Module-3

Bitcoin, Bitcoin definition, Transactions, The transaction life cycle, The transaction structure, Types of transaction, The structure of a block , The structure of a block header, The genesis block, The bitcoin network, Wallets, Smart Contracts-History, Definition, Ricardian contracts, Smart contract templates, Oracles, Smart Oracles, Deploying smart contracts on a blockchain, The DAO.

Chapter 4:pg:111-148, Chapter 6

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 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.

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 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 should send coins to some other user (a Bitcoin address), then it needs to be signed by the sender with their private key. In this case, a reference is also required to the previous transaction to show the origin of the coins. Coins are unspent transaction outputs represented in Satoshis.

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

THE TRANSACTION LIFECYCLE OF A BITCOIN

The steps of the process are as follows:
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 (miners) who are listening for the transactions verify and include this transaction in the next block to be mined. Just before the transactions are placed in the block, they are placed in a special memory buffer called the transaction pool. The purpose of the transaction pool is explained in the next section.
5. Next, the mining starts, which is the process through which the blockchain is secured and new coins are generated as a reward for the miners who spend appropriate computational resources.
Once a miner solves the PoW problem, it broadcasts the newly mined block to the network. PoW is explained in detail in the Mining section. The nodes verify the block and propagate the block further, and confirmations start to generate.
6. Finally, the confirmations start to appear in the receiver's wallet and after approximately three confirmations, the transaction is considered finalized and confirmed.
However, three to 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.
When a transaction is created by a user and sent to the network, it ends up in a special area on each Bitcoin software client. This special area is called the **transaction pool or memory pool.**

TRANSACTION POOL

- Also known as memory pools, these pools are basically created in local memory (computer RAM) by nodes (Bitcoin clients) in order to maintain a temporary list of transactions that have not yet been added to a block.
- Miners pick up transactions from these memory pools to create candidate blocks. Miners select transactions from the pool after they pass the verification and validity checks.
- The selection of which transactions to choose is based on the fee and their place in the order of transactions in the pool. Miners prefer to pick up transactions with

higher fees. To send transactions on the Bitcoin network, the sender needs to pay a fee to the miners. This fee is an incentive mechanism for the miners.

Transaction fees

Transaction fees are charged by the miners. The fee charged is dependent upon the size and weight of the transaction. Transaction fees are calculated by subtracting the sum of the inputs from the sum of the outputs.

A **simple formula** can be used:

fee = sum(inputs) – sum(outputs)

The fees are used as an incentive for miners to encourage them to include users' transactions 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. 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. This is, however, no longer practical due to the high volume of transactions and competing investors on the Bitcoin network, therefore it is advisable to always provide a fee.
- The time for transaction confirmation usually ranges from 10 minutes to over 12 hours in some cases. Transaction time is dependent on transaction fees and network activity. If the network is very busy, then naturally, transactions will take longer to process, and if you pay a higher fee then your transaction is more likely to be picked by miners first due to the additional incentive of the higher fee.

THE TRANSACTION DATA STRUCTURE

A transaction at a high level contains metadata, inputs, and outputs. Transactions are combined to create a block's body.

The transaction data structure is shown in the following table:

Field	Size	Description
Version number	4 bytes	Specifies the rules to be used by the miners and nodes for transaction processing. There are two versions of transactions, that is, 1 and 2.
Input counter	1–9 bytes	The number (a positive integer) of inputs included in the transaction.
List of inputs	Variable	Each input is composed of several fields. These include: • The previous transaction hash • The index of the previous transaction • Transaction script length • Transaction script • Sequence number The first transaction in a block is also called a coinbase transaction. It specifies one or more transaction inputs. In summary, this field describes which Bitcoins are going to be spent.
Output counter	1–9 bytes	A positive integer representing the number of outputs.
List of outputs	Variable	Outputs included in the transaction. This field depicts the target recipient(s) of the Bitcoins.
Lock time	4 bytes	This field defines the earliest time when a transaction becomes valid. It is either a Unix timestamp or the block height.

- **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.

The transaction input data structure is explained in the following table:

Field	Size	Description
Transaction hash	32 bytes	The hash of the previous transaction with UTXO
Output index	4 bytes	This is the previous transaction's output index, such as UTXO to be spent
Script length	1-9 bytes	Size of the unlocking script
Unlocking script	Variable	Input script (`ScriptSig`), which satisfies the requirements of the locking script
Sequence number	4 bytes	Usually disabled or contains lock time — disabled is represented by `0xFFFFFFFF`

- **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.

The transaction output data structure is explained in the following table:

Field	Size	Description
Value	8 bytes	The total number (in positive integers) of Satoshis to be transferred
Script size	1 – 9 bytes	Size of the locking script
Locking script	Variable	Output script (`ScriptPubKey`)

