bitcoins.

Wallet types

In bitcoin, there are different types of wallets that can be used to store private keys. As a software program, they also provide some functions to the users to manage and carryout transactions on the bitcoin network.

Non-deterministic wallets

These wallets contain randomly generated private keys and are also called *Just a Bunch of Key wallets*. The bitcoin core client generates some keys when first started and generates keys as and when required. Managing a large number of keys is very difficult and an error-prone process can lead to theft and loss of coins. Moreover, there is a need to create regular backups of the keys and protect them appropriately in order to prevent theft or loss.

Deterministic wallets

In this type of wallet, keys are derived out of a seed value via hash functions. This seed number is generated randomly and is commonly represented by

human-readable *mnemonic code* words.
Mnemonic code words are defined in BIP39. This phrase can be used to recover all keys and makes private key management comparatively easier.

Hierarchical deterministic wallets

Defined in BIP32 and BIP44, HD wallets store keys in a tree structure derived from a seed. The seed generates the parent key (master key), which is used to generate child keys and, subsequently, grandchild keys. Key generation in HD wallets does not generate keys directly; instead, it produces some information (private key generation information) that can be used to generate a sequence of private keys. The complete hierarchy of private keys in an HD wallet is easily recoverable if the master private key is known. It is because of this property that HD wallets are very easy to maintain and are highly portable.

Brain wallets

The master private key can also be derived from the hash of passwords that are memorized. The key idea is that this passphrase is used to derive the private key and if used in HD wallets, this can result in a full HD wallet that is derived from a single memorized password. This is known as brain wallet. This method is prone to password guessing and brute force attacks but techniques such as *key stretching* can be used to slow down the progress made by the attacker.

Paper wallets

As the name implies, this is a paper-based wallet with the required key material printed on it. It requires physical security to be stored. Paper wallets can be generated online from various service providers

Hardware wallets

Another method is to use a tamper-resistant device to store keys. This tamper-resistant device can be custom-built or with the advent of NFC-enabled phones, this can also be a **secure element** (**SE**) in NFC phones. Trezor and Ledger wallets (various types) are the most commonly used bitcoin hardware wallets.

Trezor Wallet

Online wallets

Online wallets, as the name implies, are stored entirely online and are provided as a service usually via cloud. They provide a web interface to the users to manage their wallets and perform various functions such as making and receiving payments. They are easy to use but imply that the user trust the online wallet service provider.

Mobile wallets

Mobile wallets, as the name suggests, are installed on mobile devices. They can provide various methods to make payments, most notably the ability to use smart phone cameras to scan QR codes quickly and make payments. Mobile wallets are available for the Android platform and iOS, for example, breadwallet, copay, and Jaxx.

Jaxx Mobile wallet

Bitcoin payments

Bitcoins can be accepted as payments using various techniques. Bitcoin is not recognized as a legal currency in many jurisdictions, but it is increasingly being accepted as a payment method by many online merchants and e-commerce websites. There are a numbers of ways in which buyers can pay the business that accepts bitcoins. For example, in an online shop, bitcoin merchant solutions can be used, whereas in traditional physical shops, point of sale terminals and other specialized hardware can be used. Customers can simply scan the QR barcode with the seller's payment URI in it and pay using their mobile devices. Bitcoin URIs allow users to make payments by simply clicking on links or scanning QR codes. **URI** (**Uniform Resource Idenfier**) is basically a string that represents the transaction information. It is defined in BIP21. The QR code can be displayed near the point of the sale terminal. Nearly all bitcoin wallets support this feature.

Business can use the following screenshot to advertise that they can accept bitcoins as payment.

bitcoin accepted here logo

Various payment solutions, such as xbtterminal and 34 bytes bitcoin POS terminal are available commercially.

34 bytes POS solution.

Bitcoin payment processor, offered by many online service providers, allows integration with e- commerce websites. A simple Internet search can reveal many options.

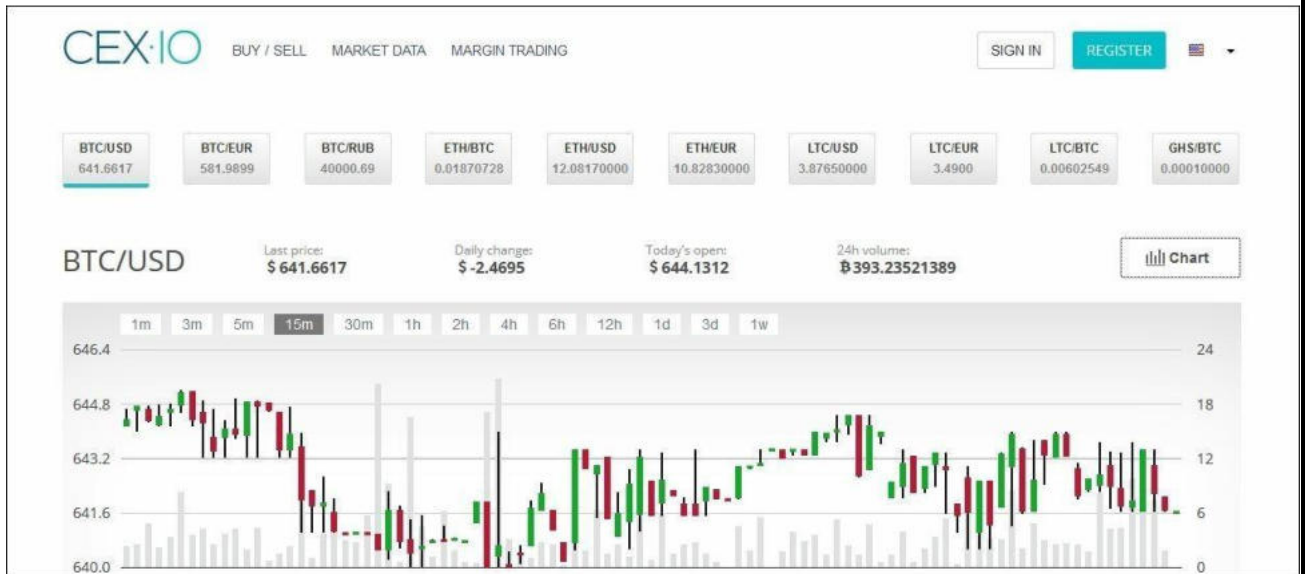
Various BIPs have been proposed and finalized in order to introduce and standardize bitcoin payments. Most notably, BIP 70 (secure payment protocol) describes the protocol for secure communication between a merchant and customers. This protocol uses X.509 certificates for authentication and runs over HTTP and HTTPS. There are three messages in this protocol: PaymentRequest, Payment, and PaymentACK. The key features of this proposal is defence against man-in-the-middle attacks and secure proof of payment. Man in-the-middle attacks can result in a scenario where the attacker is sitting between the merchant and the buyer and it would seem to the buyer that they are talking to the merchant, but in fact, the *man in the middle* is interacting with the buyer instead of the merchant. This can result in manipulation of the merchant's bitcoin address to defraud the buyer.

Several others BIPs, such as BIP71 and BIP72, have also been proposed to standardize payment message encapsulation and URI scheme to support BIP70.

Bitcoin lightning network, a solution for scalable off-chain instant payments, was introduced in early 2016, which allows off-blockchain payments. The network makes use of payments channels that run off the blockchain. This allows greater speed and scalability of bitcoin.

Bitcoin investment and buying and selling bitcoins

There are many online exchanges where users can buy and sell bitcoins. This is a big business on the Internet now and it offers bitcoin trading, CFDs, spread betting, margin trading, and various other choices. Traders can buy bitcoins or trade by opening long or short positions to make profit when bitcoin's price goes up or down. Several other features, such as exchanging bitcoins for other virtual currencies, are also possible, and many online bitcoin exchanges provide this function. Advanced market data, trading strategies, charts, and relevant data to support traders is also available. An example is shown from **CEX.IO** here. Other exchanges offer similar types of services.



Example of bitcoin exchange cex.io

The following screenshot shows the order book at the exchange where all buy and sell orders are listed:

Sell Orders — Total BTC available: 656.41831367

Price per BTC	BTC Amount	Total: (USD)
642.4085	Ƀ0.20450000	$ 131.38
642.4915	Ƀ0.20910000	$ 134.35
643.4470	Ƀ0.05000000	$ 32.18
643.4900	Ƀ0.11944972	$ 76.87
643.5000	Ƀ1.85748652	$ 1195.30
643.6500	Ƀ3.00000000	$ 1930.95
643.6999	Ƀ0.13844181	$ 89.12
643.7000	Ƀ45.80000000	$ 29481.46
643.7487	Ƀ1.22995538	$ 791.79

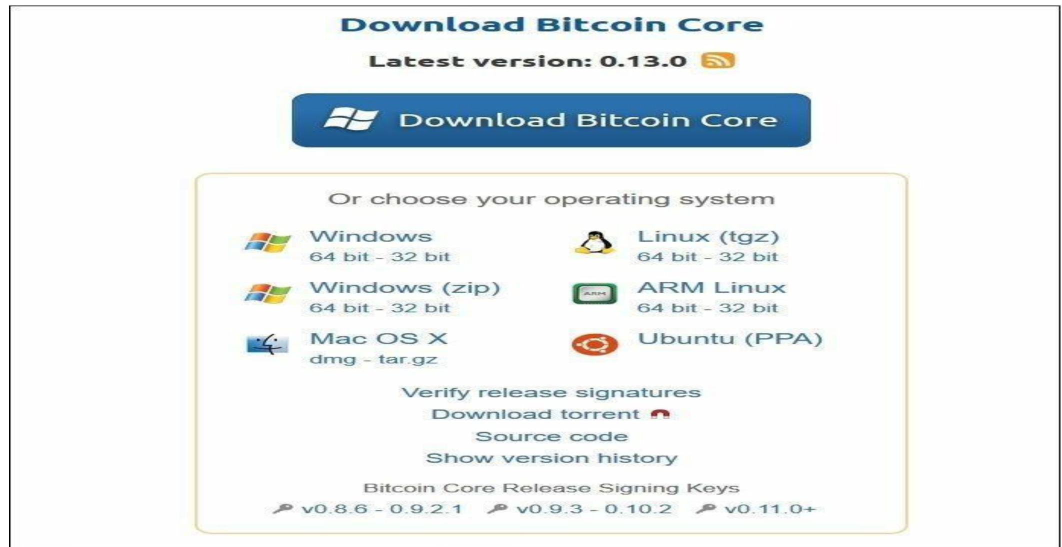
Buy Orders — Total USD available: 380739.41

Price per BTC	BTC Amount	Total: (USD)
641.6210	Ƀ0.01390000	$ 8.92
641.6201	Ƀ0.23162780	$ 148.62
641.6200	Ƀ0.12050000	$ 77.32
641.6117	Ƀ1.83477084	$ 1177.22
641.5584	Ƀ0.30000000	$ 192.47
641.5217	Ƀ0.18180000	$ 116.63
641.0217	Ƀ0.10000000	$ 64.11
640.5300	Ƀ0.67323160	$ 431.23
640.5000	Ƀ0.40815400	$ 261.43

Example of bitcoin order book at exchange cex.io

Bitcoin installation

The bitcoin core client can be installed from https://bitcoin.org/en/download. This is available for different architectures and platforms ranging from x86 windows to ARM Linux, as shown in the following image:



Setting up a bitcoin node

sample run of the bitcoin core installation on Ubuntu is shown here; for other platforms, you can get details from www.bitcoin.org.

```
drequinox@drequinox-OP7010: ~
drequinox@drequinox-OP7010:~$ sudo apt-add-repository ppa:bitcoin/bitcoin
[sudo] password for drequinox:
 Stable Channel of bitcoin-qt and bitcoind for Ubuntu, and their dependencies
 More info: https://launchpad.net/~bitcoin/+archive/ubuntu/bitcoin
Press [ENTER] to continue or ctrl-c to cancel adding it

gpg: keyring `/tmp/tmpzsl4ltrx/secring.gpg' created
gpg: keyring `/tmp/tmpzsl4ltrx/pubring.gpg' created
gpg: requesting key 8842CE5E from hkp server keyserver.ubuntu.com
gpg: /tmp/tmpzsl4ltrx/trustdb.gpg: trustdb created
gpg: key 8842CE5E: public key "Launchpad PPA for Bitcoin" imported
gpg: no ultimately trusted keys found
gpg: Total number processed: 1
gpg:               imported: 1  (RSA: 1)
OK
drequinox@drequinox-OP7010:~$
```

drequinox@drequinox-OP7010:~$ sudo apt-get update

Depending on the client required, users can use either of the following

commands, or they can issue both commands at once:

sudo apt-get install bitcoind sudo apt-get install bitcoin-qt drequinox@drequinox-OP7010:~$ sudo apt-get install bitcoin-qt bitcoind Reading package lists... Done
Building dependency tree
Reading state information... Done
....

Setting up the source code

The bitcoin source code can be downloaded and compiled if users wish to participate in the bitcoin code or for learning purpose. Git can be used to download the bitcoin source code:

$ sudo apt-get install git

$ mkdir bcsource

$ cd bcsource

drequinox@drequinox-OP7010:~/bcsource $ git clone https://github.com/bitcoin/bitcoin.git

Cloning into 'bitcoin'...

remote: Counting objects: 78960, done.

remote: Compressing objects: 100% (3/3), done.

remote: Total 78960 (delta 0), reused 0 (delta 0), pack-reused 78957

Receiving objects: 100% (78960/78960), 72.53 MiB | 1.85 MiB/s, done.

Resolving deltas: 100% (57908/57908), done.

Checking connectivity... done. drequinox@drequinox-

OP7010:~/bcsource$

Change the directory to bitcoin:

drequinox@drequinox-OP7010:~/bcsource$ cd bitcoin

After the preceding steps are completed, the code can be compiled:

drequinox@drequinox-OP7010:~/bcsource/bitcoin$ ./autogen.sh

drequinox@drequinox-OP7010:~/bcsource/bitcoin$ ./configure.sh

drequinox@drequinox-OP7010:~/bcsource/bitcoin$ make

drequinox@drequinox-OP7010:~/bcsource/bitcoin$ sudo make install

Setting up bitcoin.conf

bitcoin.conffile is a configuration file that is used by the bitcoin core client to save configuration settings. All command line options for the bitcoind client with the exception of -confswitch can be set up in the configuration file, and when bitcoin-qt or bitcoind will start up, it will take the configuration information from that file.

In Linux systems, this is usually found in $HOME/.bitcoin/, or it can also specified in the command line using the -conf=<file>switch to bitcoind core client software.

Starting up a node in testnet

The bitcoin node can be started in the testnet mode if you want to test the bitcoin network and run an experiment. This is a faster network as compared to the live network and has relaxed rules for mining and transactions.

Various faucet services are available for the bitcoin test network. One example is Bitcoin TestNet sandbox, where users can request bitcoins to be paid to their testnet bitcoin address. This can be accessed via https://testnet.manu.backend.hamburg/. This is very useful for experimentation with transactions on test net.

The command line to start up test net is as follows:

bitcoind --testnet -daemon bitcoin-cli --testnet <command>

bitcoin-qt --testnet

Starting up a node in regtest

The regtest mode (regression testing mode) can be used to create a local blockchain for testing purposes.

The following commands can be used to start up a node in the reg test mode:

bitcoind -regtest -daemon Bitcoin server starting

Blocks can be generated using the following command:

bitcoin-cli -regtest generate 200

Relevant log messages can be viewed in the .bitcoin/regtest directory on a Linux system under

```
drequinox@drequinox-OP7010:~/.bitcoin/regtest$ tail -f debug.log
2016-10-16 15:43:55 AddToWallet d461e1fb162dd6958139a2ab5e4f9993ffbd51b1a4e3a80e5b77e472cd90dd6a  new
2016-10-16 15:43:55 CreateNewBlock(): total size 1000 txs: 0 fees: 0 sigops 400
2016-10-16 15:43:55 UpdateTip: new best=37c1f40299a3724dd2edf63d26925cb580b8c5f27405289ef9204e53fe4e1b87 height=299 version=0x30000003 log2_work=9.22881
87 tx=300 date='2016-10-16 15:44:27' progress=1.000000 cache=0.1MiB(299tx)
2016-10-16 15:43:55 AddToWallet b88883e122c4f3ae66b53e4026d8fa6c916c570df2b154febf51c676235d70bf  new
2016-10-16 15:43:55 CreateNewBlock(): total size 1000 txs: 0 fees: 0 sigops 400
2016-10-16 15:43:55 UpdateTip: new best=5c22d0b090b6f3fd978fbbb14803d1d34ecccfb697a199d502beb1d88da43ad2 height=300 version=0x30000003 log2_work=9.23361
97 tx=301 date='2016-10-16 15:44:28' progress=1.000000 cache=0.1MiB(300tx)
2016-10-16 15:43:55 AddToWallet e315c5b6863aed2d4477f6e6e5cdb7ace273f40549d249b90b8793de0de0b8e1  new
2016-10-16 15:43:55 CreateNewBlock(): total size 1000 txs: 0 fees: 0 sigops 400
2016-10-16 15:43:55 UpdateTip: new best=7f9eeb78cdb34f374d426c95aab82c85810715574c0a87ec93218ab77ae9f5ae height=301 version=0x30000003 log2_work=9.23840
47 tx=302 date='2016-10-16 15:44:28' progress=1.000000 cache=0.1MiB(301tx)
2016-10-16 15:43:55 AddToWallet 428058e9e73f6862f8e126999efa4062dad2e63b253630d2e2ec086e7f5ac029  new
```

debug.log.

After block generation, the balance can be viewed as follows:

drequinox@drequinox-OP7010:~/.bitcoin/regtest$ bitcoin-cli -regtest

getbalance 8750.00000000

The node can be stopped using this:

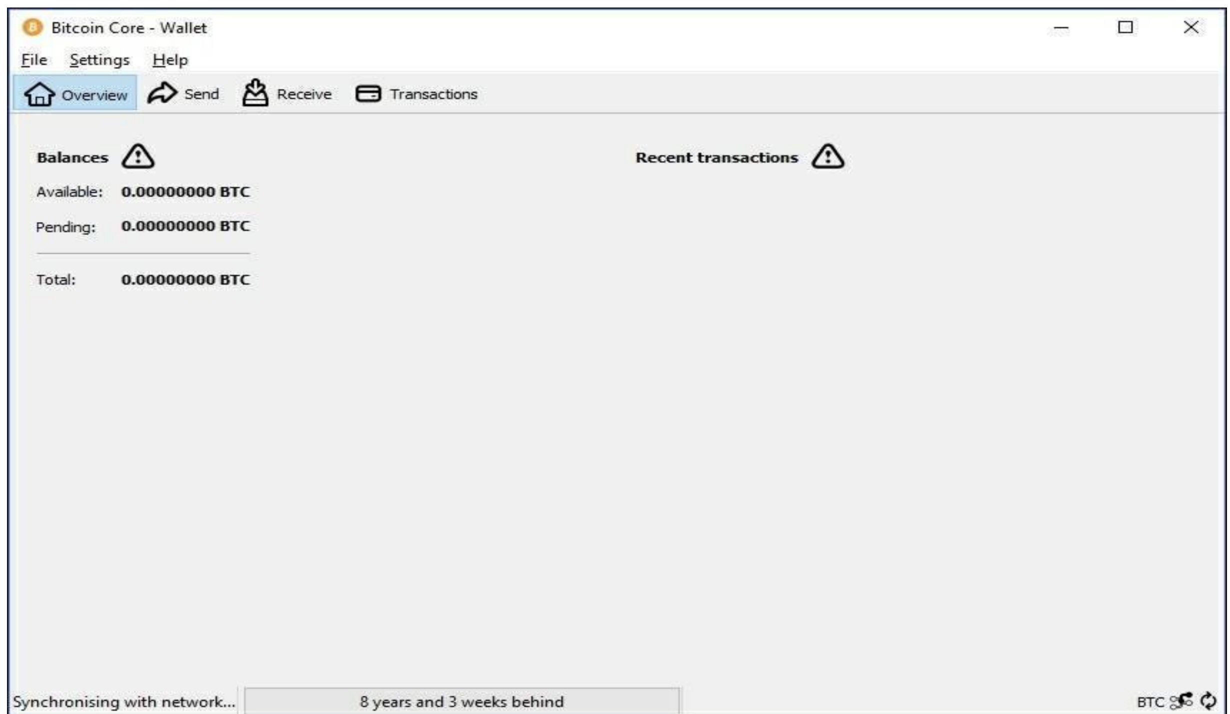
drequinox@drequinox-OP7010:~/.bitcoin$ bitcoin-cli -regtest stop

Bitcoin server stopping

Starting up a node in live mainnet

Bitcoind is the core client software that can be run as a daemon, and it provides the JSON RPC interface. *Bitcoin-cli* is the command line feature-rich tool to interact with the daemon; the daemon then interacts with the blockchain and performs various functions. Bitcoin-cli calls only JSON-RPC functions and does not perform any actions on its own on the blockchain.

Bitcoin-qt is the bitcoin core client GUI. When the wallet software starts up first, it verifies the blocks on the disk and then starts up and shows the following GUI:



Bitcoin Core QT client, just after installation, showing that blockchain is not in sync

The verification process is not specific to the Bitcoin-qt client; it is performed by the *bitcoind* client as well.

Experimenting with bitcoin-cli

Bitcoin-cli is the command-line interface available with the bitcoin core client and can be used to perform various functions using the RPC interface provided by the bitcoin core client.

```
drequinox@drequinox-OP7010:~$ bitcoin-cli getinfo
{
  "version": 130000,
  "protocolversion": 70014,
  "walletversion": 130000,
  "balance": 0.00000000,
  "blocks": 433948,
  "timeoffset": 0,
  "connections": 8,
  "proxy": "",
  "difficulty": 258522748404.5154,
  "testnet": false,
  "keypoololdest": 1475534258,
  "keypoolsize": 100,
  "paytxfee": 0.00000000,
  "relayfee": 0.00001000,
  "errors": ""
}
drequinox@drequinox-OP7010:~$
```

A sample run of bitcoin-cli getinfo; the same format can be used to invoke other commands A list of all commands can be shown via the following command:

```
drequinox@drequinox-OP7010:~$ bitcoin-cli -testnet help | more
== Blockchain ==
getbestblockhash
getblock "hash" ( verbose )
getblockchaininfo
getblockcount
getblockhash index
getblockheader "hash" ( verbose )
getchaintips
getdifficulty
getmempoolancestors txid (verbose)
getmempooldescendants txid (verbose)
getmempoolentry txid
getmempoolinfo
getrawmempool ( verbose )
gettxout "txid" n ( includemempool )
gettxoutproof ["txid",...] ( blockhash )
gettxoutsetinfo
verifychain ( checklevel numblocks )
verifytxoutproof "proof"

== Control ==
getinfo
help ( "command" )
stop
```

Testnet bitcoin-cli, this is just the first few lines of the output, actual output has many commands

- **HTTP REST**: Starting from bitcoin core client 0.10.0, the HTTP REST interface is also available. By default, this runs on the same TCP port 8332 as JSON-RPC.