- **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"
}}}
```


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 programming language-like syntax and uses a reverse polish notation in which every operand is followed by its operators.
- It is evaluated from left to right using a Last in, First Out (LIFO) stack. Scripts are composed of two components, namely elements and operations. Scripts use various operations (opcodes) or instructions to define their operations. Elements simply represent data such as digital signatures.
- Opcodes are also known as words, commands, or functions. Earlier versions of the Bitcoin node software 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 ScriptPubKey is the locking script. We will now describe how a transaction is unlocked and 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 (the locking script), which contains the conditions that when met, will unlock the output, and the coins can then be redeemed.

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_VERIFY afterwards.
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 follows:

ScriptPubKey: OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG

ScriptSig: <sig> <pubKey>

The ScriptPubKey and ScriptSig parameters are concatenated together and executed.

- **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,

- **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:

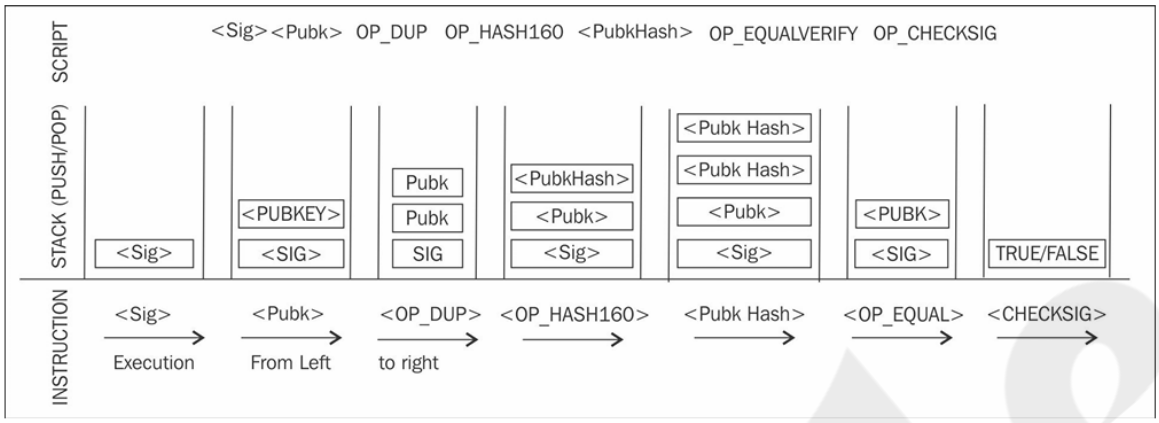
  <PubKey> OP_CHECKSIG

- **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_RETURN will fail the validation in any case. ScriptSig is 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

Now let's examine how this script is executed:

1. In the first step of the data elements, and are placed on the stack.
2. The stack item at the top is duplicated due to the OP_DUP instruction, which duplicates the top stack item.
3. After this, the instruction OP_HASH160 executes, which produces the hash of , which is the top element in the stack.
4. is then pushed on to the stack. At this stage, we have two hashes on the stack: the one that is produced as a result of executing OP_HASH160 on from the unlocking script, and the other one provided by the locking script.
5. Now the OP_EQUALVERIFY opcode instruction executes and checks that whether the top two elements (that is, the hashes) are equal or not. If they are equal, the script continues, otherwise it fails.
6. Finally, OP_CHECKSIG executes to check the validity of the signatures of the top two elements of the stack. If the signature is valid then the stack will contain the value True that is, 1, otherwise False , that is, 0.

All transactions are encoded into hex format before being transmitted over the Bitcoin network.

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 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:** 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 multiSig and 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. ScriptPubkey not 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

  Transaction validation This verification process is performed by Bitcoin nodes. There are three main things that nodes check when verifying a transaction:
  1. That transaction inputs are previously unspent. This validation step prevents double spending by verifying that the transaction inputs have not already been spent by someone else.
  2. That the sum of the transaction outputs is not more than the total sum of the transaction inputs. However, both input and output sums can be the same, or the sum of the input (total value) could be more than the total value of the outputs. This check ensures that no new bitcoins are created out of thin air.

3. That the digital signatures are valid, which ensures that the script is valid. Even though transaction construction and validation are generally a secure and sound process, some vulnerabilities exist in Bitcoin.

TRANSACTION BUGS The following are two major Bitcoin vulnerabilities that have been infamously exploited.

Transaction malleability:

- Transaction malleability is a Bitcoin attack that was introduced due to a bug in the Bitcoin implementation. Due to this bug, it became 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 without making the transaction invalid, it would seem that the transaction did not occur at all, which can then give the false impression that the transaction has not been executed, thus allowing double-deposit or withdrawal attacks.

Value overflow

- This incident is one of the most well-known events in Bitcoin history. On 15 August 2010, a transaction was discovered that created roughly 184 billion bitcoins. This problem occurred due to the integer overflow bug where the amount field in the Bitcoin code was defined as a signed integer instead of an unsigned integer.
- This bug means that the amount can also be negative, and resulted in a situation where the outputs were so large that the total value resulted in an overflow. To the validation logic in Bitcoin code, all appeared to be correct, and it looked like the fee was also positive (after the overflow). This bug was fixed via a soft fork quickly after its discovery.

BLOCKCHAIN

Block chain 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.

The data structure of a bitcoin 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.

Field	Size	Description
Version	4 bytes	The block version number that dictates the block validation rules to follow.
Previous block's header hash	32 bytes	This is a double SHA256 hash of the previous block's header.
Merkle root hash	32 bytes	This is a double SHA256 hash of the Merkle tree of all transactions included in the block.
Timestamp	4 bytes	This field contains the approximate creation time of the block in Unix-epoch time format. More precisely, this is the time when the miner started hashing the header (the time from the miner's location).
Difficulty target	4 bytes	This is the current difficulty target of the network/block.
Nonce	4 bytes	This is an arbitrary number that miners change repeatedly to produce a hash that is lower than the difficulty target.

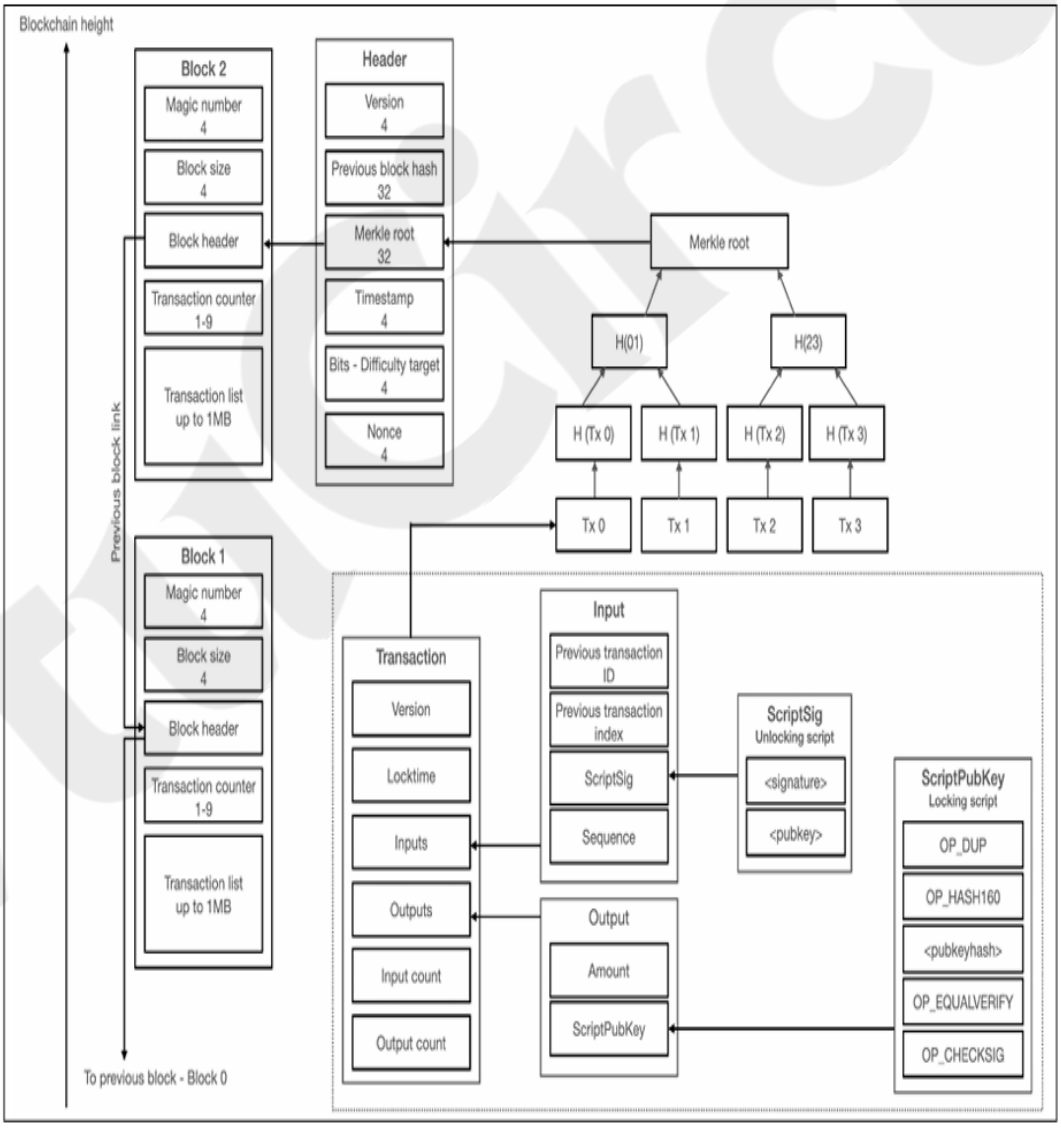


Figure 6.16: A visualization of blockchain, block, block header, transactions, and scripts

- The preceding diagram shows a high-level overview of the Bitcoin blockchain. On the left-hand side, blocks are shown starting from bottom to top. Each block contains transactions and block headers, which are further magnified on the right-hand side. At the top, first, the block header is enlarged to show various elements within the block header. Then on the right-hand side, the Merkle root element of the block header is shown in magnified view, which shows how Merkle root is constructed
- **The genesis block** This is the first block in the Bitcoin blockchain. The genesis block was hardcoded in the Bitcoin core software. In the genesis block, the coinbase transaction included a comment taken from The Times newspaper.
- This message is a proof that the first Bitcoin block (genesis block) was not mined earlier than January 3rd, 2009. This is because the genesis block was created on January 3rd, 2009 and this news excerpt was taken from that day's newspaper. The following representation of the genesis block code can be found in the chainparams.cpp file available at https://github.com/Bitcoin/Bitcoin/blob/master/src/chainparams.cpp:

```
static CBlock CreateGenesisBlock(uint32_t nTime, uint32_t nNonce, uint32_t nBits,
int32_t nVersion, const CAmount& genesisReward)
{
    const char* pszTimestamp = "The Times 03/Jan/2009 Chancellor on brink of
second bailout for banks";
    const CScript genesisOutputScript = CScript() << ParseHex("04678afdb0fe554827
1967f1a67130b7105cd6a828e03909a67962e0ea1f61deb649f6bc3f4cef38c4f35504e51ec112de5
c384df7ba0b8d578a4c702b6bf11d5f") << OP_CHECKSIG;
    return CreateGenesisBlock(pszTimestamp, genesisOutputScript, nTime, nNonce,
nBits, nVersion, genesisReward);
}
```

- Bitcoin provides protection against double-spending by enforcing strict rules on transaction verification and via mining. Transactions and blocks are added to the blockchain only after the strict rule-checking explained in the Transaction validation section on successful PoW solutions. Block height is the number of blocks before a particular block in the blockchain. PoW is used to secure the blockchain.
- Each block contains one or more transactions, out of which the first transaction is the coinbase transaction. There is a special condition for coinbase transactions that prevent them from being spent until at least 100 blocks have passed in order to avoid a situation where the block may be declared stale later on.

Stale and orphan blocks

- **Stale blocks are old blocks** that have already been mined. Miners who keep working on these blocks due to a fork, where the longest chain (main chain) has already progressed beyond those blocks, are said to be working on a stale block. In other words, these blocks exist on a shorter chain, and will not provide any reward to their miners.

- **Orphan blocks** are a slightly different concept. Their parent blocks are unknown. As their parents are unknown, they cannot be validated. This problem occurs when two or more miners discover a block at almost the same time. These are valid blocks and were correctly discovered at some point in the past but now they are no longer part of the main chain. The reason why this occurs is that if there are two blocks discovered at almost the same time, the one with a larger amount of PoW will be accepted and the one with the lower amount of work will be rejected. Similar to stale blocks, they do not provide any reward to their miners.

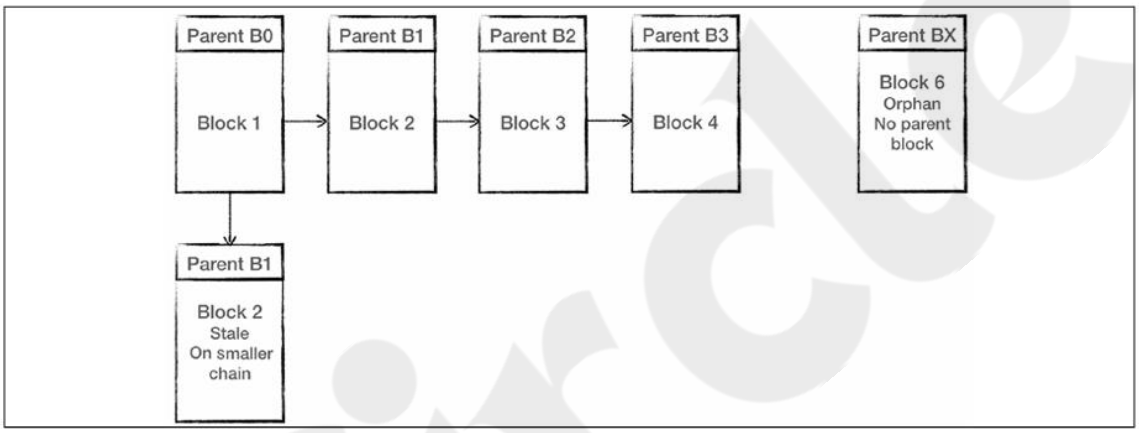


Figure 6.17: Orphan and stale blocks

- A fork is a condition that occurs when two different versions of the blockchain exist. It is acceptable in some conditions, and detrimental in a few others.
- There are different types of forks that can occur in a blockchain: • **Temporary forks • Soft forks • Hard forks.**
- Forks in a blockchain can also occur with the introduction of changes to the Bitcoin protocol. In the case of a **soft fork**, a client that chooses not to upgrade to the latest version supporting the updated protocol will still be able to work and operate normally. In this case, new and previous blocks are both acceptable, thus making a soft fork backward compatible. Miners are only required to upgrade to the new soft fork 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 changes or major protocol changes result in a hard fork.

SIZE OF THE BLOCKCHAIN

- Bitcoin is an ever-growing chain of blocks and is increasing in size. The current size of the Bitcoin blockchain stands at approximately 269 GB. The following figure shows the increase in size of the blockchain over time

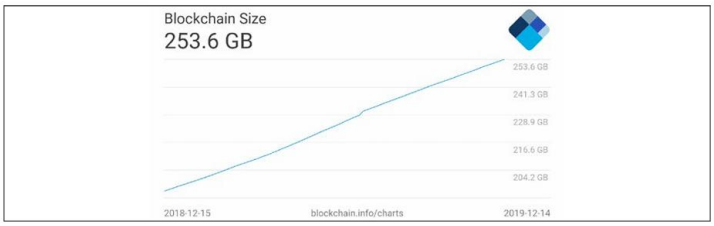


Figure 6.18: Size of Bitcoin blockchain over time

Network difficulty

- Network difficulty refers to a measure of how difficult it is to find a new block, or in other words, how difficult it is to find a hash below the given target. New blocks are added to the blockchain approximately every 10 minutes, and the network difficulty is adjusted dynamically every 2,016 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. The Previous target represents the old target value, and Time is the time spent to generate the previous 2,016 blocks. Network difficulty essentially 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 process by which new blocks are added to the blockchain. Blocks contain transactions that are validated via the mining process by mining nodes on the Bitcoin network.
- Blocks, once mined and verified, are added to the blockchain, which keeps the blockchain growing. This process is resource-intensive due to the requirements of PoW, where miners compete to find a number less than the difficulty target of the network.
- This difficulty in finding the correct value (also called sometimes the mathematical puzzle) is there to ensure that miners have spent the required resources before a new proposed block can be accepted. The miners mint new coins by solving the PoW problem, also known as the **partial hash inversion problem**.
- This process consumes a high amount of resources, including computing power and electricity. This process also secures the system against fraud and double-spending attacks while adding more virtual currency to the Bitcoin ecosystem.
- 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 an average of 144 per day. Bitcoin supply is also limited.
- In 2140, all 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.

Tasks of the miners

Once a node connects to the Bitcoin network, there are several tasks that a Bitcoin miner performs:

1. 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 that only concerns miners.

2. Transaction validation: Transactions broadcast on the network are validated by full nodes by verifying and validating signatures and outputs.

3. 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.

4. Create a new block: Miners propose a new block by combining transactions broadcast on the network after validating them.

5. Perform PoW: 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.

6. Fetch reward: Once a node solves the hash puzzle (PoW), 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 on the network due to a clash with another block found at roughly the same time, but once accepted, the miner is rewarded with 12.5 bitcoins and any associated transaction fees.

Mining rewards

- Miners are rewarded with new coins if and when they discover new blocks by solving the PoW. Miners are paid transaction fees in return, for the transactions in their proposed blocks.
- New blocks are created at an approximate fixed rate of every 10 minutes. The rate of creation of new bitcoins decreases by 50% every 210,000 blocks, which is roughly every 4 years.
- When Bitcoin started in 2009, the mining reward used to be 50 bitcoins. After every 210,000 blocks, the block reward halves. In November 2012 it halved down to 25 bitcoins.
- Currently, since May 2020, it is 6.25 bitcoins per block. This mechanism is hardcoded in Bitcoin to regulate and control inflation and limit the supply of bitcoins. In order for miners to earn the reward, they have to show that they have solved the computational puzzle. This is called the **PoW**

Proof of Work

- This is a proof that enough computational resources have been spent in order to build a valid block. 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 proportion to their computing capacity, in order to be selected.

The following equation sums up the PoW requirement in Bitcoin:

$$H(N \mid\mid P\_hash \mid\mid Tx \mid\mid Tx \mid\mid \ldots Tx) < Target$$

Here, 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 broadcast and accepted by other miners.

The mining algorithm

The mining algorithm consists of the following steps:

1. The previous block's header is retrieved from the Bitcoin network.
2. Assemble a set of transactions broadcast on the network into a block to be proposed.
3. Compute the double hash of the previous block's header, combined with a nonce and the newly proposed block, using the SHA256 algorithm.
4. Check if the resulting hash is lower than the current difficulty level (the target), then PoW is solved. As a result of successful PoW, the discovered block is broadcasted to the network and miners fetch the reward.
5. If the resultant hash is not less than the current difficulty level (target), then repeat the process after incrementing the nonce.

- As the hash rate of the Bitcoin network increased, the total amount of the 32-bit nonce was exhausted too quickly. In order to address this issue, the extra nonce solution was implemented, whereby the Coinbase transaction is used to provide a larger range of nonces to be searched by the miners.

This process is visualized in the following flowchart:

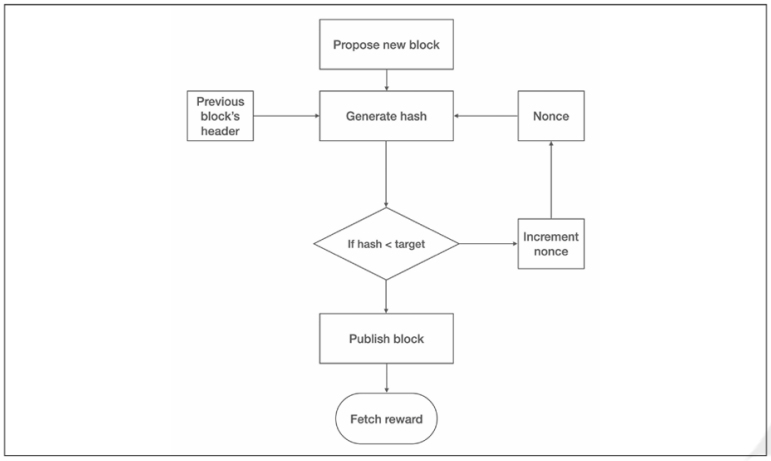


Figure 6.19: Mining process

The hash rate

- The hash rate basically represents the rate of hash calculation per second. In other words, this is the speed at which miners in the Bitcoin network are calculating hashes to find a block.
- In the early days of Bitcoin, it used to be quite small, as CPUs were used, which are relatively weak in mining terms. However, with dedicated mining pools and Application Specific Integrated Circuits (ASICs) now, this has gone up exponentially in the last few years. This has resulted in increased difficulty in the Bitcoin network.

The following hash rate graph shows the hash rate increases over time and is currently measured in exa-hashes. This means that in 1 second, Bitcoin network miners are computing more than 24,000,000,000,000,000,000 hashes per second:

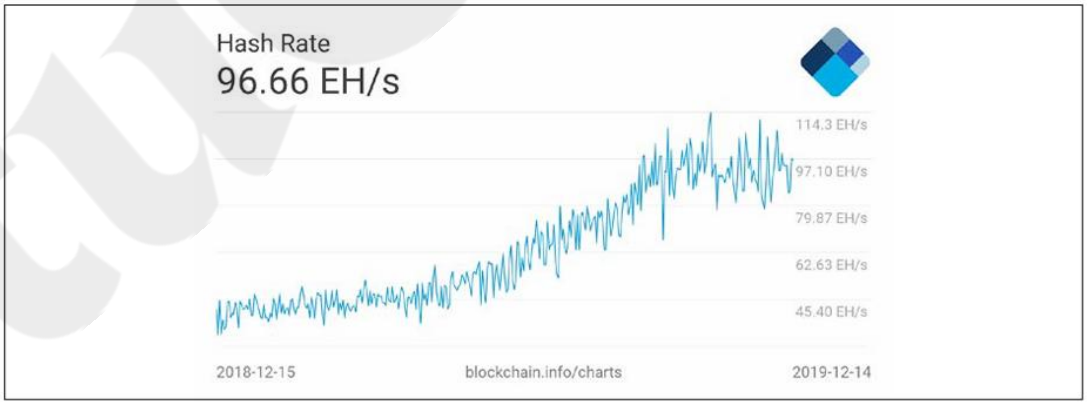


Figure 6.21: Hashing rate over time (measured in exa-hashes), shown over a period of 1 year

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, over time, experts have developed sophisticated systems to calculate the hash faster and faster. The following is a review of the different types of mining methods used in Bitcoin and how they evolved with time.

- **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.
  - CPU mining only lasted for around a year from the introduction of Bitcoin, and soon other methods were explored and tried by the miners.
- **GPU** Due to the increased difficulty of the Bitcoin network and the general tendency of finding faster methods to mine, miners started to use the 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 in parallel 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. From another angle, graphics cards have become quite expensive due to increased demand and this has impacted gamers and graphics software users.
- **FPGA** Even GPU mining did not last long, and soon miners found another way to perform mining using **Field Programmable Gate Arrays (FPGAs).**