Bitcoin programming and the command-line interface

Bitcoin programming is a very rich field now. The bitcoin core client exposes various JSON RPC commands that can be used to construct raw transactions and perform other functions via custom scripts or programs. Also, the command line tool Bitcoin-cli is available, which makes use of the JSON-RPC interface and provides a rich toolset to work with Bitcoin.

These APIs are also available via many online service provider in the form of bitcoin APIs, and they provide a simple HTTP REST interface. Bitcoin APIs, such as blockchain.infoand bitpay, block.io, and many others, offer a myriad of options to develop bitcoin-based solutions.

Bitcoin improvement proposals (BIPs)

These documents are used to propose or inform the bitcoin community about the improvements suggested, the design issues, or information about some aspects of the bitcoin ecosystem. There are three types of bitcoin improvement proposals, abbreviated as BIPs:

- **Standard BIP**: Used to describe the major changes that have a major impact on the bitcoin system, for example, block size changes, network protocol changes, or transaction verification changes.
- **Process BIP**: A major difference between standard and process BIPs is that standard BIPs cover protocol changes, whereas process BIPs usually deal with proposing a change in a process that is outside the core Bitcoin protocol. These are implemented only after a consensus among bitcoin users.
- **Informational BIP**: These are usually used to just advise or record some information about the bitcoin ecosystem, such as design issues.

Alternative Coins

Since the initial success of bitcoin, many alternative currency projects have been launched. Bitcoin was released in 2009 and the first alternative coin project (named Namecoin) was introduced in 2011. In 2013 and 2014, the altcoin market grew exponentially and many different types of alternative coin project were started. A few of those became a success, whereas many were unpopular and did not succeed. A few were *pump and dump* scams that surfaced for some time but soon disappeared.
Alternative approaches to bitcoin can be divided broadly into two categories, based on the primary purpose of their development. If the primary purpose is to build a decentralized blockchain platform, they are called alternative chains; if the sole purpose of the alternative project is to introduce a new virtual currency, it's called an altcoin.

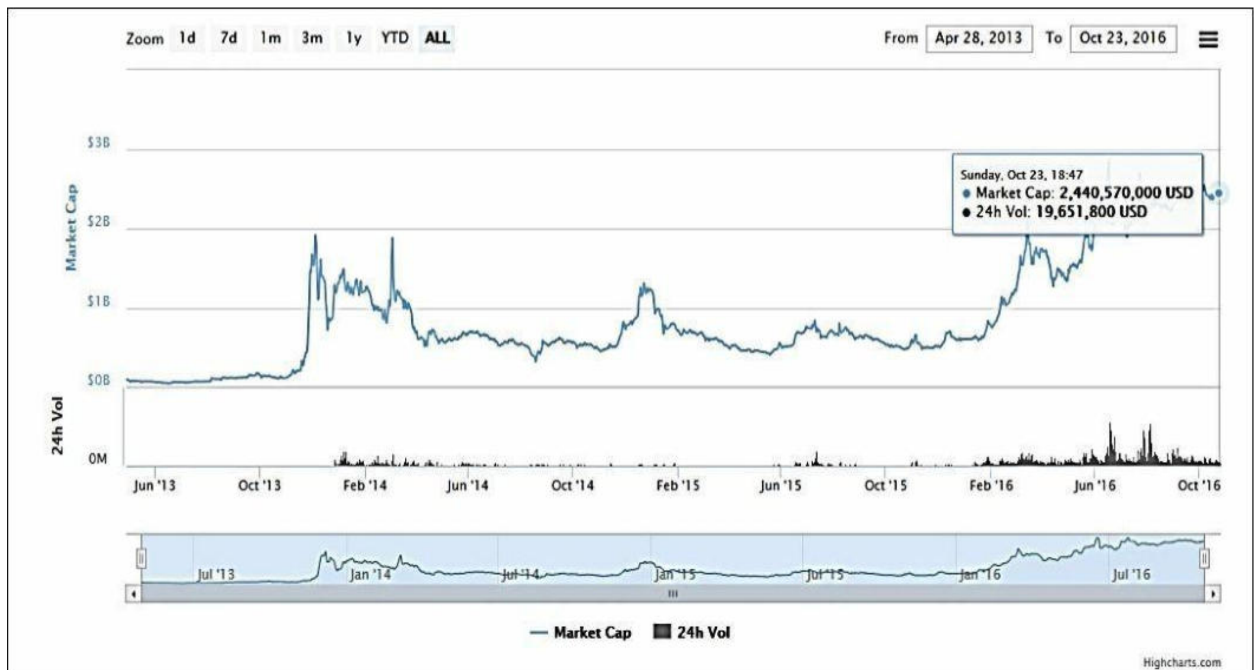
Currently, as of late 2016, there are hundreds of altcoins on the market and they hold a certain monetary value. Many of these alternative projects are direct forks of bitcoin source code although some of those have been written from scratch. Some altcoins set out to address bitcoin limitations such as privacy. Some others offer different types of mining, changes in block times, and distribution schemes.

By definition an altcoin is generated in the case of a hard fork. If bitcoin has a hard fork then the other, older chain is effectively considered another coin. However there is no established rule as to which chain becomes the altcoin. This has happened recently with Ethereum, where a hard fork caused a new currency **ETC** (**Ethereum classic**) to come into existence in addition to the **Ethereum** (**ETH**) currency. Ethereum classic is the old chain and Ether is the new chain after the fork. Such a contentious hard fork is not desirable for a number of reasons. First it is against the true spirit of decentralization as the Ethereum foundation, a central entity, decided to go ahead with the hardfork even though not everyone agreed to the proposition; second it also splits the user community due to disagreement over the hard fork. Although a hard fork in theory generates an altcoin, it is limited in what it can offer due to the fact that, even if the change results in a hard fork, usually there are no drastic changes around the fundamental parameters of the coin. They usually remain the same. For this reason, it is desirable to either write a new coin from scratch or fork the bitcoin (or another coin's source code) to create a new currency with the desired parameters and features.

Altcoins must be able to attract new users, trades, and miners otherwise the currency will have no value. Currency gains its value, especially in the virtual

currency space, due to the network effect and its acceptability by the community. If a coin fails to attract enough users then soon it will be forgotten. Users can be attracted by providing an initial amount of coins and can be achieved by using various methods. Methods of providing an initial number of altcoins are discussed as follows:

- **Create a new blockchain**: Altcoins can create a new blockchain and allocate coins to initial miners but this approach is now unpopular due to many scam schemes or *pump and dump* schemes where initial miners made a profit with the launch of a new currency and then disappeared.
- **Proof of burn**: Another approach to allocating initial funds to a new altcoin is *Proof of burn*, also called a one-way peg or price ceiling. In this method users permanently destroy a
certain quantity of bitcoins in proportion to the quantity of altcoins to be claimed. For example if 10 bitcoins were destroyed then altcoins can have a value no greater than the amount of bitcoins destroyed. This basically means that bitcoins are being *converted* into altcoins by burning them.
- **Proof of Ownership**: Instead of permanently destroying bitcoins, an alternative method is to prove that users own a certain number of bitcoins. This proof of ownership can be used to claim altcoins by tethering altcoin blocks to bitcoin blocks. For example, this can be achieved by merged mining in which effectively bitcoin miners can mine altcoin blocks while mining for bitcoin without any extra work.
- **Pegged sidechains**: Sidechains, as the name suggests, are blockchains separate from the bitcoin network but bitcoins can be transferred to them. Altcoins can also be transferred back to the bitcoin network. This concept is called a two-way peg.



This shows that at the time of writing the Combined Altcoin Market Capitalization is more than 2 billion US Dollars

Current market cap (as of Oct 2016) of the top 10 coins is shown as follows:

Name	Market Cap	Price
Bitcoin	£8,983,068,268	£563.40
Ethereum	£806,208,216	£9.44
Ripple	£248,473,625	£0.01
Litecoin	£159,226,903	£3.31
Ethereum Classic	£69,366,041	£0.81
Monero	£67,664,027	£5.12
Dash	£54,675,636	£8.00
Augur	£45,221,626	£4.11
MaidSafeCoin	£30,929,833	£0.07
NEM	£28,761,896	£0.00

There are various factors and new concepts introduced with alternative coins. Many concepts were invented even before bitcoin but with bitcoin not only were new concepts, such as a solution to the Byzantine Generals' problem, introduced for the first time but also previous concepts such as hashcash and Proof of Work were used in an ingenious way and came into the limelight. Since then, with the introduction of alternative coin projects, various new techniques and concepts have been developed and introduced. In order to appreciate the current landscape of alternative cryptocurrencies, it is essential to understand some theoretical concepts first. In the following section, some new concepts that have been introduced with altcoin projects are introduced to the reader.

Theoretical foundations

In this section, various theoretical concepts are introduced to the reader that have been developed with the introduction of different altcoins in the past few years.

Alternatives to Proof of Work

The **Proof of Work** (**PoW**) scheme in the context of cryptocurrency was first used in bitcoin and served as a mechanism to provide assurance that a miner had completed the required number of work in order to find a block. This in return provided decentralization, security, and stability for the blockchain. Also, this is the main vehicle in bitcoin for providing decentralized distributed consensus. PoW schemes are required to have a much desired property called *progress freeness*, which basically means that the reward for consuming computational resources should be random and proportional to the contribution made by the miners.

Proof of Storage

Also known as proof of irretrievability, this is another type of proof of useful work that requires storage of large number of data. Introduced by Microsoft Research, this scheme provides a useful benefit of distributed storage of archival data. Miners are required to store a pseudo, randomly- selected subset of large data in order to perform mining.

Proof of Stake

This proof is also called virtual mining. This is another type of mining puzzle that has been proposed as an alternative to traditional PoW schemes. It was

first proposed in PeerCoin in August, 2012. In this scheme, the idea is that users are required to demonstrate possession of a certain amount of currency (coins) thus proving that they have a stake in the coin. The simplest form of stake is where mining is made comparatively easier for those users who demonstrably own larger amounts of digital currency. The benefits of this scheme are twofold; first acquiring large amounts of digital currency is relatively difficult as compared to buying high-end ASIC devices and second it results in saving computational resources. Various forms of stake have been proposed and are briefly discussed below.

Proof of coinage

The age of a coin is the time since the coins were last used or held. This is a different approach from the usual form of Proof of Stake where mining is made easier for users who have the highest stake in the altcoin. In the coin-age-based approach the age of the coin (coinage) is reset every time a block is mined. The miner is rewarded for holding and not spending coins for a time period. This mechanism has been implemented in Peercoin combined with PoW in a creative way. The difficulty of mining puzzles (PoW) is inversely proportional to the coin-age, meaning that if miners consume some coin- age using *coin-stake* transactions then the PoW requirements are relieved.

Proof of deposit

The core idea behind this scheme is that newly minted blocks by miners are made un-spendable for a certain period of time. More precisely the coins get locked for a set number of blocks during the mining operation. The scheme works by allowing miners to perform mining at the cost of freezing a certain number of coins for some time. This is a type of Proof of Stake.

Proof of burn

As an alternate expenditure to computing power, proof of burn in fact destroys a certain amount of bitcoins in order to get equivalent altcoins. This is commonly used when starting up new coin projects as a means to provide a fair initial distribution. This can be considered an alternative mining scheme where the value of the new coins comes from the fact that previously a certain number of coins have been destroyed.

Proof of activity : This scheme is a hybrid of PoW and Proof of Stake. In blocks are initially produced by using PoW but then each block randomly assigns three stakeholders that are required to digitally sign it. The validity of subsequent blocks is dependent on the successful signing of previously randomly chosen blocks.

There is, however a possible issue known as the *nothing at stake* problem where it would be trivial to create a fork of the blockchain. This is possible because in PoW appropriate computational resources are required to mine whereas in Proof of Stake there is no such requirement; as a result, an attacker can try to mine on multiple chains using the same coin.

Non-outsourceable puzzles

The key motivation behind this puzzle is to develop resistance again the development of mining pools. Mining pools as previously discussed offer rewards to all participants in proportion to the computing power they consume. However, in this model the mining pool operator is a central authority to whom all the rewards go and who can enforce certain rules. Also, in this model all miners only trust each other because they are working towards a common goal together in the hope of the pool manager getting the reward. Non-outsourceable puzzles are a scheme that allows miners to claim rewards for themselves; consequently pool formation becomes unlikely due to inherent mistrust between anonymous miners.

Dark Gravity Wave

Dark Gravity Wave (**DGW**) is a new algorithm designed to address certain flaws such as the time warp attack in the KGW algorithm. This was first introduced in Dash, previously known as Darkcoin. It makes use of multiple exponential moving averages and simple move averages to achieve a smoother readjustment mechanism. The formula is shown as follows:

2222222/ (((Difficulty+2600)/9)^2)

This formula is implemented in Dash coin and various other altcoins as a mechanism to readjust difficulty.

DGW version 3.0 is the latest implementation of this algorithm and allows improved difficulty retargeting compared to KGW.

DigiShield

This is another difficulty retargeting algorithm that has recently been used in Zcash with slight variations and after adequate experimentation. This algorithm works by going through a fixed number of previous blocks to calculate the time they took to be generated and then readjusts the difficulty to the difficulty of the previous block by dividing the actual time span by averaging the target time. In this scheme, the retargeting is calculated much

much rapidly and also the recovery from a sudden increase or decrease in hashrate is quick. This algorithm protects against multipools, which can result in rapid hashrate increases. The network difficulty is readjusted every block or every minute depending on the implementation. The key innovation is faster readjust times as compared to KGW.

MIDAS

Multi Interval Difficulty Adjustment System (**MIDAS**) is an algorithm that is comparatively more complex than the algorithms discussed previously. This method responds much more rapidly to abrupt changes in hash rates. This algorithm also provides protection against time warp attacks.

Many currencies have emerged as an attempt to address various limitations in bitcoin. A brief discussion of bitcoin limitations is provided as follows.

Bitcoin limitations

Various limitations in bitcoin have also sparked some interest in altcoins, which were developed specifically to address limitations in bitcoin. The most prominent and widely discussed limitation is the lack of anonymity in bitcoin. This limitation is discussed in detail as follows.

Privacy and anonymity

As the blockchain is a public ledger of all transactions and is openly available it becomes trivial to analyse it. Combined with traffic analyses, transactions can be linked back to their source IP addresses, thus possibly revealing a transaction's originator. This is a big concern from a privacy point of view. Even though in bitcoin it is a recommended and common practice to generate a new address for every transaction, thus allowing some level of unlinkability, this is not enough and various techniques have been developed and successfully used to trace the flow of transactions throughout the network and link them back to their originator.

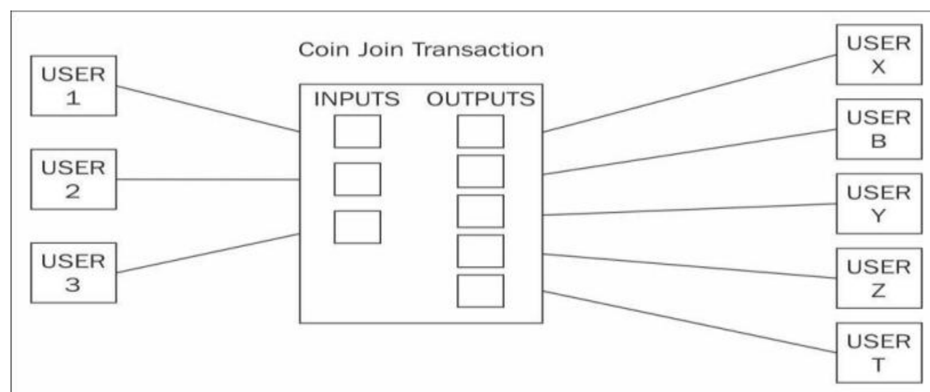
Various proposals have been made to address the privacy issue in bitcoin. These proposals fall into three categories: mixing protocols, third-party mixing networks, and inherent anonymity. A brief discussion of each category is presented as follows.

Mixing protocols

These schemes are used to provide anonymity to bitcoin transactions. In this model, a mixing service provider (an intermediary or a shared wallet) is used.

Users send coins to this shared wallet as a deposit and the shared wallet then can send some other coins (of the same value deposited by some other users) to the destination. Users can also receive coins that were sent by others via this intermediary. This way the link between outputs and inputs is no longer there and transaction graph analysis will not be able to reveal the true relationship between senders and receivers.

CoinJoin is one example of mixing protocols, where two transactions are joined together to form a single transaction while keeping the inputs and outputs unchanged. The core idea behind CoinJoin is to build a shared transaction that is signed by all participants. This technique improves privacy for all participants involved in the transactions:



CoinJoin transaction with three users joining their transaction into a single larger CoinJoin transaction

Third-party mixing protocols

Various third-party mixing services are available but if the service is centralized then it poses the threat of tracing the mapping between senders and receivers, because the mixing service knows about all inputs and outputs. In addition to this, fully centralized miners even pose the risk of the administrators of the service stealing the coins.

Various services, with varying degrees of complexity, such as CoinShuffle, Coinmux, and dark send in Dash (coin) are available that are based on the idea of CoinJoin (mixing) transactions. CoinShuffle is a decentralized alternative to traditional mixing services as it does not require a trusted third party.

CoinJoin-based schemes, however, have some weaknesses, most prominently the possibility of launching a Denial of Service attack by users who committed to signing the transactions initially but now are not providing their signature, thus delaying or stopping joint transaction a altogether.

Inherent anonymity

This category includes coins that support privacy inherently and is built into the design of the currency. The most popular is Zcash, which is discussed in detail later in the chapter. Other examples include Monero, which makes use of ring signatures to provide anonymous services.

Extended protocols on top of bitcoin

Several protocols have been proposed and implemented on top of bitcoin in order to enhance and extend the bitcoin protocol and use for various other purposes instead of just as a virtual currency.

Colored coins

Colored coins is a set of methods that have been developed to represent digital assets on the bitcoin blockchain. Coloring a bitcoin refers colloquially to updating it with some metadata representing a digital asset (smart property). The coin still works and operates as a bitcoin but additionally carries some metadata that represents some assets. This mechanism allows issuing and tracking specific bitcoins. Metadata can be recorded using the bitcoins OP_RETURN opcode or optionally in multi- signature addresses. This metadata can also be encrypted if required to address any privacy concerns. Colored coins can be used to represent a multitude of assets including but not limited to commodities, certificates, shares, bonds, and voting. It should also be noted that, in order to work with colored coins, a wallet that interprets colored coins is necessary and normal bitcoin wallets will not work.

The idea of colored coins is very appealing as it does not require any modification to the existing bitcoin protocol and can make use of the already existing secure bitcoin network. In addition to the traditional representation of digital assets, there is also the possibility of creating *smart assets* that behave according to the parameters and conditions defined for them. These parameters includes time validation, restrictions on transferability, and fees. This opens the possibility of creating smart contracts.

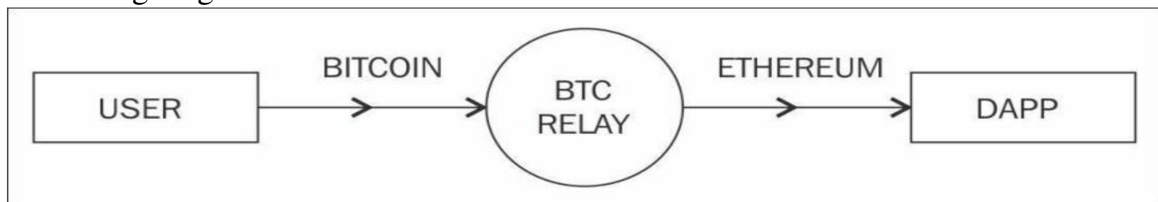
Counterparty

This is another service that can be used to create custom tokens that act as a crypto currency and can be used for various purposes such as issuing digital assets on top of bitcoin blockchain. This is quite a powerful platform and runs on bitcoin blockchains at their core but has developed its own client and other components to support issuing digital assets. The architecture consists of a counterparty server, counter block, counter wallet, and armory_utxsvr.

Counterparty works based on the same idea as coloured coins by embedding data into regular bitcoin transactions but provides a much richer library and set of powerful tools to support the handling of digital assets. This embedding is also called embedded consensus because the counterparty transactions are embedded within bitcoin transactions. The method of embedding the data is by using OP_RETURNopcode in bitcoin.

Counterparty allows the development of smart contracts on Ethereum using solidity language and allows interaction with bitcoin blockchain. In order to achieve this, BTC Relay is used as a means to provide interoperability between Ethereum and bitcoin. This is a clever concept where Ethereum contracts can talk to bitcoin blockchain and transactions through BTC Relay. The relayers (nodes that are running BTC Relay) fetch the bitcoin block headers and relay them to a smart contract on the Ethereum network that verifies the PoW.

Technically, this is basically an Ethereum contract that is capable of storing and verifying bitcoin block headers just like bitcoin simple payment verification lightweight clients do by using bloom filters. SPV clients were discussed in detail in the previous chapter. The idea can be visualized with the following diagram:



BTC relay concept

Development of altcoins

Altcoin projects can be started very easily from a coding point of view by simply forking the bitcoin or another coin's source code but this probably is not enough. When a new coin project is started, there are several things that need to be considered in order to ensure a successful launch and the coin's longevity. Usually, the code base is written in C++ as was the case with bitcoin but almost any language can be used to develop coin projects, for example Golang or Rust.

From a technical point of view, in the case of forking the code of another coin, for example bitcoin, there are various parameters that can be changed to effectively create a new coin. These parameters are required to be *tweaked* or introduced in order to create a new coin.

Consensus algorithms

There is a choice of consensus algorithm: **Proof of Work** (**PoW**) as used in bitcoin or **Proof of Stake** (**PoS**), as in Peercoin.

Hashing algorithms

This is either SHA256, Scrypt, X11, X13, X15, or any other hashing algorithm that is adequate for use as a consensus algorithm.

Difficulty adjustment algorithms

Various options are available in this category to provide difficulty retargeting mechanisms. The most prominent examples are KGW, DGW, Nite's Gravity Wave, and DigiShield. Also all these algorithms can be tweaked based on requirements to produce different results; therefore many variants are possible.

Inter-block time

This is the time elapsed between the generation of each block. For bitcoin the blocks are generated every 10 minutes, for litecoin it's 2.5 minutes. Any value can be used but an appropriate value is usually between a few minutes; if the generation time is too fast it might destabilize the blockchain, if it's too slow it may not attract many users.

Block rewards

A block reward is for the miner who solves the mining puzzle and is allowed to have a Coinbase transaction that contains the reward. This used to be 50 coins in bitcoin initially and now many altcoins set this parameter to a very high number; for example in Dogecoin it is 10,000, currently.

Reward halving rate

This is another important factor; in bitcoin it is halved every 4 years and now is set to 12.5 bitcoins. It's a variable number that can be set to any time period or none at all depending on the requirements.

Block size and transaction size

This is another important factor that determines how high or low the transaction rate can be on the network. Block sizes in bitcoin are limited to 1 MB but in altcoins it can vary depending on the requirements.

Interest rate

This property applies only to PoS systems where the owner of the coins can earn interest at a rate defined by the network in return for the amount of coins that are held on the network as a PoS to protect the network.

Coin age

This parameter defines how long the coin has to remain unspent in order for it to become eligible to be considered stakeworthy.

Total supply of coins

This number sets the total limit of the coins that can ever be generated. For example in bitcoin the limit is 21 million, whereas in Dogecoin it's unlimited. This limit is fixed by the block reward and halving schedule discussed above.

There are two options to create your own virtual currency: forking existing established crypto currency source code or writing a new one from scratch. The latter option is less popular but the first option is easier and has allowed the creation of many virtual currencies over the last few years. Fundamentally, the idea is that first a crypto currency source code is forked and then appropriate changes are made at different strategic locations in the source code to effectively create a new currency.

In the next section, readers are introduced to some altcoin projects. It is not possible to cover all alternative currencies in this chapter, but a few selected coins are discussed below. Selection is based on longevity, market cap, and innovation. Each coin is discussed from different angles such as theoretical foundations, trading, and mining.

Name Coin

Namecoin is the first fork of the bitcoin source code. The key idea behind Namecoin is not to produce an altcoin but instead to provide improved decentralization, censorship resistance, privacy, security, and faster decentralized naming. Decentralized naming services are intended to provide a response to inherent limitations such as slowness and centralized control in the traditional **Domain Name System** (**DNS**) protocols used on the Internet. Namecoin is also the first solution to Zooko's triangle, which was briefly discussed in previous chapters.

Namecoin is used to essentially provide a service to register a key/value pair. One major use case of Namecoin is that it can provide a decentralized

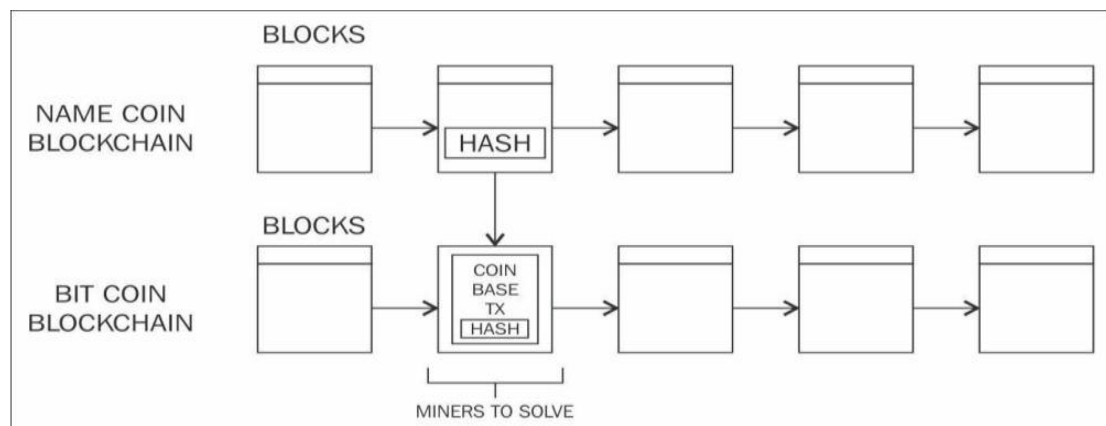
Transport Layer Security (**TLS**) certificate validation mechanism, driven by blockchain-based distributed and decentralized consensus.

In summary, Namecoin provides the following three services:

- Secure storage and transfer of names (keys)
- Attachment of some value to the names by attaching up to 520 bytes of data
- Production of a digital currency (Namecoin)

Namecoin also for the first time introduced merged mining, which allows a miner to mine on more than one chain simultaneously. The idea is simple but very effective: miners create a Namecoin block and produce a hash of that block. Then the hash is added to a bitcoin block and miners solve that block at equal to or greater than the Namecoin block difficulty in order to prove that enough work has been contributed towards solving the Namecoin block.

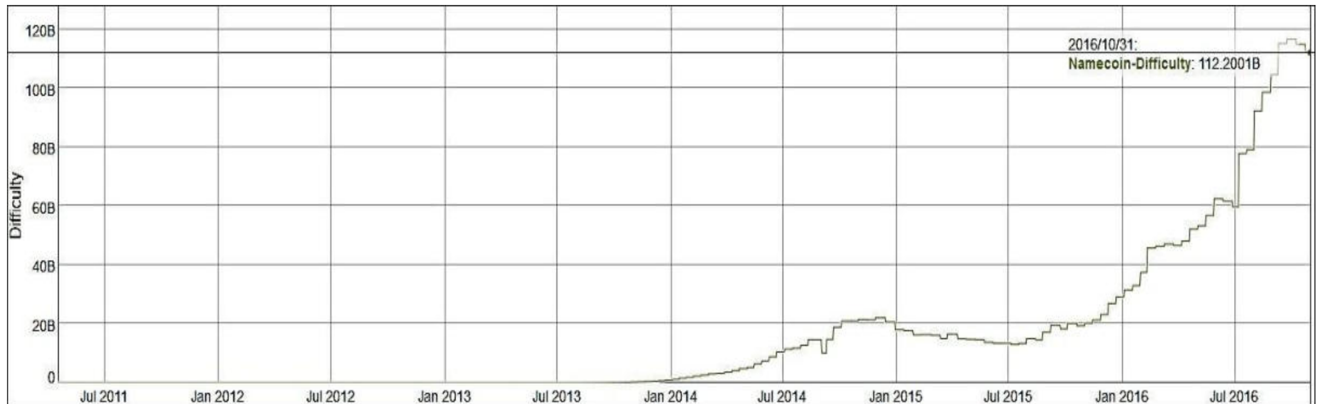
More precisely the Coinbase transaction is used to include the hash of the transactions from Namecoin (or any other altcoin). The mining task is to solve bitcoin blocks whose Coinbase scripSig contains a hash pointer to Namecoin (or any other altcoin) block. This is shown in the diagram below. If a miner manages to solve a hash at the bitcoin blockchain difficulty level, the bitcoin block is built and becomes part of the bitcoin network. In this case, the Namecoin hash is ignored by the bitcoin blockchain. On the other hand, if a miner solves a block at Namecoin blockchain difficulty level a new block is created in the Namecoin blockchain. The core benefit of this scheme is that all the computational power spent by the miners contributes towards securing both Namecoin and bitcoin:



Merged mining diagram

Obtaining Name coins

Even though Name coins can be mined independently, they are usually mined as part of bitcoin mining by utilizing the merged mining technique as explained above. This way Name coin can be mined as a by-product of bitcoin mining. Solo mining is no longer profitable as is evident from the following difficulty graph; instead it is recommended to merge-mine, use a mining pool, or even use a crypto currency exchange to buy Name coin.



For example, paying BTC to receive NMC is shown as follows:

Instant Rate 1 BTC = 3114.84374999 NMC

Deposit Min 0.00000300 BTC | Deposit Max 0.14532464 BTC | Liquidity ooooo

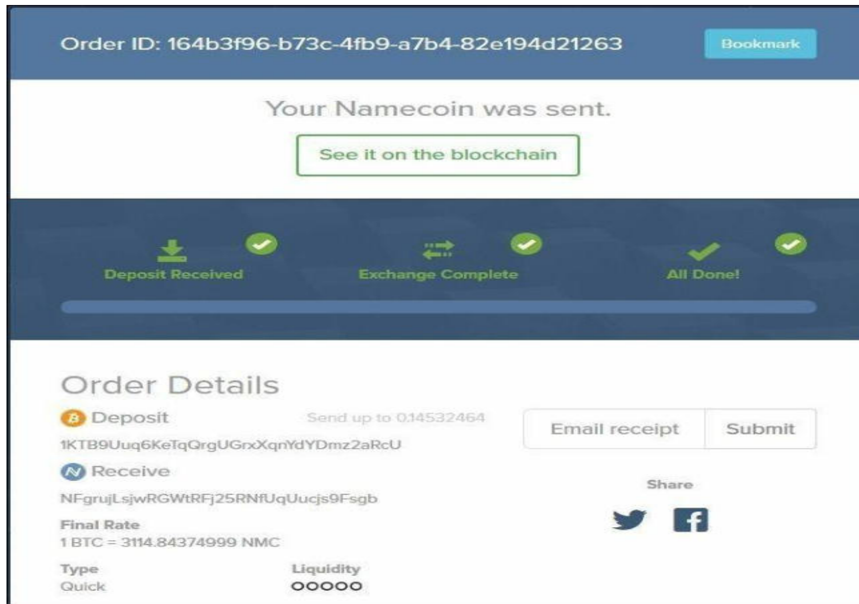
NFgrujLsjwRGWtRFj25RNfUqUucjs9Fsgb

14Koadj8xLpAeKDFke8qVWX5ETeU81amxH

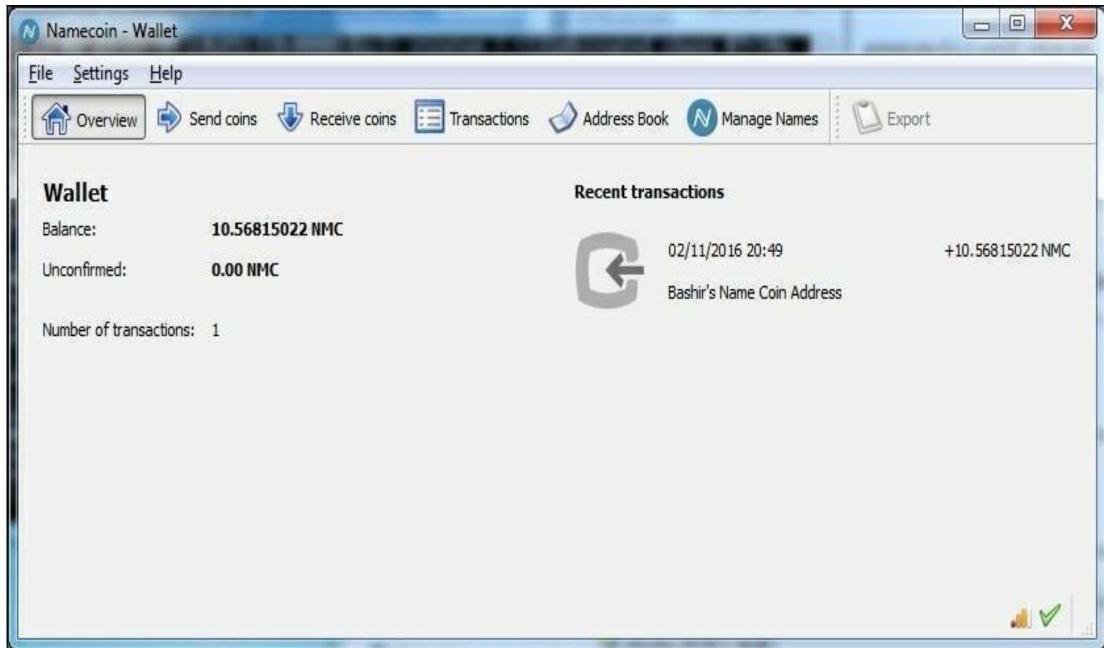
I agree to Terms — Miner Fee: NMC

Start Transaction

Once **Start Transaction** is clicked, the transaction starts and instructs the user to send the bitcoins to a specific bitcoin address. When the user sends the required amount, the conversion process starts as shown:



When the process completes, the transactions can be viewed in the Namecoin wallet:

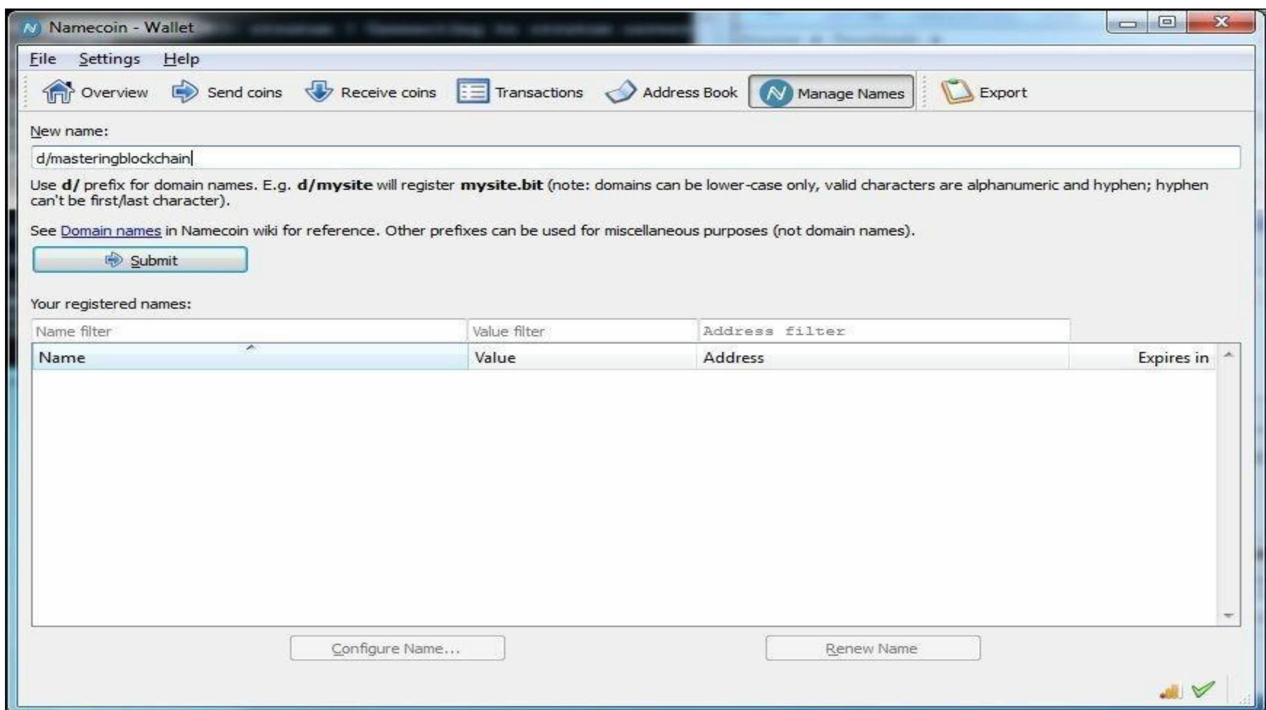


It may take some time to confirm the transactions; until that time it is not possible to use the Namecoins to manage names. Once Namecoins are available in the wallet, the **Manage Names** option can be used to generate Namecoin records.

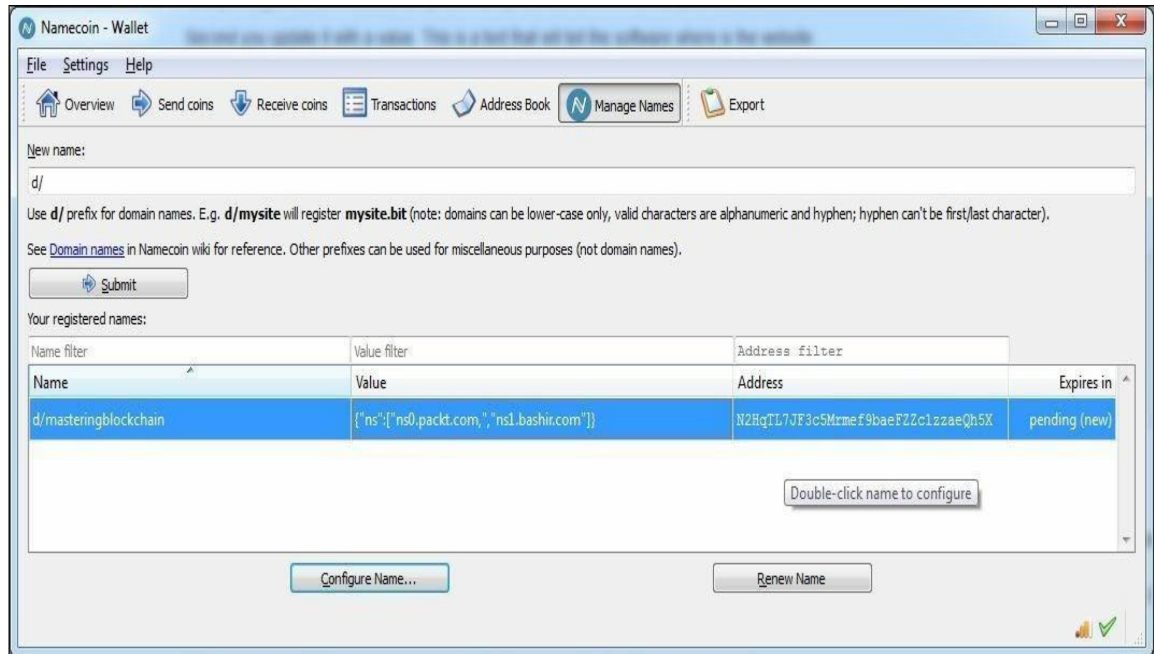
Generating Namecoin records

Namecoin records are in the form of key and value pairs. A name is a lower-case string of the form d/examplename whereas a value is a case-sensitive, UTF-8 encoded JSON object with a maximum of 520 bytes. The name should be RFC1035 (https://tools.ietf.org/html/rfc1035)-compliant. A general namecoin name can be an arbitrary binary string up to 255 bytes long with, 1024-bits of associated identifying information. A record on a Namecoin chain is only valid for around 200 days or 36,000 blocks after which it needs to be renewed. Namecoin also introduced .bit top level domains that can be registered using Namecoin and can be browsed using specialized Namecoin-enabled resolvers.
Namecoin wallet software as shown in the following screenshot can be used to register .bitdomain names.

The name is entered and, after the **Submit** button is pressed, it will ask for configuration information such as DNS, IP, or Identity:



As shown in the following screenshot, **masteringblockchain** will register as masteringblockchain.biton the Namecoin blockchain:



Litecoin

Litecoin is a fork of the bitcoin source code released in 2011. It uses Scrypt as PoW, originally introduced in the Tenebrix coin. Litecoin allows for faster transactions as compared to bitcoin due to its faster block generation time of 2.5 minutes. Also difficulty readjustment is achieved every 3.5 days roughly due to faster block generation time. The total coin supply is 84 million.

Scrypt is a sequentially memory hard function that is the first alternative to the SHA-256-based PoW algorithm. It was originally proposed as a password-based key derivation function PBKDF. The key idea is that if the function requires large number of memory to run then custom hardware such as ASICs will require more VLSI area, which would be unfeasible to build. The Scrypt algorithm requires a large array of pseudo random bits to be held in memory and a key is derived from this in a pseudo random fashion. The algorithm is based on a phenomenon called **Time-Memory Tradeoff** (**TMTO**). If memory requirements are relaxed then it results in increased computational cost. Put another way, TMTO shortens the running time of a programme if more memory is given to it. This tradeoff makes it unfeasible for an attacker to gain more memory because it's expensive and difficult to implement on custom hardware, or if the attacker chooses to not increase memory, then it results in the algorithm running slowly due to high processing requirements.

Scrypt uses the following parameters to generate a derived key (Kd):

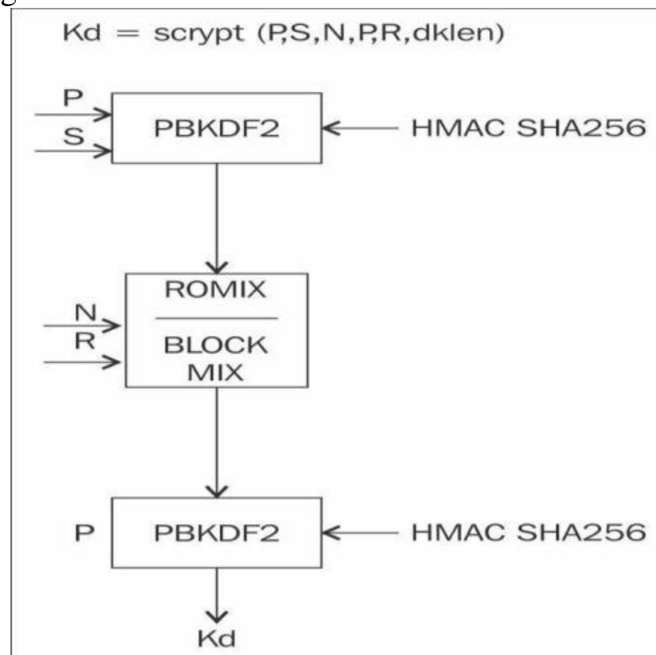
- Passphrase: This is a string of characters to hash
- Salt: This is a random string that is provided to Scrypt functions (generally all hash functions) in order to provide a defence against brute-force dictionary attacks using rainbow tables
- N: This is a memory/CPU cost parameter that must be a power of 2 > 1
- P: The parallelization parameter
- R: The block size parameter
- dkLen: The intended length of the derived key in bytes Formally, this

function can be written as follows:

Kd = scrypt (P, S, N, P, R, dkLen)

Before applying the core Scrypt function, the algorithm takes Pand S as input and applies **PBKDF2** and SHA-256-based HMAC. Then the output is fed to an algorithm called **ROMix,** which internally uses the Blockmix algorithm utilizing the Salsa20/8 core stream cipher to fill up the memory which requires large memory to operate, thus enforcing the sequentially memory hard property.

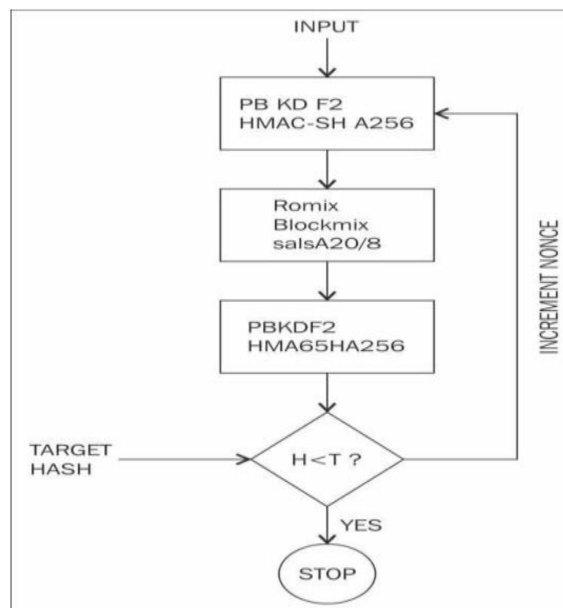
The output from this step of the algorithm is finally fed to the PBKDF2function again in order to produce a derived key. This process is shown in the following diagram:



Scrypt algorithm

Scrypt is used in litecoin mining with specific parameters where N= 1024, R = 1, P=1, and S= random 80 bytes producing a 256-bit output.