  - An 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 were started too. FPGA offered much better performance 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 for the FPGA era of Bitcoin mining.
- **ASICs** ASICs were designed to perform SHA-256 operations. 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 due to the vast amounts of energy and money needed to be spent in order to build a profitable mining platform.
  - 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—a single user can run thousands of ASICs in parallel—

but it will require dedicated data centers and hardware, therefore the cost for a single individual can become prohibitive.

Examples of these four types of hardware are shown in the following photographs:

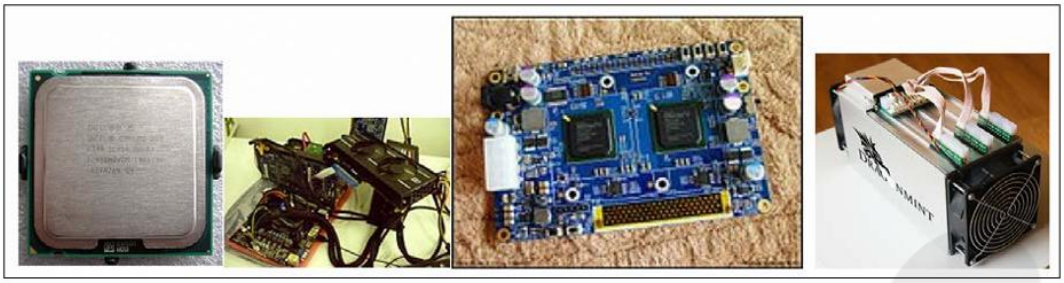


Figure 6.22: Four types of mining hardware (from left to right: a CPU, GPU, FPGA, and an ASIC)

Mining pools

A mining pool forms when a group of miners work together to mine a block.

- The pool manager receives the coinbase transaction if the block is successfully mined, and is then responsible for distributing the reward to the group of miners who invested resources to mine the block.
- This is more profitable than 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 (or more specifically, their individual node) solved the puzzle or not

Different types of mining pools:

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.
- Cloud services-> 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 (https://www.antpool. com), BTC (https://btc.com), and BTC.TOP (http://www.btc.top).

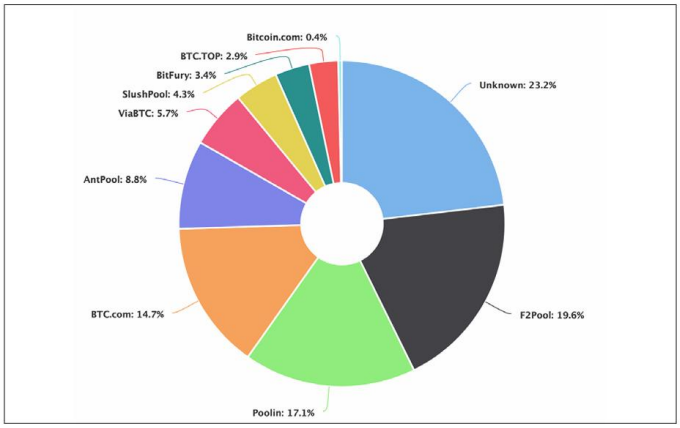


Figure 6.23: Mining pools and their hashing power (hash rate) as of late 2019. Source: https://blockchain.info/pools

THE BITCOIN NETWORK AND PAYMENTS

The Bitcoin network

- The Bitcoin network is a peer-to-peer (P2P) network where nodes perform transactions. They verify and propagate transactions and blocks. Nodes called miners also produce blocks.
- There are different types of nodes on the network. The **two** main types of nodes are **full nodes** and **simple payment verification (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 for all nodes in a Bitcoin network to perform all these functions.
- SPV nodes or lightweight clients perform only wallet and network routing functionality.
- Another type of node is **solo miner nodes**, which can perform mining, store full blockchains, and act as Bitcoin network routing nodes.
- There are a few nonstandard but heavily used nodes. These are called **pool protocol servers.** These nodes make use of alternative protocols, such as the stratum protocol. These nodes are used in mining pools.
- Nodes that only compute hashes use ***the stratum protocol*** to submit their solutions to the mining pool.
- Some nodes perform only mining functions and are called **mining nodes**. It is possible to run SPV software that runs a wallet and network routing function without a blockchain.
- SPV clients only download the headers of the blocks while syncing with the network. When required, they can request transactions from full nodes. Verifying transactions is possible by using a Merkle root in the block header with a Merkle branch to prove that the transaction is present in a block in the blockchain.
- There are also different protocols that have been developed to facilitate communication between Bitcoin nodes. One such protocol is called **Stratum**. It

is a line-based protocol that makes use of plain TCP sockets and human-readable JSON-RPC to operate and communicate between nodes. Stratum is commonly used to connect to mining pools.

A Bitcoin network is identified by its magic value. Magic values are used to indicate the message's origin network.

A list of these values is shown in the following table:

Network	Magic value (in hex)
main	0xD9B4BEF9
testnet	0xDAB5BFFA
testnet3	0x0709110B

- 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 an explanation of them 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.
  • **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 an IP address and port number.
  • **Ping:** This message is used to confirm if the TCP/IP network connection is active.
  • **Pong:** This message is the response to a ping message confirming that the network connection is live.

- 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.

- First, the client sends a protocol message, version, which 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 a verack message exchange between both nodes, indicating that the connection has been established.
- After this, getaddr and addr messages are exchanged to find the peers that the client does not know. Meanwhile, either of the nodes can send a ping message to see whether the connection is still active. getaddr and addr are message types defined in the Bitcoin protocol.

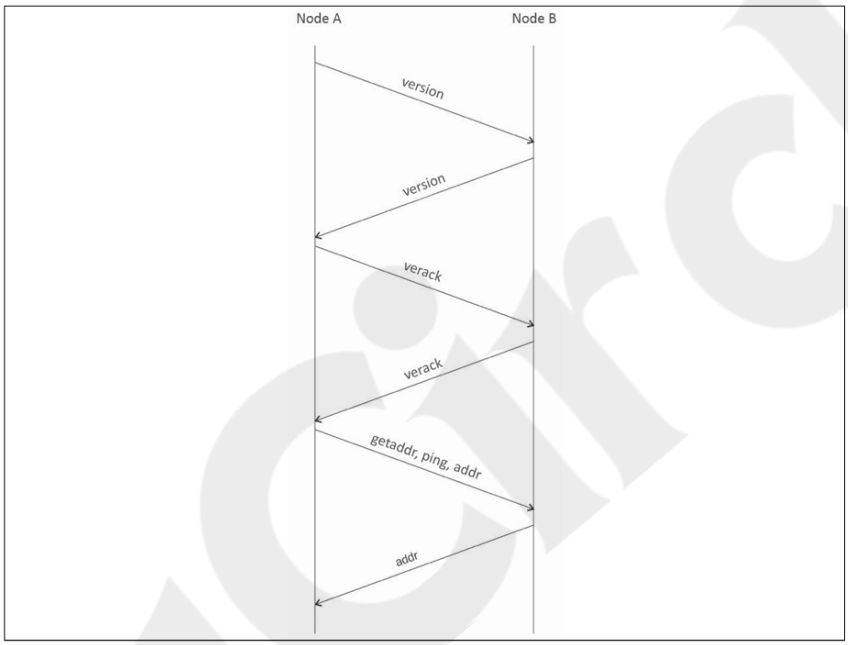


Figure 7.1: Visualization of node discovery protocol

- This network protocol sequence diagram shows communication between two Bitcoin nodes during initial connectivity. Node A is shown on the left-hand side and Node B on the right. First, Node A starts the connection by sending a version message that contains the version number and current time to the remote peer, Node B. Node B then responds with its own version message containing the version number and current time. Node A and Node B then exchange a verack message, indicating that the connection has been successfully established. After this connection is successful, the peers can exchange getaddr and addr messages to discover other peers on the network. Now, the block download can begin. If the node already has all the 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 has inventories already. If it does, 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. This was a slower process