It appears that, due to the selection of these parameters, the development of ASICs for Scrypt for litecoin mining turned out to be not very difficult. In an ASIC for litecoin mining, a sequential logic can be developed that takes the data and nonce as input and applies the **PBKDF2** algorithm with **HMAC-SHA256**; then the resultant bit stream is fed into the SALSA20/8 function which produces a hash that again is fed down to the PBKDF2and HMAC-256 functions to produce a 256-bit hash output. As is the case with bitcoin PoW, in Scrypt also if the output hash is less that the target hash (already passed as input at the start, stored in memory, and checked with every iteration) then the function terminates; otherwise, the nonce is incremented and the process is repeated again until a hash is found that is lower than the difficulty target:



Scrypt ASIC design simplified flowchart

- **Trading Litecoin**: As with other coins, trading litecoin is easily carried out on various online exchanges. The current market cap of litecoin is £161,239,005. The current price of litecoin is
£3.25/LTC.
- **Mining**: Litecoin mining can be carried out solo or in pools. At the moment, ASICs for Scrypt are available that are commonly used to mine litecoin.

Litecoin mining on a CPU is no longer profitable as is the case with many other digital currencies. There are online cloud mining providers and ASIC miners available that can be used to mine litecoin. Litecoin mining started

from the CPU, progressed through GPU mining rigs, and eventually now has reached a point where specialized ASIC miners such as ASIC Scrypt Miner Wolf available from EhsMiner are now required to be used in the hope of being able to make some coins. Generally, it is true that even with ASICs it is better to mine in pools instead of solo as solo mining is not as profitable as mining in pools due to the proportional rewards scheme used by mining pools. These miners are capable of producing a hashing rate of 2 Gh/s for Scrypt algorithms.

UNIT- IV

Introduction to smart contracts. This is not a new concept, but, with the advent of blockchain, interest in this concept has revived, and this is now an active area of research in the blockchain space. Due to the cost saving benefits that smart contracts can bring to the financial services industry by reducing the cost of transactions and simplifying complex contracts, rigorous research is being carried out by various financial and academic institutions in order to formalize and make the implementation of smart contracts easy and practical, as soon as possible.

History

Smart contracts were first theorized by *Nick Szabo* in the late 1990s, but it was almost 20 years before the true potential and benefits of them were truly appreciated. Smart contracts are described by *Szabo* as follows:

"A smart contract is a computerized transaction protocol that executes the terms of a contract. The general objectives are to satisfy common contractual conditions (such as payment terms, liens, confidentiality, and even enforcement), minimize exceptions both malicious and accidental, and minimize the need for trusted intermediaries. Related economic goals include lowering fraud loss, arbitrations and enforcement costs, and other transaction costs."

This idea of smart contracts was implemented in a limited fashion in bitcoin in 2009, where bitcoin transactions can be used to transfer the value between users, over a peer-to-peer network where users do not necessarily trust each other and there is no need for a trusted intermediary.

Definition

There is no consensus on a standard definition of smart contracts. It is essential to define what a smart contract is, and the following is the author's attempt to provide a generalized definition of a smart contract.

Note: A smart contract is a secure and unstoppable computer program representing an agreement that is automatically executable and enforceable.

Dissecting this definition further reveals that a smart contract is in fact a computer program that is written in a language that a computer or target machine can understand. Also, it encompasses agreements between parties in the form of business logic. Another key idea is that smart contracts are automatically executed when certain conditions are met. They are enforceable, which means that all contractual terms are executed as defined and expected, even in the presence of adversaries.

Enforcement is a broader term that encompasses traditional enforcement in the form of law, along with implementation of certain measures and controls that make it possible to execute contract terms without requiring any mediation. It should be noted that true smart contracts should not rely on traditional methods of enforcement. Instead, they should work on the principle that *code is law*, meaning that there is no need for an arbitrator or a third party to control or influence the execution of the smart contract. Smart contracts are self-enforcing as opposed to legally enforceable. This might be regarded as a libertarian's dream, but it is entirely possible, and is in line with the true spirit of smart contracts.

Smart contracts usually operate by managing their internal state using a state machine model. This allows development of an effective framework for programming smart contracts, where the state of a contract is advanced further based on some predefined criteria and conditions.

Smart contracts are inherently required to be deterministic in nature. This property will allow a smart contract to be run by any node on a network and achieve the same result. If the result differs even slightly between nodes, consensus then cannot be achieved and a whole paradigm of distributed consensus on blockchain can fail. Moreover, it is also desirable that the contract language itself is deterministic thus ensuring the integrity and stability of the smart contracts. By deterministic in the sense that there are no non-deterministic functions used in the language which can produce varied results on different nodes. For example, various floating point operations calculated by various functions in a variety of programming languages can produce different results in different runtime environments. Another example is of some math functions in JavaScript which can produce different results for the same input on different browsers, and which can in turn lead to various bugs. This is highly undesirable in smart contracts because, if results are inconsistent between nodes, then consensus will never be achieved. A deterministic feature ensures that smart contracts always produce the same output for a specific input. In other words, programs once compiled produce a solid and accurate business logic that is completely in line with the requirements programmed in the high level code.

In summary, a smart contract has the following four properties:

- Automatically executable
- Enforceable
- Semantically sound
- Secure and unstoppable.

The first two properties are required as a minimum, whereas the latter two may not be required or implementable in certain scenarios and can be relaxed. For example, a derivatives contract does not perhaps need to be semantically sound and unstoppable but should at least be automatically executable and enforceable at a basic level. On the other hand, a title deed needs to be semantically sound and complete therefore, in order for it to be implemented as a smart contract, the language must be understood by both computers and people. This issue of interpretation was addressed by *Ian Grigg* with his invention of **Ricardian contracts**, which we will look at in more detail in the next section.

Ricardian contracts

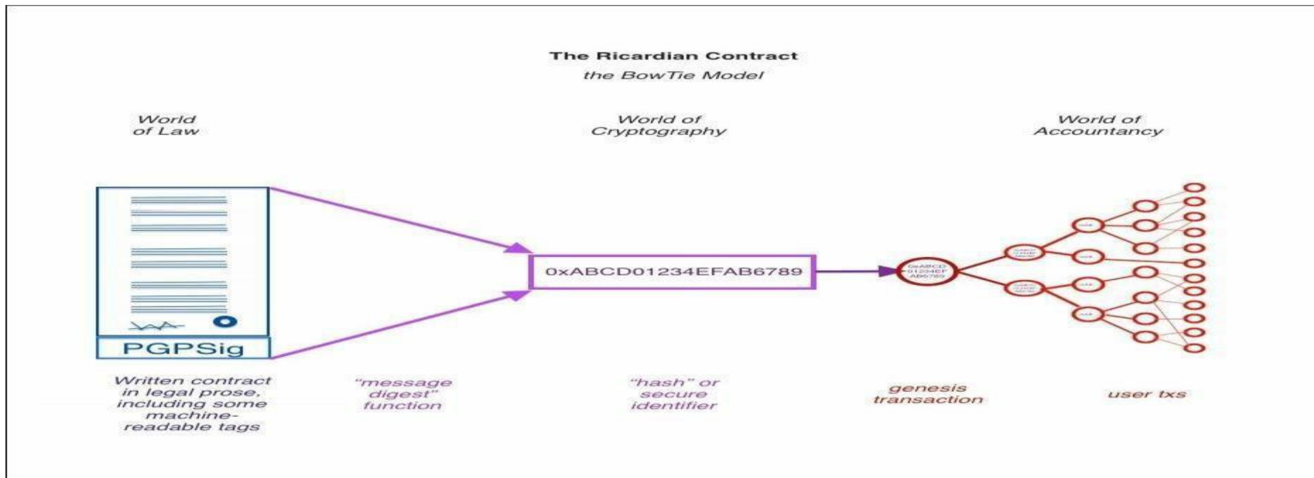
Ricardian contracts were originally proposed in the *Financial Cryptography in 7 Layers* paper by *Ian Grigg* in late 1990s. These contracts were used initially in a bond trading and payment system called **Ricardo**. The key idea is to write a document which is understandable and acceptable by both a court of law and computer software. Ricardian contracts address the challenge of issuance of value over the Internet. It identifies the issuer and captures all the terms and clauses of the contract in a document in order to make it acceptable as a legally binding contract. Based on the original definition by *Ian Grigg* at http://iang.org/papers/ricardian_contract.html, a Ricardian contract is a document that has several of the following properties:

- A contract offered by an issuer to holders
- A valuable right held by holders, and managed by the issuer
- Easily readable by people (like a contract on paper)
- Readable by programs (parseable, like a database)
- Digitally signed
- Carries the keys and server information
- Allied with a unique and secure identifier

The contracts are implemented by producing a single document that contains the terms of the contract in legal prose and the required machine-readable tags. This document is digitally signed by the issuer using their private key. This document is then hashed using a message digest function to produce a hash by which the document can be identified. This hash is then further used and signed by parties during the performance of the contract in order to link each transaction, with the identifier hash thus serving as evidence of intent. This is depicted in the diagram below, usually called a *bowtie* model.

The diagram below shows the **World of Law** on the left hand side from where the document originates. It is then hashed and the resultant message digest is used as an indentifier throughout the **World of Accountancy**. The **World of Accountancy** can basically represent any or multiple accounting, trading and information systems that are being used in a business to perform various business operations. The idea behind this flow is that the message digest generated by hashing the document is first used in a so called *genesis transaction*, or first transaction, and then used in every transaction as an indentifier throughout the operational execution of the contract.

Asecure link is created between the original written contract and every transaction in the *World of Accounting*.



Ricardian contracts, bowtie diagram

A Ricardian contract is different from a smart contract in the sense that a smart contract does not include any contractual document and is focused purely on the execution of the contract. A Ricardian contract, on the other hand, is more concerned with the semantic richness and production of a document that contains contractual legal prose. The semantics of a contract can be divided into two types: operational semantics and denotational semantics. The first type defines the actual execution, correctness and safety of the contract, and the latter is concerned with the real-world meaning of the full contract. Some researchers have differentiated between smart contract code and smart legal contracts where a smart contract is only concerned with the execution of the contract and the second type encompasses both the denotational and operational semantics of a legal agreement. It makes sense to perhaps categorize smart contracts based on the difference between semantics, but it is better to consider smart contracts as a standalone entity that is capable of encoding legal prose and code (business logic) in it.

At bitcoin, a very simple implementation of a smart contract can be observed which is fully oriented towards the execution of the contract, whereas a Ricardian contract is more geared towards producing a document that is understandable by humans, with some parts that a computer program can understand. This can be viewed as legal semantics vs operational performance (semantics vs performance) as shown in the following diagram. This was originally proposed by *Ian Grigg* in his paper *On the intersection of Ricardian and smart contracts*.

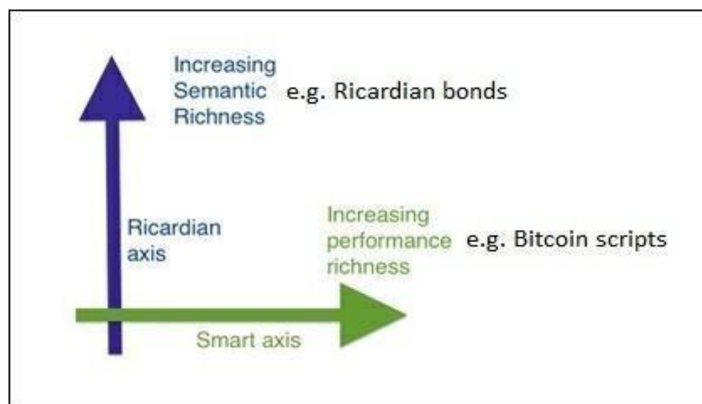


Diagram explaining performance v. semantics are orthogonal issues as described by Ian Grigg; slightly modified to show examples of different types of contracts on both axis

A smart contract is made up to have both of these elements (performance and semantics) embedded together, which completes an ideal model of a smart contract.

A Ricardian contract can be represented as a tuple of three objects, namely *Prose*, *parameters* and *code*. Prose represents the legal contract in regular language; code represents the program that is a computer-understandable representation of legal prose; and parameters join the appropriate parts of the legal contract to the equivalent code.

Ricardian contracts have been implemented in many systems, such as Common Accord, OpenBazaar, OpenAssets, and Askemos.

Smart contract templates

Smart contracts can be implemented for any industry where required but most current use cases are related to the financial industry. Recent work in smart contract space specific to the financial industry has proposed the idea of smart contract templates. The idea is to build standard templates that provide a framework to support legal agreements for financial instruments. This was proposed by ***Clack*** in their paper named ***Smart Contract Templates: Foundations, design landscape and research directions*.** The paper also proposed that domain-specific languages should be built in order to support design and implementation of smart contract templates. A language named *CLACK*, a common language for augmented contract knowledge has been proposed and research has begun to develop the language. This language is intended to be very rich and provide a large variety of functions ranging from supporting legal prose to the ability to be executed on multiple platforms and cryptographic functions.

Contracts in the finance industry is not a new concept and various domain-specific language DSLs are already in use in the financial industry to provide specific language for a specific domain. For example, there are DSLs available that support development of insurance products, represent energy derivatives, or are being used to build trading strategies.

It's important to understand the concept of domain-specific languages. These languages are developed with limited expressiveness for a particular application or area of interest. **Domain-specific languages** (**DSLs**) are different from **general-purpose programming languages** (**GPLs**): DSLs have a small set of features that are sufficient and optimized for the domain they are intended to be used in and, unlike GPLs, are usually not used to build general purpose large application programmes. Based on the design philosophy of DSLs it can be envisaged that such languages can be developed specifically to write smart contracts. Some work has already been done and Solidity is one such language that has been introduced with Ethereum blockchain to write smart contracts. Serpent is another language that has been introduced with Ethereum even though it's not used as much as Solidity.

This idea of domain-specific languages for smart contract programming can be further extended to a graphical domain-specific language, a smart contract modelling platform where a domain expert (not a programmer) can use a graphical user interface and a canvas to define and draw the semantics and performance of a financial contract. Once the flow has been drawn and completed, it can be emulated first to test and then to deploy from the same system to the target platform, which can be a blockchain. This is also not a new concept and a similar approach is used in the Tibco streambase product, which is a Java based system used for building event-driven high frequency trading systems.

It is proposed that research should also be conducted in the area of developing high level DSLs that can be used to programme a smart contract in a user friendly graphical user interface thus allowing a non-programmer to design a smart contract.

Oracles

Oracles are an important component of the smart contract ecosystem. The limitation with smart contracts is that they cannot access external data which might be required to control the execution of the business logic; for example, the stock price of a security that is required by the contract to release the dividend payments. Oracles can be used to provide external data to smart contracts. An Oracle is an interface that delivers data from an external source to smart contracts. Depending on the industry and requirements, Oracles can deliver different types of data ranging from weather reports, real- world news, and corporate actions to data coming from **Internet of Things** (**IoT**) devices. Oracles are trusted entities that use a secure channel to transfer data to a smart contract.

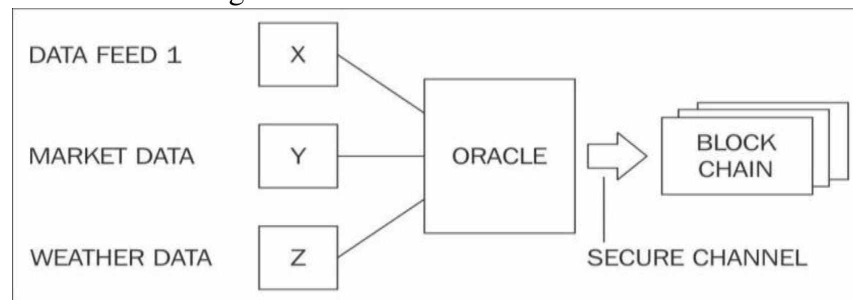
Oracles are also capable of digitally signing the data proving that the source of the data is authentic. Smart contracts can then subscribe to the Oracles and the smart contracts can either pull the data or Oracles can push the data to the smart contracts. It is also necessary that Oracles should not be able to manipulate the data they provide and must be able to provide authentic data. Even though Oracles are trusted, it may still be possible in some cases that the data is incorrect due to manipulation.

Another type of Oracle, which essentially emerged due to the decentralization requirements, can be called *decentralized* Oracles. These types of Oracles can be built based on some distributed mechanism. It can also be envisaged that the Oracles can themselves source data from another blockchain which is driven by distributed consensus, thus ensuring the authenticity of data. For example, one institution running their own private blockchain can publish their data feed via an Oracle that can then be consumed by other blockchains.

Another concept of hardware Oracles is also introduced by researchers where real-world data from physical devices is required. For example, this can be used in telemetry and IoT. However, this approach however requires a mechanism in which hardware devices cannot be tampered with. This can be achieved by using tamper-proof devices.

The following diagram shows a generic model of an oracle and smart contract ecosystem:

A simplified model of an oracle interacting with smart contract on blockchain



Smart Oracles

An idea of Smart Oracle has also been proposed and implemented in *Codius*. Smart Oracles are basically entities just like Oracles, but with the added capability of contract code execution. Smart Oracles proposed by Codius run using Google Native Client. which is a sandboxed environment for running untrusted x86 native code. Codius is available at https://www.codius.org/.

Deploying smart contracts on a blockchain

Smart contracts may or may not be deployed on a blockchain but it makes sense to deploy them on a blockchain due to the distributed consensus mechanism provided by blockchain. Ethereum is an example of a blockchain that natively supports the development and deployment of smart contracts. Smart contracts on Ethereum blockchain are usually part of a larger application such as **Decentralized Autonomous organization** (**DAOs**).

As a comparison, in bitcoin blockchain the `lock_time` field in the bitcoin transaction can be seen as an enabler of a basic version of a smart contract. The `lock_time` field enables a transaction to be locked until a specified time or after a number of blocks, thus enforcing a basic contract that a certain transaction can only be unlocked if certain conditions (elapsed time or number of blocks) is met.
However, this is very limited in nature and should be only viewed as an example of a basic smart contract. In addition to the above mentioned example, bitcoin scripting language, though limited, can be used to construct basic smart contracts. One possibility is to fund a bitcoin address that can be spent by anyone who demonstrates a hash collision attack.

The DAO

The DAO is one of the highest crowdfunded projects, and started in April 2016. This was basically a set of smart contracts written in order to provide a platform for investment. Due to a bug in the code this was hacked in June 2016 and an equivalent of 50 million dollars was siphoned out of the DAO into another account. This resulted in a hard fork on Ethereum in order to recover from the attack. It should be noted that the notion of *code is law*, or unstoppable smart contracts, should be viewed with some scepticism as the implementation of these concepts is not mature enough to warrant full and

unquestionable trust. This is evident from the recent events where the Ethereum foundation was able to stop and change the execution of *The DAO* by introducing a hard fork. Though this hard fork was introduced for genuine reasons, it goes against the true spirit of decentralization and the notion of *code is law*. On the other hand, resistance against this hard fork and some miners who decided to keep mining on the original chain resulted in the creation of Ethereum Classic. This is the original, non-forked Ethereum blockchain where *code is still law*.

This attack highlights the dangers of smart contracts and the absolute need to develop a formal language for smart contracts. The attack also highlighted the importance of thorough testing. There have been various vulnerabilities discovered in Ethereum recently around the smart contract development language. Therefore it is of utmost importance that a standard framework is developed to address all these issues. Some work has already begun as discussed previously, but this area is ripe for more research in order to address limitations in smart contract languages.

Ethereum

Introduction

Ethereum was conceptualized by *Vitalik Buterin* in November 2013. The key idea proposed was the development of a Turing-complete language that allows the development of arbitrary programs (smart contracts) for blockchain and decentralized applications. This is in contrast to bitcoin, where the scripting language is very limited and allows basic and necessary operations only.

Ethereum clients and releases

Various Ethereum clients have been developed using different languages and currently most popular are go-Ethereum and parity. go-Ethereum was developed using Golang, whereas parity was built using Rust. There are other clients available too, but usually, the go-Ethereum client known as *geth* is sufficient for all purposes. Mist is a user-friendly **Graphical User Interface** (**GUI**) wallet that runs geth in the background to sync with the network. More details on this will be provided later in the chapter, in the installation and mining section.

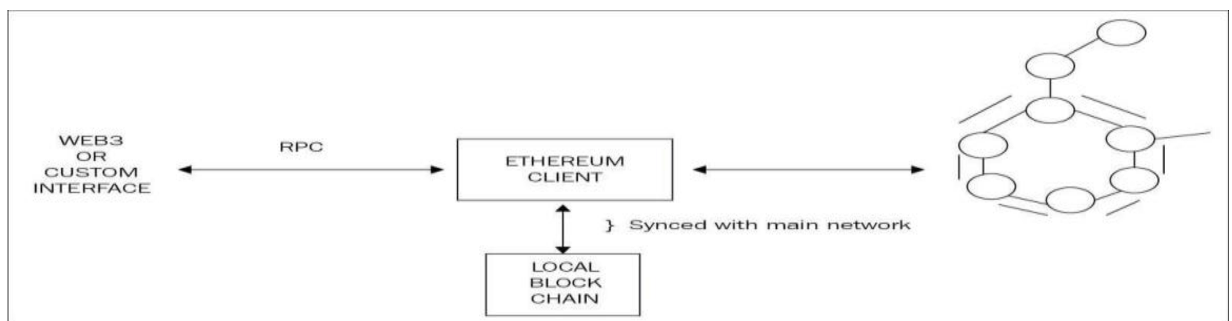
The first release of Ethereum was known as *Frontier*, and the current release of Ethereum is called *homestead release*. The next version is named metropolis and it focuses on protocol simplification and performance improvement. The final release is named *serenity*, which is envisaged to have a Proof of Stake algorithm (Casper) implemented with it. Other areas of research targeted with serenity include scalability, privacy, and **Ethereum virtual machine** (**EVM**) upgrade. As this is a continuous development effort and the Ethereum ecosystem will undergo constant improvement and development, serenity should not really be considered a *final* version but a major milestone in a long journey of continuous improvement. Further releases are envisaged but have not been named yet. The vision of *web 3.0* has already been proposed and is being discussed in the community. Web 3.0 is a concept that basically proposes a semantic and intelligent web as an evolution of the existing web 2.0 technology. This is the vision of an ecosystem where people, applications, data, and web are all connected together

and are able to interact with each other in an intelligent fashion. With the advent of the blockchain technology, an idea of decentralized web has also emerged, which in fact was the original vision of the Internet. The core idea is that all major services, such as DNS, search engines, and identity on the Internet will be decentralized in web 3.0. This is where Ethereum is being envisaged as a platform that can help realize this vision.

The Ethereum stack

The Ethereum stack consists of various components. At the core, there is the Ethereum blockchain running on the P2P Ethereum network. Secondly, there's an Ethereum client (usually geth) that runs on the nodes and connects to the peer-to-peer Ethereum network from where blockchain is downloaded and stored locally. It provides various functions, such as mining and account management. The local copy of the blockchain is synchronized regularly with the network. Another component is the `web3.js` library that allows interaction with geth via the **Remote Procedure Call** (**RPC**) interface.

This can be visualized in the following diagram:

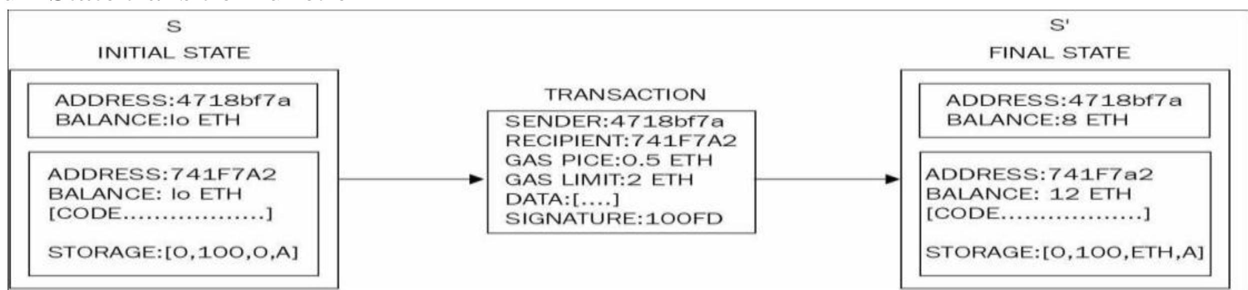


The Ethereum stack showing various components

Ethereumblockchain

Ethereum, just like any other blockchain, can be visualized as a transaction-based state machine. This is mentioned in the Ethereum yellow paper written by *Dr. Gavin Wood*. The idea is that a genesis state is transformed into a final state by executing transactions incrementally. The final transformation is then accepted as the absolute undisputed version of the state. In the following diagram, the Ethereum state transition function is shown, where a transaction execution has resulted in a state transition.

Ethereum State transition function



In the preceding example, a transfer of 2 Ether from **Address 4718bf7a** to **Address 741f7a2** is initiated. The initial state represents the state before the transaction execution and the final state is what the morphed state looks like. This will be discussed in more detail later in the chapter, but the aim of this example is to introduce the core idea of state transition in Ethereum.

Currency (ETH and ETC)

As an incentive to the miners, Ethereum also rewards its native currency called Ether, abbreviated as ETH. After the DAO hack (described later), a hard fork was proposed in order to mitigate the issue; therefore, there are now two Ethereum blockchains: one is called Ethereum classic and its currency is represented by ETC, whereas the hard-forked version is ETH, which continues to grow and on which active development is being carried out. ETC, however, has its own following with a dedicated community that is further developing ETC, which is the nonforked original version of Ethereum. This chapter is focused mainly on ETH, which is the currently the most active and official Ethereum blockchain.

Forks

With the latest release of homestead, due to major protocol upgrades, it resulted in a hard fork. The protocol was upgraded at block number 1,150,000, resulting in the migration from the first version of Ethereum known as Frontier to the second version of Ethereum called homestead.

A recent unintentional fork that occurred on November 24, 2016, at 14:12:07 UTC was due to a bug in the geth client's journaling mechanism. Network fork occurred at block number 2,686,351. This bug resulted in geth failing to revert empty account deletions in the case of the empty out-of-gas exception. This was not an issue in parity (another popular Ethereum client). This means that from block number 2686351, the Ethereum blockchain is split into two, one running with parity clients and the other with geth. This issue was resolved with the release of geth version 1.5.3.

Gas

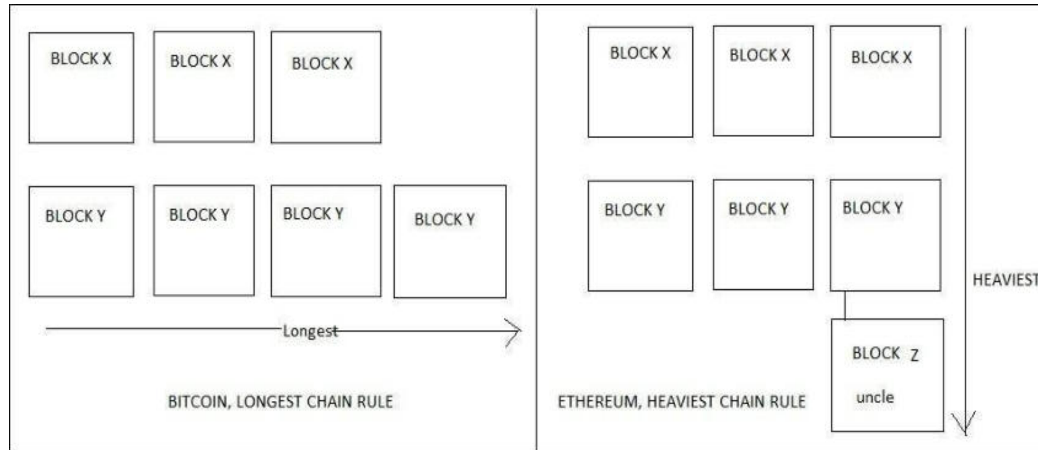
Another key concept in Ethereum is that of gas. All transactions on the Ethereum blockchain are required to cover the cost of computation they are performing. The cost is covered by something called *gas* or *crypto fuel*, which is a new concept introduced by Ethereum. This gas as *execution fee* is paid upfront by the transaction originators. The *fuel* is consumed with each operation. Each operation has a predefined amount of gas associated with it. Each transaction specifies the amount of gas it is willing to consume for its execution. If it runs *out of gas* before the execution is completed, any operation performed by the transaction up to that point is rolled back. If the transaction is successfully executed, then any remaining gas is refunded to the transaction originator.

The consensus mechanism

The consensus mechanism in Ethereum is based on the GHOST protocol originally proposed by *Zohar* and *Sompolinsky* in December 2013

Ethereum uses a simpler version of this protocol, where the chain that has most computational effort spent on it in order to build it is identified as the definite version. Another way of looking at it is to find the longest chain, as the longest chain must have been built by consuming adequate mining effort. **Greedy Heaviest Observed Subtree** (**GHOST**) was first introduced as a mechanism to alleviate the issues arising out of fast block generation times that led to stale or orphan blocks. In GHOST, stale blocks are added in calculations to figure out the longest and heaviest chain of blocks. Stale blocks are called Uncles or Ommers in Ethereum.

The following diagram shows a quick comparison between the longest and heaviest chain:



Longest versus heaviest chain

The world state

The world state in Ethereum represents the global state of the Ethereum blockchain. It is basically a mapping between Ethereum addresses and account states. The addresses are 20 bytes long. This mapping is a data structure that is serialized using **Recursive Length Prefix** (**RLP**). RLP is a specially developed encoding scheme that is used in Ethereum to serialize binary data for storage or transmission over the network and also to save the state in a Patricia tree. The RLP function takes an item as an input, which can be a string or a list of items, and produces raw bytes that are suitable for storage and transmission over the network. RLP does not encode data; instead, its main purpose is to encode structures.

The account state : The account state consists of four fields: nonce, balance, storageroot and codehash and is described in detail here.

Nonce : This is a value that is incremented every time a transaction is sent from the address. In case of contract accounts, it represents the number of contracts created by the account. Contract accounts are one of the two types of accounts that exist in Ethereum; they will be explained later on in the chapter in more detail.

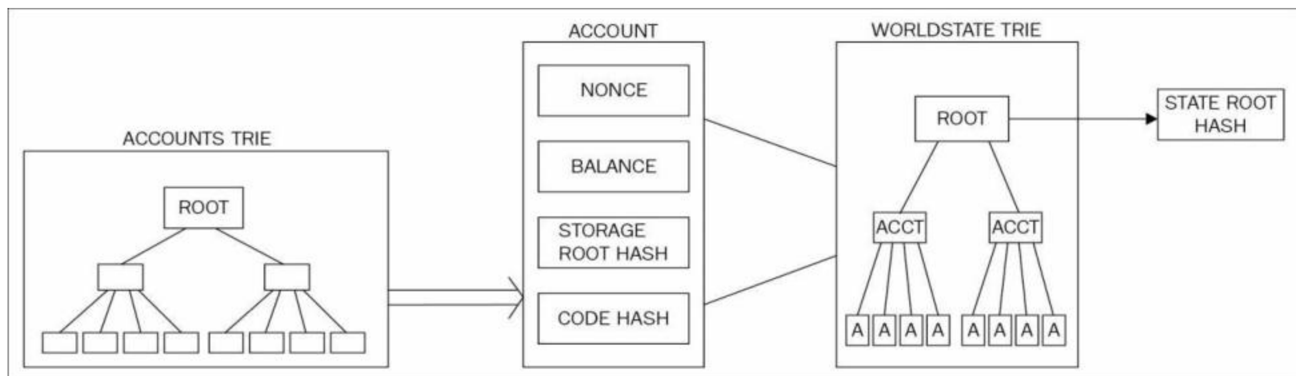
Balance : This value represents the number of Weis which is the smallest unit of the currency (Ether) in Ethereum held by the address.

Storageroot : This field represents the root node of a Merkle Patricia tree that encodes the storage contents of the account.

Codehash

This is an immutable field that contains the hash of the smart contract code that is associated with the account. In the case of normal accounts, this field contains the Keccak 256-bit hash of an empty string. This code is invoked via a message call.

The world state and its relationship with accounts trie, accounts, and block header can be visualized in the following diagram. It shows the account data structure in the middle of the diagram, which contains a storage root hash derived from the root node of the account storage trie shown on the left. The account data structure is then used in the world state trie, which is a mapping between addresses and account states. Finally, the root node of the world state trie is hashed using the Keccak 256-bit algorithm and made part of the block header data structure, which is shown on the right-hand side of the diagram as state root hash.



Accounts trie (storage contents of account), account tuple, world state trie, and state root hash and their relationship Accounts trie is basically a Merkle Patricia tree used to encode the storage contents of an account. The contents are stored as a mapping between keccak 256-bit hashes of 256-bit integer keys to the RLP-encoded 256-bit integer values.

Transactions

A transaction in Ethereum is a digitally signed data packet using a private key that contains the instructions that, when completed, either result in a message call or contract creation. Transactions can be divided into two types based on the output they produce:

- **Message call transactions**: This transaction simply produces a message call that is used to pass messages from one account to another.
- **Contract creation transactions**: As the name suggests, these transactions result in the creation of a new contract. This means that when this transaction is executed successfully, it creates an account with the associated code.

Both of these transactions are composed of a number of common fields, which are described here.

Nonce : Nonce is a number that is incremented by one every time a transaction is sent by the sender. It must be equal to the number of transactions sent and is used as a unique identifier for the transaction. A nonce

value can only be used once.

gasPrice : The `gasPrice` field represents the amount of Wei required in order to execute the transaction.

gasLimit : The `gasLimit` field contains the value that represents the maximum amount of gas that can be consumed in order to execute the transaction. The concept of gas and gas limit will be covered later in the chapter in more detail. For now, it is sufficient to say that this is the amount of fee in Ether that a user (for example, the sender of the transaction) is willing to pay for computation.

To : As the name suggests, the `to` field is a value that represents the address of the recipient of the transaction.

Value : `Value` represents the total number of Wei to be transferred to the recipient; in the case of a contract account, this represents the balance that the contract will hold.

Signature

Signature is composed of three fields, namely *v*, *r*, and *s*. These values represent the digital signature (*R*, *S*) and some information that can be used to recover the public key (*V*). Also of the transaction from which the sender of the transaction can also be determined. The signature is based on ECDSA scheme and makes use of the SECP256k1 curve.

V is a single byte value that depicts the size and sign of the elliptic curve point and can be either 27 or 28. *V* is used in the ECDSA recovery contract as a recovery ID. This value is used to recover (derive) the public key from the private key. In secp256k1, the recovery ID is expected to be either 0 or 1. In Ethereum, this is offset by 27. More details on the ECDSARECOVER function will be provided later in this chapter.

R is derived from a calculated point on the curve. First, a random number is picked up, which is multiplied with the generator of the curve to calculate a point on the curve. The *x* coordinate part of this point is *R*. *R* is encoded as a 32 byte sequence. *R* must be greater than 0 and less than the secp256k1n limit (115792089237316195423570985008687907852837564279074904382605163141518161494337).

S is calculated by multiplying *R* with the private key and adding it into the hash of the message to be signed and by finally dividing it with the random number chosen to calculate *R*. *S* is also a 32 byte sequence. *R* and *S* together represent the signature.

In order to sign a transaction, the `ECDSASIGN` function is used, which takes the message to be signed and the private key as an input and produces *V*, a single byte value; *R*, a 32 byte value, and *S*, another 32 byte value. The equation is as follows:

ECDSASIGN (Message, Private Key) = (V, R, S)

Init

The `Init` field is used only in transactions that are intended to create contracts. This represents a byte array of unlimited length that specifies the EVM code to be used in the account initialization process. The

code contained in this field is executed only once, when the account is created for the first time, and gets destroyed immediately after that.

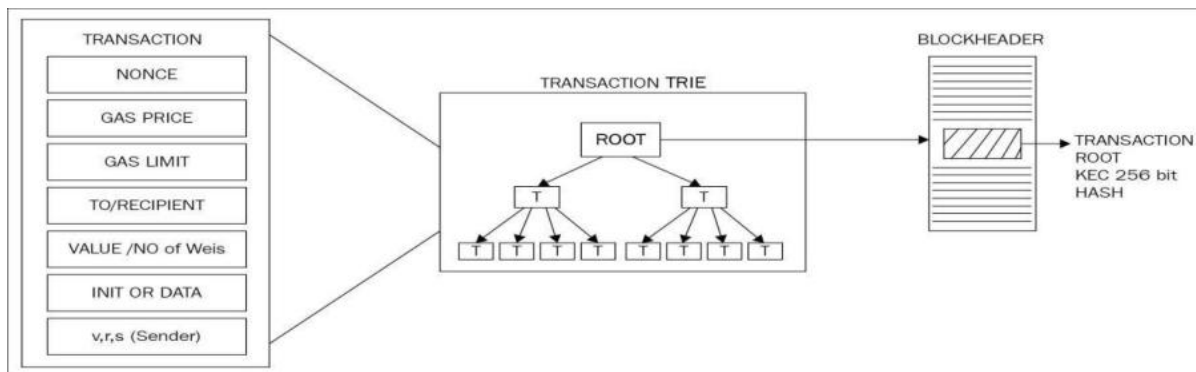
`Init` also returns another code section called *body*, which persists and runs in response to message calls that the contract account may receive. These message calls may be sent via a transaction or an internal code execution.

Data

If the transaction is a message call, then the `data` field is used instead of `init`, which represents the input data of the message call. It is also unlimited in size and is organized as a byte array.

This can be visualized in the following diagram, where a transaction is a tuple of the fields mentioned earlier, which is then included in a transaction trie (a modified Merkle-Patricia tree) composed of the transactions to be included. Finally, the root node of transaction trie is hashed using a Keccak 256-bit algorithm and is included in the block header along with a list of transactions in the block.

Transactions can be found in either transaction pools or blocks. When a mining node starts its operation of verifying blocks, it starts with the highest paying transactions in the transaction pool and executes them one by one. When the gas limit is reached or no more transactions are left to be processed in the transaction pool, the mining starts. In this process, the block is repeatedly hashed until a valid nonce is found that, once hashed with the block, results in a value less than the difficulty target. Once the block is successfully mined, it will be broadcasted immediately to the network, claiming success, and will be verified and accepted by the network. This process is similar to Bitcoin's mining process discussed in the previous chapter. The only difference is that Ethereum's Proof of Work algorithm is ASIC-resistant, known as *Ethash*, where finding a nonce requires large memory.



Relationship between transaction, transaction trie and block header

Contract creation transaction

There are a few essential parameters that are required when creating an account. These parameters are listed as follows:

- Sender
- Original transactor
- Available gas
- Gas price
- Endowment, which is the amount of ether allocated initially
- A byte array of arbitrary length
- Initialization EVM code
- Current depth of the message call/contract-creation stack (current depth means the number of items that are already there in the stack)

Addresses generated as a result of contract creation transaction are 160-bit in length. Precisely, as defined in the yellow paper, they are the rightmost 160-bits of the Keccak hash of the RLP encoding of the structure containing only the sender and the nonce. Initially, the nonce in the account is set to zero. The balance of the account is set to the value passed to the contract. Storage is also set to empty. Code hash is Keccak 256-bit hash of the empty string.

The account is initialized when the EVM code (Initialization EVM code) is executed. In the case of any exception during code execution, such as not having enough gas, the state does not change. If the execution is successful, then the account is created after the payment of appropriate gas costs. The current version of Ethereum (homestead) specifies that the result of contract transaction is either a new contract with its balance, or no new contract is created with no transfer of value. This is in contrast to previous versions, where the contract could be created regardless of the contract code deployment being successful or not due to an out-of-gas exception.

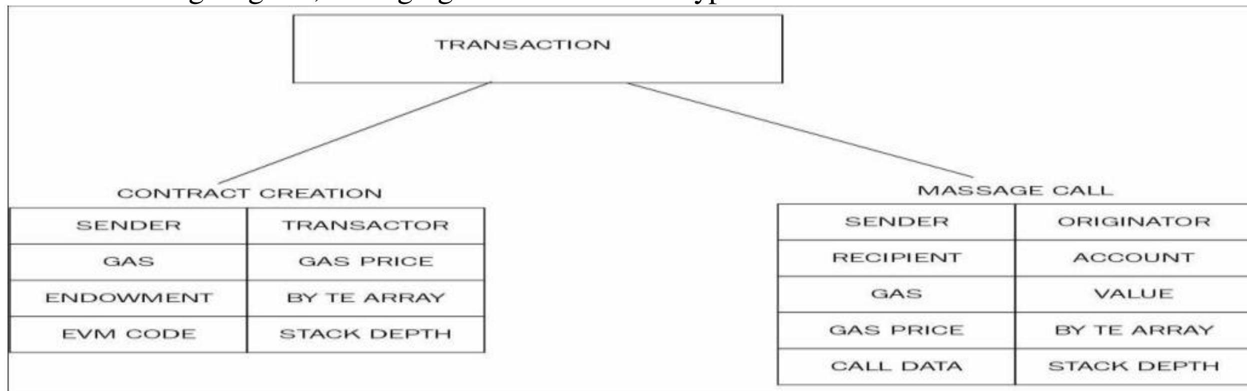
Message call transaction

A message call requires several parameters for execution, which are listed as follows:

- Sender
- The transaction originator
- Recipient
- The account whose code is to be executed
- Available gas
- Value
- Gas price
- Arbitrary length byte array
- Input data of the call
- Current depth of the message call/contract creation stack

Message calls result in state transition. Message calls also produce output data, which is not used if transactions are executed. In cases where message calls are triggered by VM code, the output produced by the transaction execution is used.

In the following diagram, the segregation between two types of transaction is shown:



Elementsof the Ethereumblockchain

In the following section, you will be introduced to various components of the Ethereum network and the blockchain. First, the basic concept of the EVM is given in the next section.

Ethereum virtual machine (EVM)

EVM is a simple stack-based execution machine that runs byte code instructions in order to transform the system state from one state to another. The word size of the virtual machine is set to 256-bit. The stack size is limited to 1024 elements and is based on the **LIFO** (**Last in First Out**) queue. EVM is a Turing-complete machine but is limited by the amount of gas that is required to run any instruction. This means that infinite loops that can result in denial of service attacks are not possible due to gas requirements. EVM also supports exception handling in case exceptions occur, such as not having enough gas or invalid instructions, in which case the machine would immediately halt and return the error to the executing agent.

EVM is a fully isolated and sandboxed runtime environment. The code that runs on the EVM does not have access to any external resources, such as a network or filesystem.

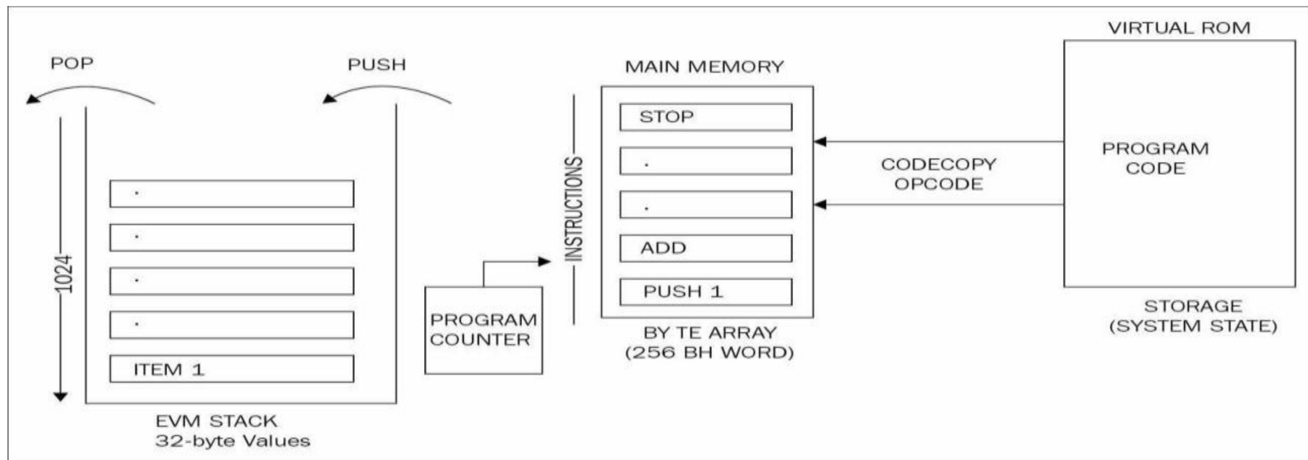
As discussed earlier, EVM is a stack-based architecture. EVM is big-endian by design and it uses 256-bit wide words. This word size allows for Keccak 256-bit hash and elliptic curve cryptography computations.

There are two types of storage available to contracts and EVM. The first one is called memory, which is a byte array. When a contract finishes the code execution, the memory is cleared. It is akin to the concept of RAM. The other type, called storage, is permanently stored on the blockchain. It is a key value store.

Memory is unlimited but constrained by gas fee requirements. The storage associated with the virtual machine is a word addressable *word array* that is nonvolatile and is maintained as part of the system state. Keys and value are 32 bytes in size and storage. The program code is stored in a **virtual read-only memory** (**virtual ROM**) that is accessible using the CODECOPY instruction.