known as the blocks-first approach and was replaced with the headers-first approach in 0.10.0.

- The initial block download can use the blocks-first or 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 on 16th February 2015 with the release of version 0.10.0. Since version 0.10.0, the initial block download method named headers-first was introduced. This resulted in major performance improvement, and blockchain synchronization that used to take days to complete started taking only a few hours.
- The core idea is that the new node first asks peers for block headers and then validates them. Once this is completed, blocks are requested in parallel from all available peers. This happens because the blueprint of the complete chain is already downloaded in the form of the block header chain. In this method, when the client starts up, it checks whether the blockchain is fully synchronized 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 blockchain 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.

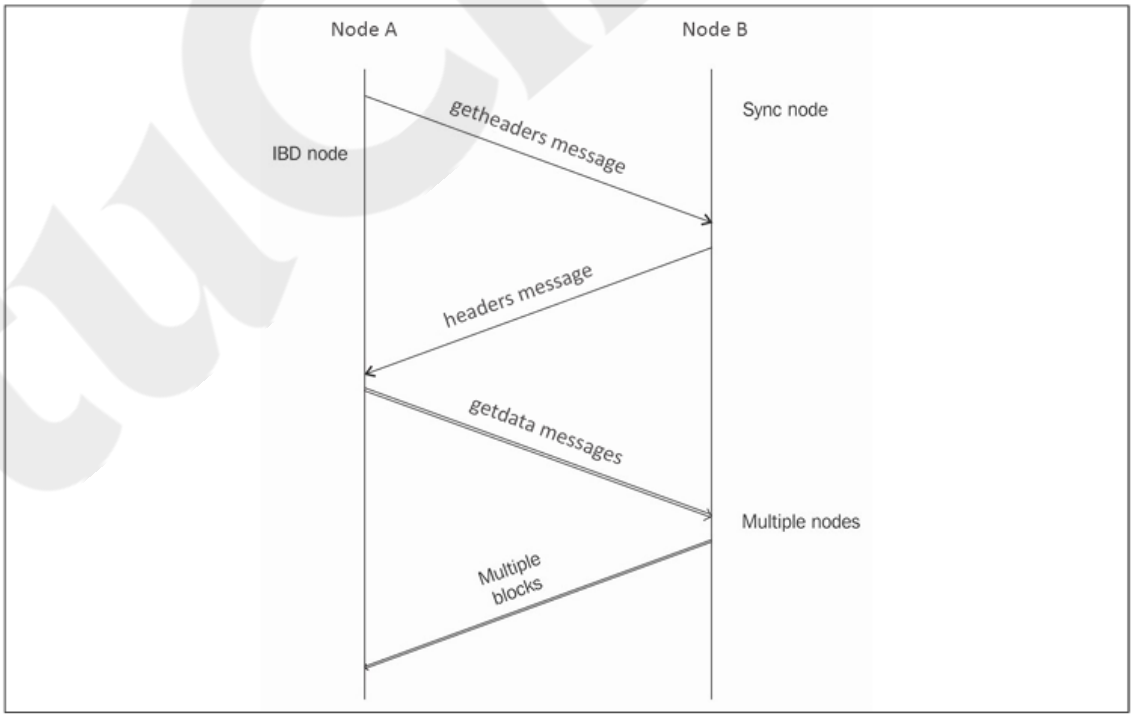


Figure 7.2: Bitcoin Core client >= 0.10.0 header and block synchronization

- The preceding diagram shows the Bitcoin block synchronization process between two nodes on the Bitcoin network. Node A, shown on the left-hand side, is called an Initial Block Download (IBD) node, and Node B, shown on the right, is called a sync node

- **IBD** node means that this is the node that is requesting the blocks, while **sync node** means the node where the blocks are being requested from.
- The process starts by Node A first sending the getheaders message, which is met with a getheaders message response from the sync node. The payload of the getheaders message is one or more header hashes. If it's a new node, then there is only the first genesis block's header hash.
- Sync Node B replies by sending up to 2,000 block headers to IBD Node A. After this, the IBD node, Node A, starts to download more headers from Node B and blocks from multiple nodes in parallel; that is, it acts as the IBD and receives multiple blocks from multiple nodes, including Node B.
- If the sync node does not have more headers than 2,000 when the IBD node makes a getheaders request, the IBD node sends a getheaders message to other nodes. This process continues in parallel until the blockchain synchronization is complete.
- The Getblockchaininfo and getpeerinfo Remote Procedure Calls (RPCs) were updated with a new functionality to cater for this change.
- An RPC known as getchaintips is used to list all known branches of the blockchain. This also includes headers-only blocks. Getblockchaininfo is used to provide information about the current state of the blockchain. getpeerinfo is used to list both the number of blocks and the headers that are common between peers.
- Wireshark can also be used to visualize message exchange between peers and can serve as an invaluable tool to learn about the Bitcoin protocol. A sample of this is shown here. This is a basic example showing the version, verack, getaddr, ping, addr, and inv messages.

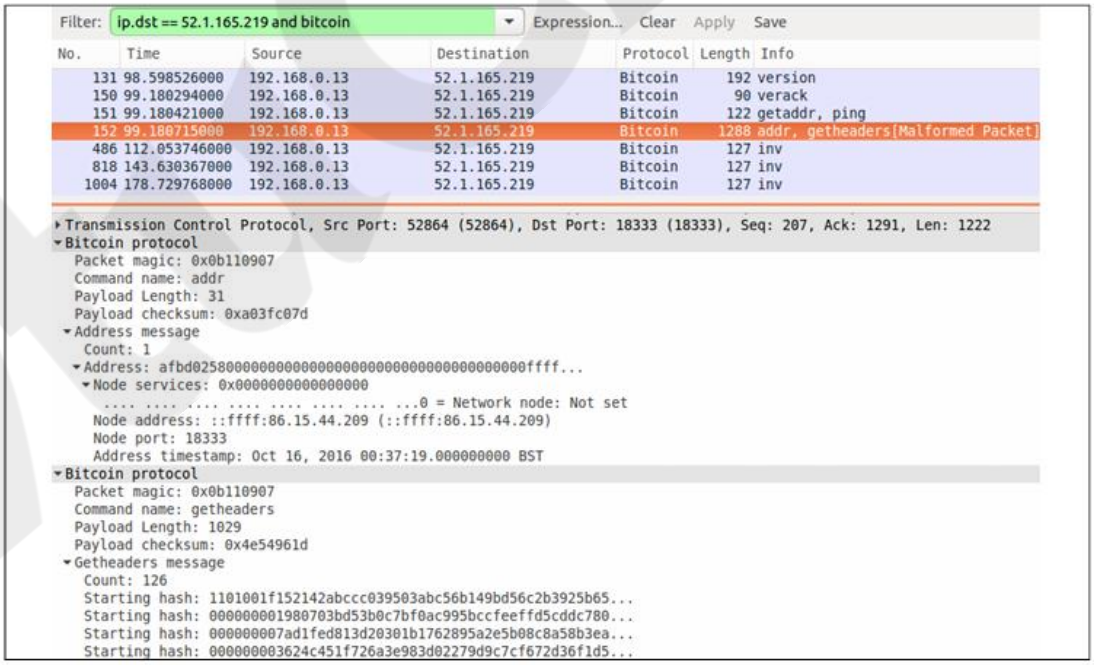


Figure 7.3: A sample block message in Wireshark

The analysis being performed here by Wireshark shows messages being exchanged between two nodes. If you look closely, you'll notice that the top three messages show the node discovery protocol that we introduced earlier:

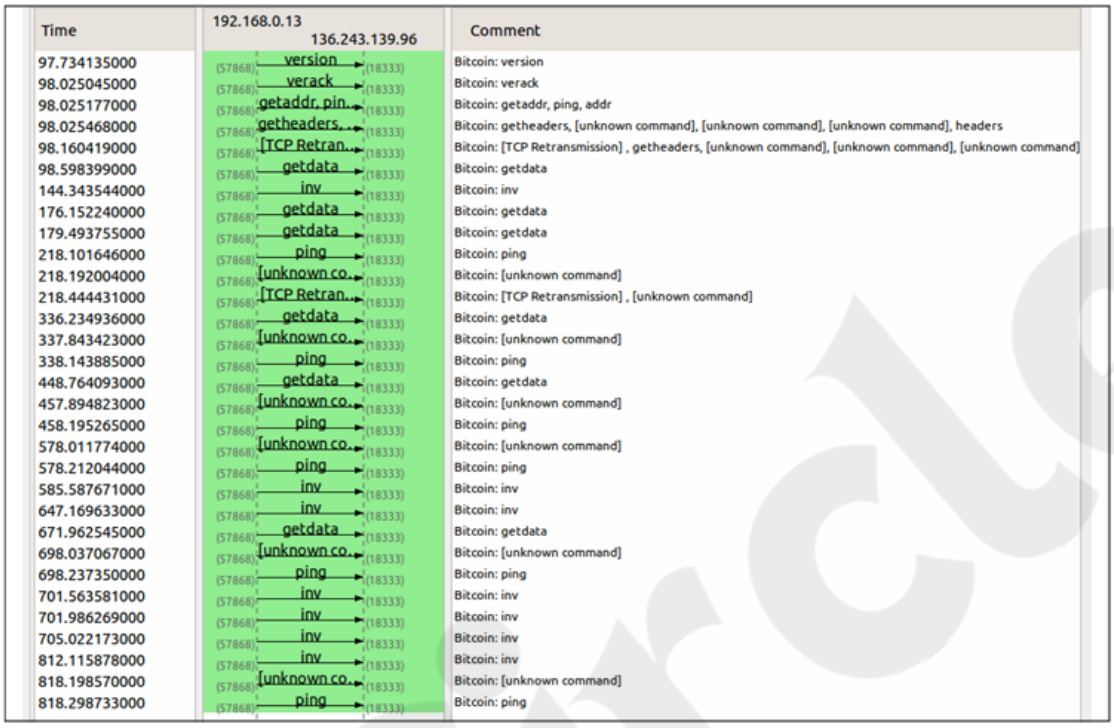


Figure 7.4: Bitcoin node discovery protocol in Wireshark

Nodes run different Bitcoin client software. The most common are full and SPV clients.

Full client and SPV client Bitcoin network nodes can fundamentally operate in two modes: full client or lightweight 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. 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 longest valid blockchain. Verification is performed by looking at the Merkle branch, which links the transactions to the original block the transaction was accepted in. This is not very practical and requires a more pragmatic approach, which was implemented with BIP37

Bloom filters A bloom filter is a data structure (a bit vector with indexes) that is used to test the membership of an element in a probabilistic manner. It provides probabilistic lookup with false positives but no false negatives

- This means that this filter can produce an output where an element that is not a member of the set being tested is wrongly considered to be in the set. Still, it can never produce an output where an element does exist in the set, but it asserts that it does not. In other words, false positives are possible, but false negatives are not.