The CODECOPY instruction is used to copy the program code into the main memory. Initially, all storage and memory is set to zero in the EVM.

The following diagram shows the design of the EVM where the virtual ROM stores the program code that is copied into main memory using **CODECOPY**. The main memory is then read by the EVM by referring to the program counter and executes instructions step by step. The program counter and EVM stack are updated accordingly with each instruction execution.



EVM operation

EVM optimization is an active area of research and recent research has suggested that EVM can be optimized and tuned to a very fine degree in order to achieve high performance. Research into the possibility of using **Web assembly** (**WASM**) is underway already. WASM is developed by Google, Mozilla, and Microsoft and is now being designed as an open standard by the W3C community group. The aim of WASM is to be able to run machine code in the browser that will result in execution at native speed. Similarly, the aim of EVM 2.0 is to be able to run the EVM instruction set (Opcodes) natively in CPUs, thus making it faster and efficient.

Execution environment

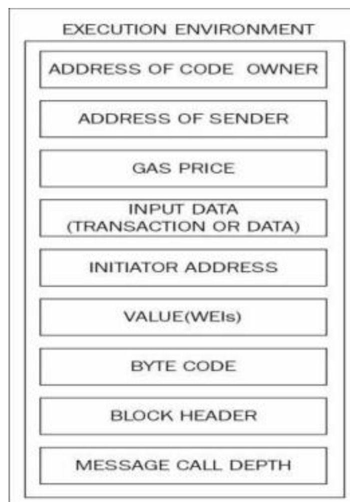
There are some key elements that are required by the execution environment in order to execute the code. The key parameters are provided by the execution agent, for example, a transaction. These are listed as follows:

1. The address of the account that owns the executing code.
2. The address of the sender of the transaction and the originating address of this execution.
3. The gas price in the transaction that initiated the execution.
4. Input data or transaction data depending on the type of executing agent. This is a byte array; in the case of a message call, if the execution agent is a transaction, then the transaction data is included as input data.
5. The address of the account that initiated the code execution or transaction sender. This is the address

of the sender in case the code execution is initiated by a transaction; otherwise, it's the address of the account.

6. The value or transaction value. This is the amount in Wei. If the execution agent is a transaction, then it is the transaction value.
7. The code to be executed presented as a byte array that the iterator function picks up in each execution cycle.
8. The block header of the current block
9. The number of message calls or contract creation transactions currently in execution. In other words, this is the number of CALLs or CREATEs currently in execution.

The execution environment can be visualized as a tuple of nine elements, as follows:



Execution environment Tuple

In addition to the previously mentioned nine fields, system state and the remaining gas are also provided to the execution environment. The execution results in producing the resulting state, gas remaining after the execution, self-destruct or suicide set (described later), log series (described later), and any gas refunds.

Machine state

Machine state is also maintained internally by the EVM. Machine state is updated after each execution cycle of EVM. An iterator function (detailed in the next section) runs in the virtual machine, which outputs the results of a single cycle of the state machine. Machine state is a tuple that consist of the following elements:

- Available gas
- The program counter, which is a positive integer up to 256
- Memory contents
- Active number of words in memory

- Contents of the stack

The EVM is designed to handle exceptions and will halt (stop execution) in case any of the following exceptions occur:

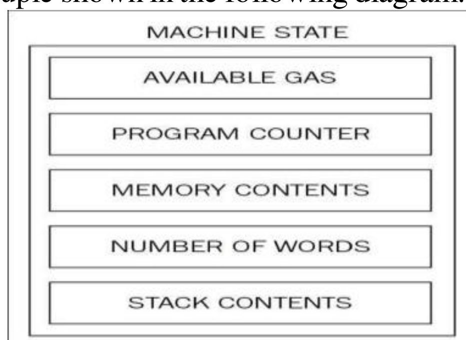
- Not having enough gas required for execution
- Invalid instructions
- Insufficient stack items
- Invalid destination of jump op codes
- Invalid stack size (greater than 1024)

The iterator function

The iterator function mentioned earlier performs various important functions that are used to set the next state of the machine and eventually the world state. These functions include the following:

- It fetches the next instruction from a byte array where the machine code is stored in the execution environment.
- It adds/removes (PUSH/POP) items from the stack accordingly.
- Gas is reduced according to the gas cost of the instructions/Opcodes.
- It increments the **program counter** (**PC**).

Machine state can be viewed as a tuple shown in the following diagram:



Machine state tuple

The virtual machine is also able to halt in normal conditions if STOP or SUICIDE or RETURN Opcodes are encountered during the execution cycle.

Code written in a high-level language such as serpent, LLL, or Solidity is converted into the byte code that EVM understands in order for it to be executed by the EVM. Solidity is the high-level language that has been developed for Ethereum with JavaScript such as syntax to write code for smart contracts. Once the code is written, it is compiled into byte code that's understandable by the EVM using the Solidity compiler called solc.

LLL (**Lisp-like Low-level language**) is another language that is used to write smart contract code. Serpent is a Python-like high-level language that can be used to write smart contracts for Ethereum.

For example, a simple program in solidity is shown as follows:

```
pragma solidity ^0.4.0; contract Test1
{
uint x=2;
function addition1(uint x) returns (uint y)
 {
y=x+2;
}
}
```

This program is converted into bytecode, as shown here. Details on how to compile solidity code with examples will be given in the next chapter.

Runtime byte code

606060405260e060020a6000350463989e17318114601c575b6000565b346000576029600435603b 565b60408051918252519081900360200190f35b600281015b91905056
Opcodes PUSH1 SWAP1 SUB PUSH1 0x20 ADD SWAP1 RETURN JUMPDEST PUSH1 0x2 DUP2 ADD JUMPDEST SWAP2 SWAP1 POP JUMP

Opcodes and their meaning

There are different opcodes that have been introduced in the EVM. Opcodes are divided into multiple categories based on the operation they perform. The list of opcodes with their meaning and usage is presented here.

Arithmetic operations

All arithmetic in EVM is modulo 2^256. This group of opcodes is used to perform basic arithmetic operations. The value of these operations starts from 0x00 up to 0x0b.

Mnemonic	**Value**	**POP**	**PUSH**	**Gas**	**Description**
STOP	0x00	0	0	0	Halts execution
ADD	0x01	2	1	3	Adds two values

MUL	0x02	2	1	5	Multiplies two values
SUB	0x03	2	1	3	Subtraction operation
DIV	0x04	2	1	5	Integer division operation
SDIV	0x05	2	1	5	Signed integer division operation
MOD	0x06	2	1	5	Modulo remainder operation
SMOD	0x07	2	1	5	Signed modulo remainder operation
ADDMOD	0x08	3	1	8	Modulo addition operation
MULMOD	0x09	3	1	8	Module multiplication operation
EXP	0x0a	2	1	10	Exponential operation (repeated multiplication of the base)
SIGNEXTEND	0x0b	2	1	5	Extends the length of 2s complement signed integer

Note that STOP is not an arithmetic operation but is categorized in this list of arithmetic operations due to the range of values (0s) it falls in.

Logical operations

Logical operations include operations that are used to perform comparisons and Boolean logic operations. The value of these operations is in the range of 0x10 to 0x1a.

Mnemonic	Value	POP	PUSH	Gas	Description
LT	0x10	2	1	3	Less than
GT	0x11	2	1	3	Greater than
SLT	0x12	2	1	3	Signed less than comparison
SGT	0x13	2	1	3	Signed greater than comparison
EQ	0x14	2	1	3	Equal comparison
ISZERO	0x15	1	1	3	Not operator
AND	0x16	2	1	3	Bitwise AND operation
OR	0x17	2	1	3	Bitwise OR operation
XOR	0x18	2	1	3	Bitwise exclusive OR (XOR) operation
NOT	0x19	1	1	3	Bitwise NOT operation
BYTE	0x1a	2	1	3	Retrieve single byte from word

Cryptographic operations

There is only one operation in this category named SHA3. It is worth noting that this is not the standard SHA3 standardized by NIST but the original Keccak implementation.

Mnemonic	Value	POP	PUSH	Gas	Description
SHA3	0x20	2	1	30	Used to calculate Keccak 256-bit hash.

Environmental information

There are a total of 13 instructions in this category. These opcodes are used to provide information related to addresses, runtime environments, and data copy operations.

Mnemonic	Value	POP	PUSH	Gas	Description

ADDRESS	0x30	0	1	2	Used to get the address of the currently executing account
BALANCE	0x31	1	1	20	Used to get the balance of the given account
ORIGIN	0x32	0	1	2	Used to get the address of the sender of the original transaction
CALLER	0x33	0	1	2	Used to get the address of the account that initiated the execution
CALLVALUE	0x34	0	1	2	Retrieves the value deposited by the instruction or transaction
CALLDATALOAD	0x35	1	1	3	Retrieves the input data that was passed a parameter with the message call
CALLDATASIZE	0x36	0	1	2	Used to retrieve the size of the input data passed with the message call
CALLDATACOPY	0x37	3	0	3	Used to copy input data passed with the message call from the current environment to the memory.
CODESIZE	0x38	0	1	2	Retrieves the size of running the code in the current environment
CODECOPY	0x39	3	0	3	Copies the running code from current environment to the memory
GASPRICE	0x3a	0	1	2	Retrieves the gas price specified by the initiating transaction.
EXTCODESIZE	0x3b	1	1	20	Gets the size of the specified account code
EXTCODECOPY	0x3c	4	0	20	Used to copy the account code to the memory.

Block Information

This set of instructions is related to retrieving various attributes associated with a block:

Mnemonic	**Value**	**POP**	**PUSH**	**Gas**	**Description**
BLOCKHASH	0x40	1	1	20	Gets the hash of one of the 256 most recently completed blocks

COINBASE	0x41	0	1	2	Retrieves the address of the beneficiary set in the block
TIMESTAMP	0x42	0	1	2	Retrieves the time stamp set in the blocks
NUMBER	0x43	0	1	2	Gets the block's number
DIFFICULTY	0x44	0	1	2	Retrieves the block difficulty
GASLIMIT	0x45	0	1	2	Gets the gas limit value of the block

Stack, memory, storage and flow operations

Mnemonic	Value	POP	PUSH	Gas	Description
POP	0x50	1	0	2	Removes items from the stack
MLOAD	0x51	1	1	3	Used to load a word from the memory.
MSTORE	0x52	2	0	3	Used to store a word to the memory.
MSTORE8	0x53	2	0	3	Used to save a byte to the memory
SLOAD	0x54	1	1	50	Used to load a word from the storage
SSTORE	0x55	2	0	0	Saves a word to the storage
JUMP	0x56	1	0	8	Alters the program counter
JUMPI	0x57	2	0	10	Alters the program counter based on a condition
PC	0x58	0	1	2	Used to retrieve the value in the program counter before the increment.
MSIZE	0x59	0	1	2	Retrieves the size of the active memory in bytes.

GAS	0x5a	0	1	2	Retrieves the available gas amount
JUMPDEST	0x5b	0	0	1	Used to mark a valid destination for jumps with no effect on the machine state during the execution.

Push operations

These operations include PUSH operations that are used to place items on the stack. The range of these instructions is from 0x60 to 0x7f. There are 32 PUSH operations available in total in the EVM. PUSH operation, which reads from the byte array of the program code.

Mnemonic	Value	POP	PUSH	Gas	Description
PUSH1 . . . PUSH 32	0x60 ... 0x7f	0	1	3	Used to place *N* right-aligned big-endian byte item(s) on the the stack. *N* is a value that ranges from 1 byte to 32 bytes (full word) based on the mnemonic used.

Duplication operations

As the name suggests, duplication operations are used to duplicate stack items. The range of values is from 0x80 to 0x8f. There are 16 DUP instructions available in the EVM. Items placed on the stack or removed from the stack also change incrementally with the mnemonic used; for example, DUP1 removes one item from the stack and places two items on the stack, whereas DUP16 removes 16 items from the stack and places 17 items.

Mnemonic	Value	POP	PUSH	Gas	Description
DUP1 . . . DUP16	0x80 ... 0x8f	X	Y	3	Used to duplicate the nth stack item, where *N* is the number corresponding to the DUP instruction used. *X* and *Y* are the items removed and placed on the stack, respectively.

Exchange operations

SWAP operations provide the ability to exchange stack items. There are 16 SWAP instructions available and with each instruction, the stack items are removed and placed incrementally up to 17 items depending on the type of Opcode used.

Mnemonic	Value	POP	PUSH	Gas	Description
SWAP1 . . . SWAP16	0x90 ... 0x9f	X	Y	3	Used to swap the nth stack item, where *N* is the number corresponding to the SWAP instruction used. *X* and *Y* are the items removed and placed on the stack, respectively.

Logging operations

Logging operations provide opcodes to append log entries on the sub-state tuple's log series field. There are four log operations available in total and they range from value 0x0a to 0xa4.

Mnemonic	Value	POP	PUSH	Gas	Description
LOG0 . . . LOG4	0x0a ... 0xa4	X	Y (0)	375, 750, 1125, 1500, 1875	Used to append log record with *N* topics, where *N* is the number corresponding to the LOG Opcode used. For example, LOG0 means a log record with no topics, and LOG4 means a log record with four topics. *X* and *Y* represent the items removed and placed on the stack, respectively. *X* and *Y* change incrementally, starting from 2, 0 up to 6, 0 according to the LOG operation used.

System operations

System operations are used to perform various system-related operations, such as account creation, message calling, and execution control. There are six Opcodes available in total in this category.

Mnemonic	Value	POP	PUSH	Gas	Description

CREATE	0xf0	3	1	32000	Used to create a new account with the associated code.
CALL	0xf1	7	1	40	Used to initiate a message call into an account.
CALLCODE	0xf2	7	1	40	Used to initiate a message call into this account with an alternative account's code.
RETURN	0xf3	2	0	0	Stops the execution and returns output data.
DELEGATECALL	0xf4	6	1	40	The same as CALLCODE but does not change the current values of the sender and the value.
SUICIDE	0xff	1	0	0	Stops (halts) the execution and the account is registered for deletion later

In this section, all EVM opcodes have been discussed. There are 129 opcodes available in the EVM of the homestead release of Ethereum in total.

Precompiled contracts

There are four precompiled contracts in Ethereum. Here is the list of these contracts and details.

The elliptic curve public key recovery function

ECDSARECOVER (**Elliptic curve DSA recover function**) is available at address 1. It is denoted as ECREC and requires 3000 gas for execution. If the signature is invalid, then no output is returned by this function. Public key recovery is a standard mechanism by which the public key can be derived from the private key in elliptic curve cryptography.

The ECDSA recovery function is shown as follows:

ECDSARECOVER(H, V, R, S) = Public Key

It takes four inputs: H, which is a 32 byte hash of the message to be signed and *V*, *R*, and *S*, which represent the ECDSA signature with the recovery ID and produce a 64 byte public key. *V*, *R*, and *S* have been discussed in detail previously in this chapter.

The SHA-256 bit hash function

The SHA-256 bit hash function is a precompiled contract that is available at address 2 and produces a SHA256 hash of the input. It is almost like a pass-through function. Gas requirement for SHA-256 (SHA256) depends on the input data size. The output is a 32 byte value.

The RIPEMD-160 bit hash function

The RIPEMD-160 bit hash function is used to provide RIPEMD 160-bit hash and is available at address 3. The output of this function is a 20-byte value. Gas requirement, similar to SHA-256, is dependent on the amount of input data.

The identity function

The identity function is available at address 4 and is denoted by the ID. It simply defines output as input; in other words, whatever input is given to the ID function, it will output the same value. Gas requirement is calculated by a simple formula: $15 + 3 [I_d/32]$ where I_d is the input data. This means that at a high level, the gas requirement is dependent on the size of the input data albeit with some calculation performed, as shown in the preceding equation.

All the previously mentioned precompiled contracts can become native extensions and can be included in the EVM opcodes in the future.

Accounts

Accounts are one of the main building blocks of the Ethereum blockchain. The state is created or updated as a result of the interaction between accounts. Operations performed between and on the accounts represent state transitions. State transition is achieved using what's called the Ethereum state transition function, which works as follows:

1. Confirm the transaction validity by checking the syntax, signature validity, and nonce.
2. Transaction fee is calculated and the sending address is resolved using the signature. Furthermore, sender's account balance is checked and subtracted accordingly and nonce is incremented. An error is returned if the account balance is not enough.
3. Provide enough ether (gas price) to cover the cost of the transaction. This is charged per byte incrementally according to the size of the transaction.
4. In this step, the actual transfer of value occurs. The flow is from the sender's account to receiver's account. The account is created automatically if the destination account specified in the transaction does not exist yet. Moreover, if the destination account is a contract, then the contract code is executed. This also depends on the amount of gas available. If enough gas is available, then the contract code will be executed fully; otherwise, it will run up to the point where it runs out of gas.
5. In cases of transaction failure due to insufficient account balance or gas, all state changes are rolled back with the exception of fee payment, which is paid to the miners.
6. Finally, the remainder (if any) of the fee is sent back to the sender as change and fee is paid to the miners accordingly. At this point, the function returns the resulting state.

Types of accounts

There are two types of accounts in Ethereum: Externally owned accounts
Contract accounts

The first is **externally owned accounts** (**EOAs**) and the other is contract accounts. EOAs are similar to accounts that are controlled by a private key in bitcoin. Contract accounts are the accounts that have code associated with them along with the private key. An EOA has ether balance, is able to send transactions, and has no associated code, whereas a **Contract Account** (**CA**) has ether balance, associated code, and the ability to get triggered and execute code in response to a transaction or a message. It is worth noting that due to the Turing-completeness property of the Ethereum blockchain, the code within contract accounts can be of any level of complexity. The code is executed by EVM by each mining node on the Ethereum network. In addition, contract accounts are able to maintain their own permanent state and can call other contracts. It is envisaged that in the serenity release, the distinction between externally owned accounts and contract accounts may be eliminated.

Block

As discussed earlier, blocks are the main building blocks of a blockchain. Ethereum blocks consist of various components, which are described as follows:

- The block header
- The transactions list
- The list of headers of Ommers or Uncles

The transaction list is simply a list of all transactions included in the block. In addition, the list of headers of Uncles is also included in the block. The most important and complex part is the block header, which is discussed here.

Block header

Block headers are the most critical and detailed components of an Ethereum block. The header contains valuable information, which is described in detail here.

Parent hash

This is the Keccak 256-bit hash of the parent (previous) block's header.

Ommers hash

This is the Keccak 256-bit hash of the list of Ommers (Uncles) blocks included in the block.

Beneficiary

Beneficiary field contains the 160-bit address of the recipient that will receive the mining reward once

the block is successfully mined.

State root

The state root field contains the Keccak 256-bit hash of the root node of the state trie. It is calculated after all transactions have been processed and finalized.

Transactions root

The transaction root is the Keccak 256-bit hash of the root node of the transaction trie. Transaction trie represents the list of transactions included in the block.

Receipts root

The receipts root is the keccak 256 bit hash of the root node of the transaction receipt trie. This trie is composed of receipts of all transactions included in the block. Transaction receipts are generated after each transaction is processed and contain useful post-transaction information. More details on transaction receipts are provided in the next section.

Logs bloom

The logs bloom is a bloom filter that is composed of the logger address and log topics from the log entry of each transaction receipt of the included transaction list in the block. Logging is explained in detail in the next section.

Difficulty: The difficulty level of the current block.

Number: The total number of all previous blocks; the genesis block is block zero.

Gas limit : The field contains the value that represents the limit set on the gas consumption per block.

Gas used : The field contains the total gas consumed by the transactions included in the block.

Timestamp: Timestamp is the epoch Unix time of the time of block initialization.

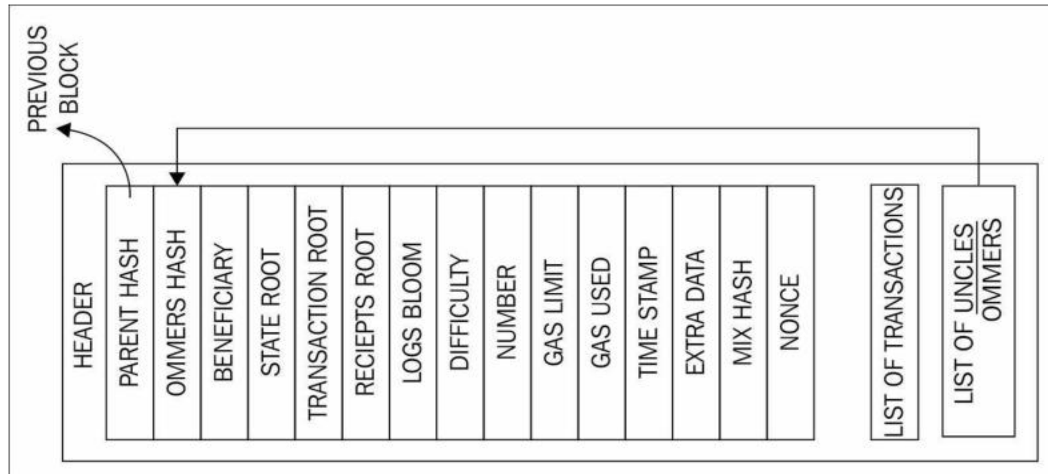
Extra data: Extra data field can be used to store arbitrary data related to the block.

Mixhash

Mixhash field contains a 256-bit hash that once combined with the nonce is used to prove that adequate computational effort has been spent in order to create this block.

Nonce

Nonce is a 64-bit hash (a number) that is used to prove, in combination with the mixhash field, that adequate computational effort has been spent in order to create this block.



The figure shows the detailed structure of the block and block header:

Detailed diagram of block structure with block header

The genesis block

The genesis block varies slightly with regard to the data it contains and the way it has been created from a normal block. It contains 15 items that are described here.

From Etherscan.io, the actual version is shown as follows:

Element	**Description**
Timestamp	(Jul-30-2015 03:26:13 PM +UTC)
Transactions	8893 transactions and 0 contract internal transactions in this block
Hash	0xd4e56740f876aef8c010b86a40d5f56745a118d0906a34e69aec8c0db1cb8fa3
Parent hash	0x0000000000000000000000000000000000000000000000000000000000000000
Sha3Uncles	0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347
Mined by	0x00000000000000000000000000000000000000 IN 15 secs
Difficulty	17,179,869,184
Total Difficulty	17,179,869,184

<table>
<tr><td>Size</td><td>540 bytes</td></tr>
<tr><td>Gas Limit</td><td>5,000</td></tr>
<tr><td>Gas Used</td><td>0</td></tr>
<tr><td>Nonce</td><td>0x0000000000000042</td></tr>
<tr><td>Block Reward</td><td>5 Ether</td></tr>
<tr><td>Uncles Reward</td><td>0</td></tr>
<tr><td>Extra Data</td><td>»èÛN4{NŒ"|ƒpäµí3³ÛiËÛz8áå ,ú
(Hex:0x11bbe8db4e347b4e8c937c1c8370e4b5ed33adb3db69cbdb7a38e1e50b1b82fa)</td></tr>
</table>

Transaction receipts

Transaction receipts are used as a mechanism to store the state after a transaction has been executed. In other words, these structures are used to record the outcome of the transaction execution. It is produced after the execution of each transaction. All receipts are stored in an index-keyed trie. Hash (Keccak 256-bit) of the root of this trie is placed in the block header as the receipts root. It is composed of four elements that are described here.

The post-transaction state

This item is a trie structure that holds the state after the transaction has executed. It is encoded as a byte array.

Gas used

This item represents the total amount of gas used in the block that contains the transaction receipt. The value is taken immediately after the transaction execution is completed. The total gas used is expected to be a non-negative integer.

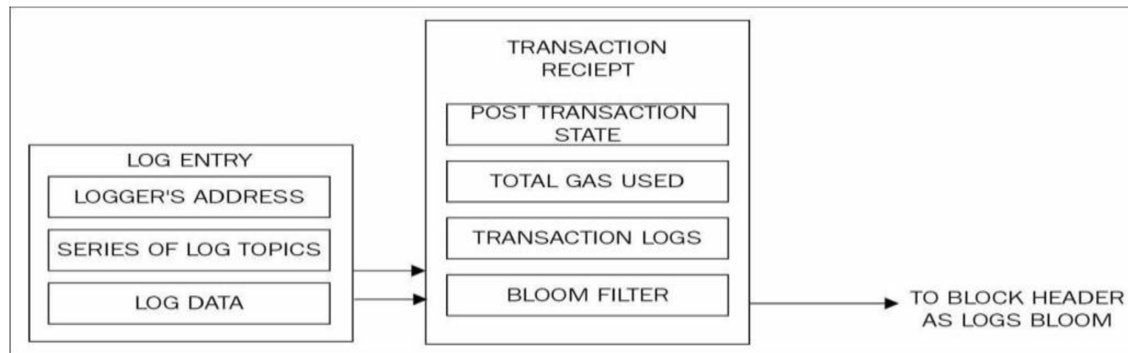
Set of logs

This field shows the set of log entries created as a result of transaction execution. Log entries contain the logger's address, a series of log topics, and the log data.

The bloom filter : A bloom filter is created from the information contained in the set of logs discussed earlier. A log entry is reduced to a hash of 256 bytes, which is then embedded in the header of the block as

the logs bloom. Log entry is composed of the logger's address and log topics and log data. Log topics are encoded as a series of 32 byte data structures. Log data is made up of a few bytes of data.

This process can be visualized in the following diagram:



Transaction receipts and logs bloom

Transaction validation and execution

Transactions are executed after verifying the transactions for validity. Initial tests are listed as follows:

- A transaction must be well-formed and RLP-encoded without any additional trailing bytes
- The digital signature used to sign the transaction is valid
- Transaction nonce must be equal to the sender's account's current nonce
- Gas limit must not be less than the gas used by the transaction
- The sender's account contains enough balance to cover the execution cost

The transaction sub state

A transaction sub-state is created during the execution of the transaction that is processed immediately after the execution completes. This transaction sub-state is a tuple that is composed of three items.

Suicide set : This element contains the list of accounts that are disposed of after the transaction is executed.

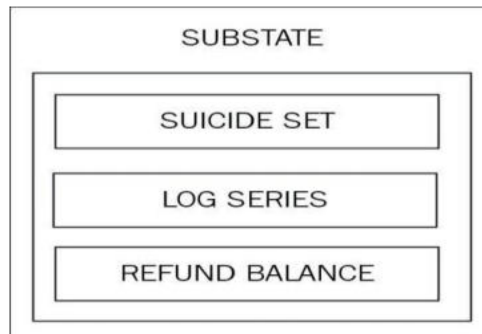
Log series

This is an indexed series of checkpoints that allow the monitoring and notification of contract calls to the entities external to the Ethereum environment, such as application frontends. It works like a trigger mechanism that is executed every time a specific function is invoked or a specific event occurs. Logs are created in response to events occurring in the smart contract. It can also be used as a cheaper form of storage.

Refund balance

This is the total price of gas in the transaction that initiated the execution. Refunds are not immediately executed; instead, they are used to partially offset the total execution cost.

The following diagram describes the transaction sub-state tuple:



Sub-state tuple

The block validation mechanism

An Ethereum block is considered valid if it passes the following checks:

- Consistent with Uncles and transactions. This means that all Ommers (Uncles) satisfy the property that they are indeed Uncles and also if the Proof of Work for Uncles is valid.
- If the previous block (parent) exists and is valid.
- If the timestamp of the block is valid. This basically means that the current block's timestamp must be higher than the parent block's timestamp. Also, it should be less than 15 minutes into the future. All block times are calculated in epoch time (Unix time).

If any of these checks fails, the block will be rejected.

Block finalization

Block finalization is a process that is run by miners in order to validate the contents of the block and apply rewards. It results in four steps being executed. These steps are described here in detail.

Ommers validation

Validate Ommers (stale blocks also called Uncles). In the case of mining, determine Ommers. The validation process of the headers of stale blocks checks whether the header is valid and the relationship of the Uncle with the current block satisfies the maximum depth of six blocks. A block can contain a maximum of two Uncles.

Transaction validation

Validate transactions. In the case of mining, determine transactions. The process involves checking whether the total gas used in the block is equal to the final gas consumption after the final transaction.

Reward application

Apply rewards, which means updating the beneficiary's account with a reward balance. In Ethereum, a

reward is also given to miners for stale blocks, which is 1/32 of the block reward. Uncles that are included in the blocks also receive 7/8 of the total block reward. The current block reward is 5 Ether. A block can have a maximum of two Uncles.

State and nonce validation

Verify the state and nonce. In the case of mining, compute a valid state and nonce.

Block difficulty

Block difficulty is increased if the time between two blocks decreases, whereas it increases if the block time between two blocks decreases. This is required to maintain a roughly consistent block generation time. The difficulty adjustment algorithm in Ethereum's homestead release is shown as follows:

```
block_diff = parent_diff + parent_diff // 2048 *
max(1 - (block_timestamp - parent_timestamp) // 10, -99) + int(2**((block.number // 100000) -
2))
```

The preceding algorithm means that, if the time difference between the generation of the parent block and the current block is less than 10 seconds, the difficulty goes up. If the time difference is between 10 to 19 seconds, the difficulty level remains the same. Finally, if the time difference is 20 seconds or more, the difficultly level decreases. This decrease is proportional to the time difference.

In addition to timestamp-difference-based difficulty adjustment, there is also another part (shown in the last line of the preceding algorithm) that increases the difficulty exponentially after every 100,000 blocks. This is the so called *difficulty time bomb* or *Ice age* introduced in the Ethereum network, which will make it very hard to mine on the Ethereum blockchain at some point in the future. This will encourage users to switch to Proof of Stake as mining on the POW chain will eventually become prohibitively difficult. According to the latest update and estimates based on the algorithm, the block generation time will become significantly high during the second half of the year 2017 and in 2021, it will become so high that it will be virtually impossible to mine on the POW chain. This way, miners will have no choice but to switch to the Proof of Stake scheme proposed by Ethereum called Casper.

UNIT- V

Hyperledger is not a blockchain, but it is a project that was initiated by Linux foundation in December 2015 to advance blockchain technology. This project is a collaborative effort by its members to build an open source distributed ledger framework that can be used to develop and implement cross- industry blockchain applications and systems. The key focus is to build and run platforms that
support global business transactions. The project also focuses on improving the reliability and performance of blockchain systems.

Projects under Hyperledger undergo various stages of development, starting from **proposal** to **incubation** and graduating to an **active** state. Projects can also be **deprecated** or in **End of Life** state where they are no longer actively developed. In order for a project to be able to move into incubation stage, it must have a fully working code base along with an active community of developers.

Projects

Currently there are six projects under the **Hyperledger umbrella: Fabric, Iroha, Sawtooth lake, blockchain explorer, Fabric chaintool, and Fabric SDK Py.** Corda is the most recent addition that is expected to be added to the Hyperledger project.

A brief introduction of all these projects follows, after which we will provide more details around the design, architecture, and implementation of Fabric and Sawtooth lake.

Fabric

Fabric is a blockchain project that was proposed by IBM and **DAH** (**Digital Asset Holdings**). This is intended to provide a foundation for the development of blockchain solutions and is based on pluggable architecture where various components, such as consensus algorithm, can be plugged into the system as required.

Sawtooth lake

Sawtooth lake is a blockchain project proposed by Intel in April 2016 with some key innovations focusing on **decoupling** of ledgers from transactions, flexible usage across multiple business areas using *transaction families*, and **pluggable consensus**. Decoupling can be explained more precisely by saying that the *transactions* are decoupled from the *consensus layer* by making use of a new concept called *Transaction families*. Instead of transactions being individually coupled with the ledger,

transaction families are used, which allows for more flexibility, rich semantics and unrestricted design of business logic. Transactions follow the patterns and structures defined in the transaction families. Intel has also introduced a novel consensus algorithm abbreviated as PoET, proof of elapsed time, which makes use of **Intel Software Guard Extensions** (**Intel's SGX**) architecture's **trusted execution environment** (**TEE**) in order to provide a safe and random leader election process. It also supports permissioned and permission-less setups.

Iroha

Iroha was proposed by Hitachi, NTT Data, and Soramitsu, Colu in September 2016. Iroha is aiming to build a library of reusable components that users can choose to run on their Hyperledger-based distributed ledgers. Iroha's main goal is to complement other Hyperledger projects by providing reusable components written in C++ with an emphasis on mobile development. This project has also proposed a novel consensus algorithm called Sumeragi, which is a chain based Byzantine fault tolerant consensus algorithm. Various libraries have been proposed and are being worked on by Iroha, including but not limited to a digital signature library (ed25519), an SHA-3 hashing library, a transaction serialization library, a P2P library, an API server library, an iOS library, an Android library, and a JavaScript library.

Blockchain explorer

This project aims to build a blockchain explorer for Hyperledger that can be used to view and query the transactions, blocks, and associated data from the blockchain. It also provides network information and the ability to interact with chain code.

Currently there are two other projects that are in incubation: Fabric chaintool, and Fabric SDK Py. These projects are aimed at supporting Hyperledger Fabric.

Fabric chaintool

Hyperledger chaincode compiler is being developed to support Fabric chaincode development. The aim is to build a tool that reads in a high-level Google protocol buffer structure and produces a chaincode. Additionally, it packages the chaincode so that it can be deployed directly. It is envisaged that this tool will help developers in various stages of development, such as compiling, testing, packaging, and deployment.

Fabric SDK Py

The aim of this project is to build a python based SDK library that can be used to interact with the blockchain (Fabric).

Corda

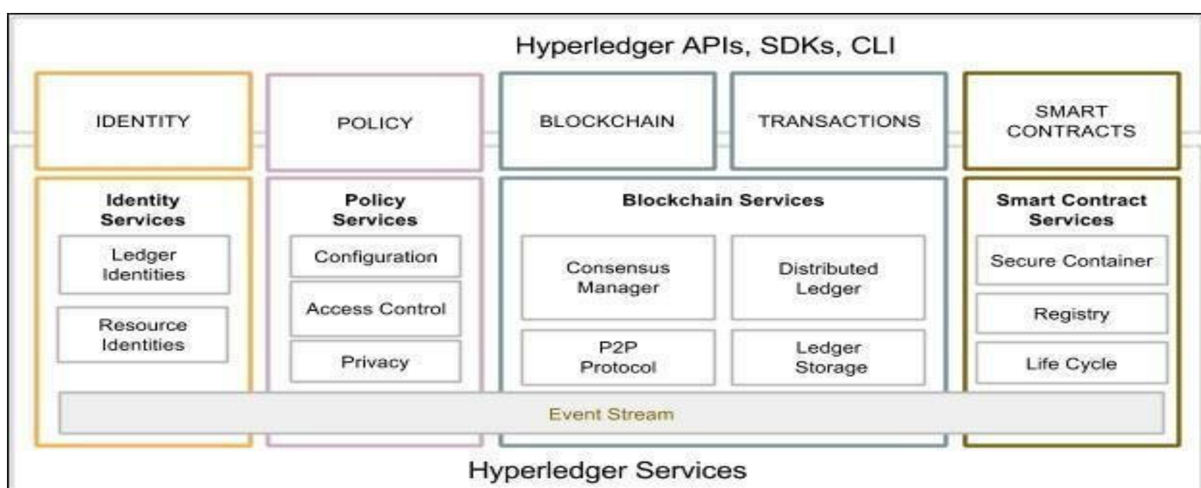
Corda is the latest project that has been contributed by R3 to the Hyperledger project. It was open sourced on November 30, 2016. Corda is heavily oriented towards the financial services industry and has been developed in collaboration with major banks and organizations in the financial industry. At the time of writing it is not yet in incubation under the Hyperledger project. Technically, Corda is not a blockchain but has key features similar to those of a blockchain, such as consensus, validity, uniqueness, immutability, and authentication.

Hyperledgeras a protocol

Hyperledger is aiming to build a new blockchain platform that is driven by industry use cases. As there have been number of contributions made to the Hyperledger project by the community, Hyperledger blockchain platform is evolving into a protocol for business transactions. Hyperledger is also evolving into a specification that can be used as a reference to build blockchain platforms as compared to earlier blockchain solutions that address only a specific type of industry or requirement.

Reference architecture

Hyperledger has published a white paper with reference architecture that can serve as a guideline to build permissioned distributed ledgers. The reference architecture consists of **two main components: Hyperledger services and Hyperledger APIs, SDKs, and CLI.** Hyperledger services provide various services such as identity services, policy services, blockchain services, and smart contract services. On the other hand, Hyperledger APIs, SDKs, and CLIs provide an interface into blockchain services via appropriate application programming interfaces, software development kits, or command line interfaces. Moreover, an event stream, which is basically a gRPC channel, runs across all services. It can receive and send events. Events are either pre-defined or custom. Validating peers or chaincode can emit events



to which external application can respond or listen to.

Hyperledger architecture, as proposed in the latest draft V2.0.0 of Hyperledger white paper.

Requirements

There are certain requirements of a blockchain service. The reference architecture is driven by the needs and requirements raised by the participants of the Hyperledger project and after studying the industry use cases. There are several categories of requirements that have been deduced from the study of industrial use cases and are discussed in the following sections.

Modular approach

The main requirement of Hyperledger is a modular structure. It is expected that, as a cross-industry fabric (blockchain), it will be used in many business scenarios. As such, functions related to storage, policy, chaincode, access control, consensus and many other blockchain services should be pluggable. The modules should be plug and play and users should be able to easily remove and add a different module that meets the requirements of the business.

For example, if a business blockchain needs to be run only between already trusted parties and performs very basic business operations, then perhaps there is no need to have advanced cryptographic support for confidentiality and privacy, and therefore users should be able to remove that functionality (module) or replace that with a more appropriate module that suits their needs.
Similarly, if users need to run a cross-industry blockchain, then confidentiality and privacy can be of paramount importance. In this case, users should be able to plug an advanced cryptographic and access control mechanism (module) into the blockchain (fabric).

Privacy and confidentiality

Privacy and confidentiality of transactions and contracts is of utmost importance in a business blockchain. As such, Hyperledger's vision is to provide a wide range of cryptographic protocols and algorithms and it is expected that users will be able to choose appropriate modules according to their business requirements. The fabric should be able to handle complex cryptographic algorithms without compromising performance.

Identity: In order to provide privacy and confidentiality services, a flexible PKI model that can be used to handle the access control functionality is also required. The strength and type of cryptographic mechanisms is also expected to vary according to the needs and requirements of the users. In certain scenarios it might be required for a user to hide their identity, and as such the Hyperledger is expected to provide this functionality.

Auditability : Auditability is another requirement of a Hyperledger Fabric. It is expected that an immutable audit trail of all identities, related operations and any changes is kept.

Interoperability

Currently there are many blockchain solutions available, but they cannot communicate with each other and this can be a limiting factor in the growth of a blockchain based global business ecosystem. It is envisaged that many blockchain networks will operate in the business world for specific needs, but it is important that they are able to communicate with each other. There should be a common set of standards that all blockchains can follow in order to allow communication between different ledgers. It is expected that a protocol will be developed that will allow the exchange of information between many Fabrics.

Portability

The portability requirement is concerned with the ability to run across multiple platforms and environments without the need to change anything at code level. Hyperledger is envisaged to be portable, not only at infrastructure level but also at code, libraries, and API levels so that it can support uniform development across various implementations of Hyperledger.

Fabric

In order to understand various projects under incubation in Hyperledger project, it is important to understand the foundations of Hyperledger first. A few terminologies that are specific to Hyperledger needs some clarification before readers are introduced to more in-depth material. First there is the concept of Fabric.

Fabric can be defined as a collection of components providing a foundation layer that can be used to deliver a blockchain network. There are various types and capabilities of a fabric network, but all fabrics share common attributes such as immutability and are consensus driven. Some fabrics can provide modular approach towards building blockchain networks. In this case the blockchain network can have multiple pluggable modules to perform various function on the network. For example, consensus algorithms can be a pluggable module in a blockchain network where, depending on the requirements of the network, an appropriate consensus algorithm can be chosen and *plugged* into the network. The modules can be based on some particular specification of the fabric and can include APIs, access control, and various other components. Fabrics can also be designed either to be private or public and can allow the creation of multiple business networks. As an example, bitcoin is an application that runs on top of its fabric (blockchain network). As discussed earlier, blockchain can either be permissioned or permission-less and the same is truefor fabric in Hyperledger terminology.

Fabric is also the name given to the code contribution made by IBM to the Hyperledger foundation and is formally called Hyperledger Fabric. IBM also offers blockchain as a service (IBM Blockchain) via its Bluemix cloud service.

Hyperledger Fabric

Fabric is the contribution originally made by IBM to the Hyperledger project. The aim of this contribution is to enable a modular, open and flexible approach towards building blockchain networks. Various functions in the fabric are pluggable, and it also allows use of any language to develop smart contracts. This is possible because it is based on container technology which can host any language. Chaincode (smart contract) is sandboxed into a secure container which includes a secure operating system, chaincode language, runtime environment and SDKs for Go, Java, and Node.js. Other languages can be supported too if required. Smart contracts are called chaincode in the Fabric. This is a very powerful feature compared to domain specific languages in Ethereum, or the very limited scripted language in bitcoin. It is a permissioned network that aims to address issues such as scalability, privacy, and confidentiality. The key idea behind this is modular technology, which would allow for flexibility in design and implementation. This can then result in achieving scalability, privacy and other desired attributes. Transactions in fabric are private, confidential and anonymous for general users, but they can still be traced and linked to the users by authorized auditors. As a permissioned network, all participants are required to be registered with the membership services in order to access the blockchain network. This ledger also provided auditability functionality in order to meet the regulatory and compliance needs.

Fabric architecture

The Fabric is logically organized into three main categories based on the type of service provided. These include membership services, blockchain services, and chaincode services. In the following section, all these categories and associated components are discussed in detail. The current stable version of Hyperledger Fabric is v0.6, however the latest version v1.0 is available but is not yet stable. In version 1.0, many architectural changes have been made, and in later sections of this chapter some changes that have been made in version 1.0 will also be discussed.

Membership services

These services are used to provide access control capability for the users of the fabric network. The following list shows the functions that membership services perform:

1. User identity validation.
2. User registration.
3. Assign appropriate permissions to the users depending on their roles.

Membership services makes use of **Public Key Infrastructure** (**PKI**) in order to support identity management and authorization operations. Membership services are made up of various components:

- **Registration authority** (**RA**): A service that authenticates the users and assesses the identity of the

fabric participants for issuance of certificates.

- **Enrolment certificate authority**: **Enrolment certificates** (**Ecerts**) are long term certificates issued by ECA to registered participants in order to provide identification to the entities participating on the network.
- **Transaction certificate authority**:In order to send transactions on the networks, participants are required to hold a transaction certificate. TCA is responsible for issuing transaction certificates to holders of Enrolment certificates and is derived from Ecerts.
- **TLS certificate authority**:In order to secure the network level communication between nodes on the Fabric, TLS certificates are used. TLS certificate authority issues TLS certificates in order to ensure security of the messages being passed between various systems on the blockchain network.

Blockchain services

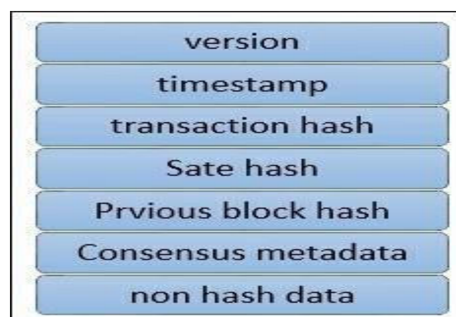
Blockchain services are at the core of the Hyperledger Fabric. Components within this category are as follows.

Consensus manager

Consensus manager is responsible for providing the interface to the consensus algorithm. This serves as an adapter that receives the transaction from other Hyperledger entities and executes them under criteria according to the type of algorithm chosen. Consensus is pluggable and currently there are three types of consensus algorithm available in Fabric, namely the batch PBFT protocol, SIEVE algorithm, and NOOPS.

Distributed ledger

Blockchain and world state are two main elements of the distributed ledger. Blockchain is simply a linked list of blocks (as introduced in earlier chapters) and world ledger is a key-value database.
This database is used by smart contracts to store relevant states during execution by the transactions. The blockchain consists of blocks that contain transactions. These transactions contain chaincode, which runs transactions that can result in updating the world state. Each node saves the world state on disk in RocksDB. The following diagram shows a typical block in the Hyperledger Fabric with the relevant fields. **Block structure**



The fields shown in the preceding diagram are as follows:

- **Version:** Used for keeping track of changes in the protocol.
- **Timestamp:** Timestamp in UTC epoch time, updated by block proposer.
- **Transaction hash:** This field contains the Merkle root hash of the transactions in the block.
- **State hash:** This is the Merkle root hash of the world state.
- **Previous hash:** This is the previous block's hash, which is calculated after serializing the block message and then creating the message digest by applying the SHA3 SHAKE256 algorithm.
- **Consensus metadata:** This is an optional field that can be used by the consensus protocol to provide some relevant information about the consensus.
- **Non-Hash data:** This is some metadata that is stored with the block but is not hashed. This feature makes it possible to have different data on different peers. It also provides the ability to discard data without any impact on the blockchain.

Peer to Peer protocol

P2P protocol in the Hyperledger Fabric is built using **google RPC** (**gRPC**). It uses protocol buffers to define the structure of the messages.