- Elements are added to the bloom filter after hashing them several times and then setting the corresponding bits in the bit vector to 1 via the corresponding index. To check the presence of an element in the bloom filter, the same hash functions are applied and then compared with the bits in the bit vector to see whether the same bits are set to 1. Note that 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, murmur, and Jenkins. These filters are mainly used by simple payment verification SPV clients to request transactions and the Merkle blocks that 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 transactions and blocks that have been requested by the SPV client.
- Once version messages have been exchanged and the connection is established between the peers, the nodes can set filters according to their requirements. These probabilistic filters offer a varying degree of privacy or precision, depending on 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. BIP37 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

Wallets The wallet software is used to generate and store cryptographic keys. It performs various useful functions, such as receiving and sending Bitcoin, backing up keys, and keeping track of the balance available. Bitcoin client software usually offers both functionalities: Bitcoin client and wallet. On disk, the Bitcoin Core client wallets are stored as a Berkeley DB file

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

- Private keys are generated by randomly choosing a 256-bit number provided by the wallet software
- Private keys are used by wallets to sign the outgoing transactions. 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 number of bitcoins.

the common types of wallets.

- **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 also generates keys as and when required. Managing a large number of keys is very difficult and an error-prone process that can lead to the theft and loss of coins. Moreover, there is a need to create regular backups of the keys and protect them appropriately, for example, by encrypting them in order to prevent theft or loss.
- **Deterministic wallets** In this type of wallet, keys are derived from 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, a Bitcoin improvement proposal for Mnemonic code for generating deterministic keys.
- **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 hashes 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 a 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
- Hardware wallets Another method is to use a tamper-resistant device to store keys. This tamper-resistant device can be custom-built. 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.
- **Online wallets**:Online wallets, as the name implies, are stored entirely online and are provided as a service usually via the 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 trusts the online wallet service provider. An example of an online wallet is GreenAddress, which is available at https://greenaddress.it/en/.

- **Mobile wallets** Mobile wallets, as the name suggests, are installed on mobile devices. They can provide us with various methods to make payments, most notably the ability to use smartphone cameras to scan QR codes quickly and make payments. Mobile wallets are available for Android and iOS and include Blockchain Wallet, Breadwallet, Copay, and Jaxx:

Figure 7.6: Jaxx mobile wallet

- The choice of Bitcoin wallet depends on several factors such as security, ease of use, and available features. Out of all these attributes, security, of course, comes first, and when deciding about which wallet to use, security should be of paramount importance. Hardware wallets tend to be more secure compared to web wallets because of their tamper-resistant design. Web wallets, by their very nature, are hosted on websites, which may not be as secure as a tamper resistant hardware device.
- Generally, mobile wallets for smart phone devices are quite popular due to a balanced combination of features and security. There are many companies offering these wallets on the iOS App Store and Google Play. It is, however, quite difficult to suggest which type of wallet should be used as it also depends on personal preferences and the features available in the wallet. Security should be kept in mind while making a decision on which wallet to choose.

Bitcoin payments Bitcoin can be accepted as payment 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 number of ways in which buyers can pay a business that accepts Bitcoin. For example, on an online shop, Bitcoin merchant solutions can be used, whereas in traditional, physical shops, Point of Sale (POS) 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. A Uniform Resource Identifier (URI) 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. Businesses can use the following image to advertise that they can accept Bitcoin as payment from customers

Figure 7.7: "Bitcoin accepted here" logo

Various payment solutions, such as XBT terminal and the 34 Bytes Bitcoin POS terminal, are available commercially.

Generally, these solutions work by following these steps:

1. The salesperson enters the amount of money to be charged in fiat currency; for example, US dollars.
2. Once the value is entered in the system, the terminal prints a receipt with a QR code on it and other relevant information, such as the amount to be paid.
3. The customer can then scan this QR code using their mobile Bitcoin wallet to send the payment to the Bitcoin address of the seller embedded within the QR code.
4. Once the payment is received at the designated Bitcoin address, a receipt is printed out as physical evidence of the sale.