Messages are passed between nodes in order to perform various functions. There are four main types of messages in Hyperledger Fabric: Discovery, transaction, synchronization and consensus.
Discovery messages are exchanged between nodes when starting up in order to discover other peers on the network.

Transaction messages can be divided into two types: Deployment transactions and Invocation transactions. The former is used to deploy new chaincode to the ledger, and the latter is used to call functions from the smart contract. Transactions can be public, confidential, and confidential chaincode transactions. Public transactions are open and available to all participants. Confidential transactions are allowed to be queried only by transaction owners and participants. Confidential chaincode transactions have encrypted chaincode and can only be decrypted by validating nodes.
Validating nodes run consensus, validate the transactions and maintain the blockchain. Non-validating nodes on the other hand, provide transaction verification, stream server, and REST services. They also act as a proxy between the transactors and the validating nodes. Synchronization messages are used by peers to keep the blockchain updated and in synch with other nodes. Consensus messages are used in consensus management and broadcasting payloads to validating peers. These are generated internally by the consensus framework.

Ledger storage

In order to save the state of the ledger, RocksDB is used, and it is stored at each peer. RocksDB is a high performance database

Chaincode services

These services allow the creation of secure containers that are used to execute the chaincode. Components in this category are as follows:

- **Secure container:** Chaincode is deployed in Docker containers that provide a locked down sandboxed environment for smart contract execution. Currently Golang is supported as the main smart contract language, but any other main stream language can be added and enabled if required.
- **Secure registry:** This provides a record of all images containing smart contracts.

Events

Events on the blockchain can be triggered by validator nodes and smart contracts. External applications can listen to these events and react to them if required via event adapters. They are similar to the concept of events introduced in solidity in the last chapter.

APIs and CLIs

An application programming interface provides an interface into the fabric by exposing various REST APIs. Additionally, command line interfaces that provide a subset of REST APIs and allow for quick testing and limited interaction with the blockchain are also available.

Components of the Fabric

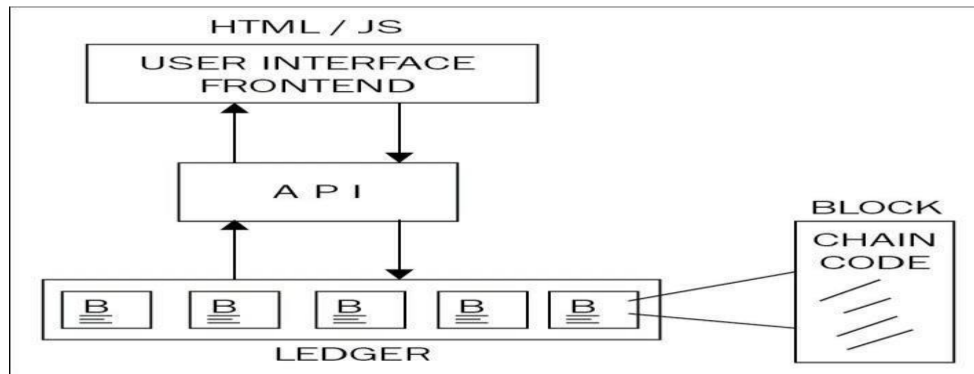
There are various components that can be part of the blockchain. These components include but are not limited to the ledger, chaincode, consensus mechanism, access control, events, system monitoring and management, wallets and system integration components.

Peers or nodes : There are two main types of peers that can be run on a fabric network: Validating and non-validating. Simply put, a validating node runs consensus, creates and validates a transaction, and contributes towards updating the ledger and maintaining the chaincode.

A non-validating peer does not execute transactions and only constructs transactions that are then forwarded to validating nodes.

Applications on blockchain

A typical application on Fabric is simply composed of a user interface, usually written in JavaScript/HTML, that interacts with the backend chaincode (smart contract) stored on the ledger via an API layer.



Typical blockchain application

Hyperledger provides various APIs and command line interfaces to enable interaction with the ledger. These APIs include interfaces for identity, transactions, chaincode, ledger, network, storage, and events.

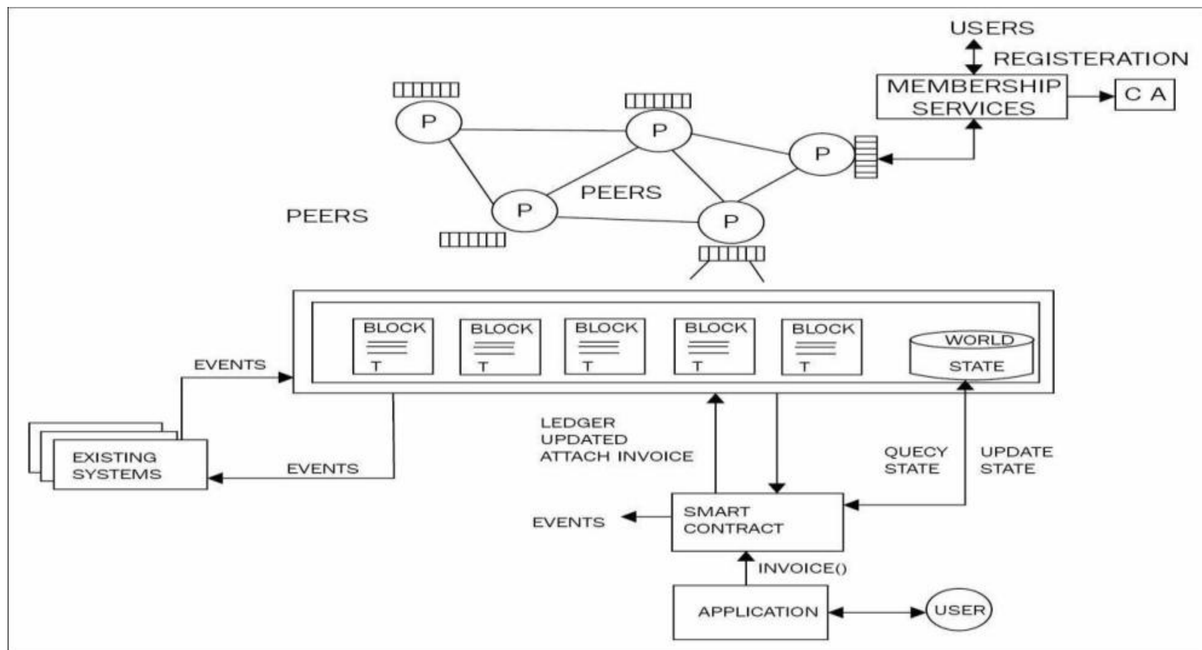
Chaincode implementation

Chaincode is usually written in Golang or Java. Chaincode can be public, confidential or access controlled. These codes serve as a smart contract that users can interact with via APIs. Users can call functions in the chaincode that result in a state change, and consequently updates the ledger. There are also functions that are only used to query the ledger and do not result in any state change.

Chaincode implementation is performed by first creating the chaincode shim interface in the code. It can either be in Java or Golang code. The following four functions are required in order to implement the chaincode:

- **Init():** This function is invoked when chaincode is deployed onto the ledger. This initializes the chaincode and results in making a state change, which accordingly updates the ledger.
- **Invoke():** This function is used when contracts are executed. It takes a function name as parameters along with an array of arguments. This function results in a state change and writes to the ledger.
- **Query():** This function is used to query the current state of a deployed chaincode. This function does not make any changes to the ledger.
- **Main():** This function is executed when a peer deploys its own copy of the chaincode. The chaincode is registered with the peer using this function.

The following diagram illustrates the general overview of Hyperledger Fabric:



High-level overview of Hyperledger Fabric

Application model

Any blockchain application for Hyperledger Fabric follows MVC-B architecture. This is based on the popular MVC design pattern. Components in this model are Model, View, Control, and Blockchain:

- **View logic**: This is concerned with the user interface. It can be a desktop, web application or mobile frontend.
- **Control logic**: This is the orchestrator between user interface, data model, and APIs.
- **Data model**: This model is used to manage the off-chain data.
- **Blockchain logic**: This is used to manage the blockchain via the controller and the data model via transactions.

Due to the fact that Hyperledger current release v0.6 is under heavy refactoring to build V1.0, no practical exercises have been introduced in this section.

Sawtooth lake

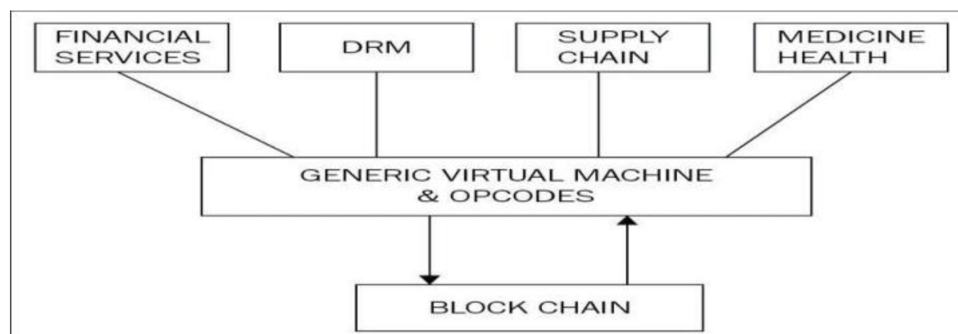
Sawtooth lake can run in both permissioned and non-permissioned modes. It is a distributed ledger that proposes two novel concepts: The first is the introduction of a new consensus algorithm called **Proof of Elapsed Time** (**PoET**); and the second is the idea of transaction families. A brief description of these novel proposals is given in the following section.

PoET

PoET is a novel consensus algorithm that allows a node to be selected randomly based on the time that the node has waited before proposing a block. This is in contrast to other leader election and lottery based proof of work algorithms, where an enormous amount of electricity and computer resources are used in order be elected as a block proposer, for example in the case of bitcoin. PoET is a type of Proof of Work algorithm but, instead of spending computer resources, it uses a trusted computing model to provide a mechanism to fulfill Proof of Work requirements. PoET makes use of Intel's SGX architecture to provide a trusted execution environment to ensure randomness and cryptographic security of the process. It should be noted that the current implementation of Sawtooth lake does not require real hardware SGX based TEE, as it is simulated for experimental purposes only and as such should not be used in production environments.

Transaction families

A traditional smart contract paradigm provides a solution that is based on a general purpose instruction set for all domains. For example, in the case of Ethereum, a set of opcodes has been developed for the **Ethereum virtual machine** (**EVM**) that can be used to build smart contracts to address any type of requirements for any industry. Whilst this model has its merits, it is becoming clear that this approach is not very secure as it provides a single interface into the ledger with a powerful and expressive language, which potentially offers a larger attack surface for malicious code. This complexity and generic virtual machine paradigm has resulted in several vulnerabilities that were found and exploited recently by hackers. A recent example is the DAO hack and further **Denial of Services** (**DoS**) attacks that exploited limitations in some EVM opcodes. A model shown in the following figure describes the traditional smart contract model, where a generic virtual machine has been used to provide the interface into the blockchain for all domains:

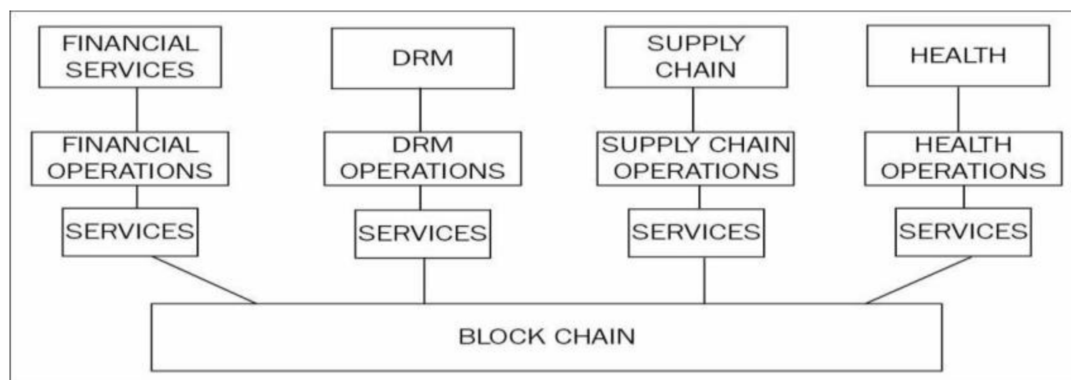


Traditional smart contract paradigm

In order to address this issue, Sawtooth lake has proposed the idea of transaction families. A

transaction family is created by decomposing the logic layer into a *set of rules* and a *composition layer* for a specific domain. The key idea is that business logic is composed within *transaction families*, which provides a more secure and powerful way to build smart contracts. Transaction families contain the domain-specific rules and another layer that allows for creating transactions for that domain. Another way of looking at it is that transaction families are a combination of a data model and a transaction language that implements a logic layer for a specific domain. The data model represents the current state of the blockchain (ledger) whereas the transaction language modifies the state of the ledger. It is expected that users will build their own transaction families according to their business requirements.

The following diagram represents this model, where each specific domain, like financial services, **digital rights management** (**DRM**), supply chain, and the health industry, has its own logic layer comprised of operations and services specific to that domain. This makes the logic layer both restrictive and powerful at the same time. Transaction families ensure that operations related to only the required domain are present in the control logic, thus removing the possibility of executing needless, arbitrary and potentially harmful operations.



Sawtooth (transaction families) smart contract paradigm

Intel has provided three transaction families with Sawtooth: Endpoint registry, Integerkey, and MarketPlace.

1. **Endpoint registry** is used for registering ledger services.
2. **Integerkey** is used for testing deployed ledgers.
3. **MarketPlace** is used for selling, buying and trading operations and services.

Sawtooth_bond has been developed as a proof of concept to demonstrate a bond trading platform.

Consensus in Sawtooth

Sawtooth has two types of consensus mechanisms based on the choice of network. PoET, as discussed previously, is a trusted executed environment based lottery function that elects a leader randomly based on the time a node has waited for block proposal. There is another consensus type called quorum voting, which is an adaptation of consensus protocols built by Ripple and Stellar. This consensus algorithm allows instant transaction finality, which is usually desirable in permissioned networks.

Corda

Corda is not a blockchain. Traditional blockchain solutions, as discussed before, have the concept of transactions that are bundled together in a block and each block is linked back cryptographically to its parent block, which provides an immutable record of transactions. This is not the case with Corda: Corda has been designed entirely from scratch with a new model for providing all blockchain benefits, but without a traditional blockchain. It has been developed purely for the financial industry to solve issues arising from the fact that each organization manages their own ledgers and thus have their own view of *truth*, which leads to contradictions and operational risk. Moreover, data is also duplicated at each organization which results in an increased cost of managing individual infrastructures and complexity. These are the types of problems within the financial industry that Corda aims to resolve by building a decentralized database platform.

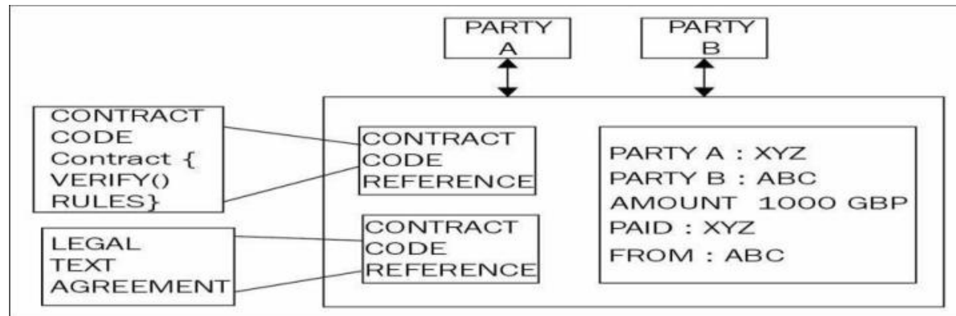
Corda it is written in a language called Kotlin, which is a statically typed language targeting the **Java Virtual Machine** (**JVM**).

Architecture

The main components of the Corda platform include state objects, contract code, legal prose, transactions, consensus, and flows.

State objects

State objects represent the smallest unit of data that represent a financial agreement. They are created or deleted as a result of a transaction execution. They refer to **contract code** and **legal prose.** Legal prose is optional and provides legal binding to the contract. However, contract code is mandatory in order to manage the state of the object. It is required in order to provide a state transition mechanism for the node according to the business logic defined in the contract code. State objects contain a data structure that represent the current state of the object. For example, in the following diagram, a state object represents the current state of the object. In this case, it is a simple mock agreement between **Party A** and **Party B** where **Party ABC** has paid **Party XYZ 1,000 GBP**. This represents the current state of the object; however the referred contract code can change the state via transactions. State objects can be thought of as a state machine, which are consumed by transactions in order to create updated state objects.



An example state object

Transactions

Transactions are used to perform transitions between different states. For example, the state object shown in the preceding diagram is created as a result of a transaction. Corda uses a bitcoin-style UTXO based model for its transaction processing. The concept of state transition by transactions is same as in bitcoin. Similar to bitcoin, transactions can have none, single or multiple inputs, and single or multiple outputs. All transactions are digitally signed. Moreover, Corda has no concept of mining because it does not use blocks to arrange transactions in a blockchain. Instead, notary services are used in order to provide temporal ordering of transactions. In Corda, new transaction types can be developed using JVM bytecode, which makes it very flexible and powerful.

Consensus

The consensus model in Corda is quite simple and is based on notary services that are discussed in a later section. The general idea is that the transactions are evaluated for their uniqueness by the notary service and, if they are unique, they are signed as valid. There can be single or multiple clustered notary services running on a Corda network. Various consensus algorithms like PBFT or Raft can be used by notaries to reach consensus.

There are two main concepts regarding consensus in Corda: Consensus over state validity, and consensus over state uniqueness. The first concept is concerned with the validation of the transaction, ensuring that all required signatures are available and states are appropriate. The second concept is a means to detect double--spend attack and ensures that a transaction has not been already been spent and is unique.

Flows

Flows in Corda are a novel idea that allow the development of decentralized workflows. All communication on the Corda network is handled by these flows. These are transaction-building protocols that can be used to define any financial flow of any complexity using code. Flows run as an asynchronous state machine and they interact with other nodes and users. During the execution, they can be suspended or resumed as required.

Components

The Corda network has multiple components. All these components are described in the next section.

Nodes

Nodes in a Corda network operated under a trust-less model and run by different organizations. Nodes run as part of an authenticated peer-to-peer network. Nodes communicate directly with each other using the **Advanced Message Queuing Protocol** (**AMQP**), which is an approved international standard (ISO/IEC 19464) and ensures that messages across different nodes are transferred safely and securely. AMQP works over **Transport Layer Security** (**TLS**) in Corda, thus ensuring privacy and integrity of data communicated between nodes.

Nodes also make use of a local relational database for storage. Messages on the network are encoded in a compact binary format. They are delivered and managed by using the **Apache Artemis message broker** (**Active MQ**). A node can serve as a network map service, notary, Oracle, or a regular node. The following diagram shows a high-level view of two nodes communicating with each other:

Two nodes communicating in a Corda network

In the preceding diagram, **Node 1** is communicating with **Node 2** over a TLS communication channel using the AMQP protocol, and the nodes have a local relational database for storage.

Permissioning service

A Permissioning service is used to provision TLS certificates for security. In order to participate on the network, participants are required to have a signed identity issued by a root certificate authority. Identities are required to be unique on the network and the Permissioning service is used to sign these identities. The naming convention used to recognise participants is based on the X.500 standard. This ensures the uniqueness of the name.

Network map service

This service is used to provide a network map in the form of a document of all nodes on the network. This service publishes IP addresses, identity certificates and a list of services offered by nodes. All nodes announce their presence by registering to this service when they first start up, and when a connection request is received by a node, the presence of the requesting node is checked on the network map first.

Put another way, this service resolves the identities of the participants to physical nodes.

Notary service

In a traditional blockchain, mining is used to ascertain the order of blocks that contain transactions. In Corda, notary services are used to provide transaction ordering and timestamping services. There can be multiple notaries in a network and they are identified by composite public keys. Notaries can use different consensus algorithms like BFT or Raft depending on the requirements of the applications.
Notary services sign the transactions to indicate validity and finality of the transaction which is then persisted to the database.

Notaries can be run in a load-balanced configuration in order to spread the load across the nodes for performance reasons; and, in order to reduce latency, the nodes are recommended to be run physically closer to the transaction participants.

Oracle service

Oracle services either sign a transaction containing a fact, if it is true, or can themselves provide factual data. They allow real world feed into the distributed ledgers.

Transactions : Transactions in a Corda network are never transmitted globally, but in a semi-private network. They are shared only between a subset of participants who are related to the transaction. This is in contrast to traditional blockchain solutions like Ethereum and bitcoin, where all transactions are broadcasted to the entire network globally. Transactions are digitally signed and either consume state(s) or create new state(s).

Transactions on a Corda network are composed of the following elements:

- **Input references**: This is a reference to the states the transaction is going to consume and use as an input.
- **Output states**: These are new states created by the transaction.
- **Attachments**: This is a list of hashes of attached zip files. Zip files can contain code and other relevant documentation related to the transaction. Files themselves are not made part of the transaction, instead, they are transferred and stored separately.
- **Commands**: A command represents the information about the intended operation of the transaction as a parameter to the contract. Each command has a list of public keys which represents all parties that are required to sign a transaction.
- **Signatures**: This represents the signature required by the transaction. The total number of signatures required is directly proportional to the number of public keys for commands.
- **Type**: There are two types of transactions namely, Normal or Notary changing. Notary changing transactions are used for reassigning a notary for a state.
- **Timestamp**: This field represents a bracket of time during which the transaction has taken place.

These are verified and enforced by notary services. Also, it is expected that if strict timings are required, which is desirable in many financial services scenarios, notaries should be synched with an atomic clock.

- **Summaries**: This is a text description that describes the operations of the transaction.

Vaults : Vaults run on a node and are akin to the concept of wallets in bitcoin. As the transactions are not globally broadcast, each node will have only that part of data in their vaults that is considered relevant to them. Vaults store their data in a standard relational database and as such can be queried by using standard SQL. Vaults can contain both on ledger and off ledger data, meaning that it can also have some part of data that is not on ledger.

CorDapp: The core model of Corda consists of state objects, transactions and transaction protocols, which when combined with contract code, APIs, wallet plugins, and user interface components results in constructing a Corda distributed application (CorDapp).

Smart contracts in Corda are written using Kotlin or Java. The code is targeted for JVM. JVM has been modified slightly in order to achieve deterministic results of execution of JVM bytecode. There are three main components in a Corda smart contract as follows:

1. Executable code that defines the validation logic to validate changes to the state objects.
2. State objects represent the current state of a contract and either can be consumed by a transaction or produced (created) by a transaction.
3. Commands are used to describe the operational and verification data that defines how a transaction can be verified.