A Bitcoin POS device from **34 Bytes** is shown in the following image:

Figure 7.8: 34 Bytes POS solution

Bitcoin payment processors are offered by many online service providers. It allows integration with e-commerce websites to facilitate Bitcoin payments. These payment processors can be used to accept Bitcoin as

payment. Some service providers also allow secure storage of Bitcoin, for example, BitPay (https://bitpay.com).

- Another example is the Bitcoin merchant solutions available at https://www.bitcoin.com/merchant-solutions. Various BIPs have been proposed and finalized in order to introduce and standardize Bitcoin payments. Most notably, BIP70 (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 are defense 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 other BIPs, such as BIP71 (Payment Protocol MIME types) and BIP72 (URI extensions for Payment Protocol), have also been implemented to standardize payment scheme to support BIP70 (Payment Protocol).
- Another innovative development is the Lightning Network. It is a solution for scalable off-chain instant payments. It was introduced in early 2016 and allows off-blockchain payments. This network makes use of payments channels that run off the blockchain, which allows greater speed and scalability of Bitcoin.

Innovation in Bitcoin

- Bitcoin has undergone many changes and is still evolving into a more and more robust and better system by addressing various weaknesses in the system. Performance has been a topic of hot debate among Bitcoin experts and enthusiasts for many years. As such, various proposals have been made in the last few years to improve Bitcoin performance, resulting in greater transaction speed, increased security, payment standardization, and overall performance improvement at the protocol level. These improvement proposals are usually made in the form of **Bitcoin Improvement Proposals (BIPs)** or fundamentally new versions of Bitcoin protocols, resulting in new networks altogether.
- Some of the changes proposed can be implemented via a soft fork, but a few need a hard fork and, as a result, give birth to a new currency. In the following sections, we will look at the various BIPs that can be proposed for improvement in Bitcoin. These documents, also referred to as BIPs, are used to propose improvements or inform the Bitcoin community about the improvements suggested, the design issues, or some aspects of the Bitcoin ecosystem.

- There are **three types of 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

Advanced Protocols

various advanced protocols that have been suggested or implemented for improving the Bitcoin protocol

For example, transaction throughput is one of the critical issues that need a solution. The Bitcoin network can only process approximately three to seven transactions per second, which is a tiny number compared to other financial networks. For example, the Visa network can process approximately, on average, 24,000 transactions per second. PayPal can process approximately 200 transactions per second, whereas Ethereum can process up to, on average, 20. As the Bitcoin network has grown exponentially over the last few years, these issues have started to grow even further. The difference in processing speed is also shown in the following graph, which shows the scale of difference between Bitcoin and other networks' transaction speeds. The graph uses a logarithmic scale, which demonstrates the vast difference between the networks' transaction speeds.

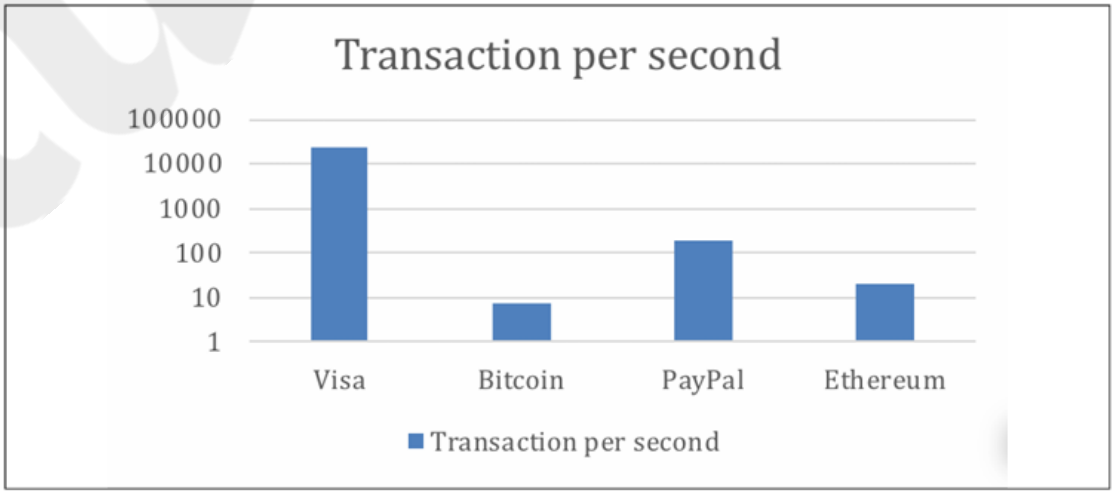


Figure 7.9: Bitcoin transaction speed compared to other networks (on a logarithmic scale)

1. Segregated Witness The SegWit or Segregated Witness is a soft fork-based upgrade of the Bitcoin protocol that addresses

weaknesses such as throughput and security in the Bitcoin protocol. SegWit offers a number of improvements, as listed here:

- **Fix for transaction malleability** due to the separation of signature data from transactional data. In this case, it is no longer possible to modify the transaction ID because it is no longer calculated based on the signature data present within the transaction.

  • By segregating the signature data and transaction data, lightweight clients do not need to download the transactions with all signatures unnecessarily. The transactions can be verified without the useless signatures, which allows for increased bandwidth efficiency.

  • Reduction in transaction signing and verification times, which results in faster transactions. A new transaction hashing algorithm for signature verification has been introduced and is detailed in BIP0143 (https://en.bitcoin.it/wiki/BIP_0143). Due to this change, the verification time grows linearly with the number of inputs instead of in a quadratic manner, resulting in quicker verification time.

  • Script versioning capability, which allows for easier script language upgrades. The version number is prefixed to the locking scripts to depict the version. This change allows upgrades and improvements to be made to the scripting language, without requiring a hard fork, by just increasing the version number of the script.

  • Increased block size by introducing a weight limit instead of a size limit on the block and the removal of signature data. This concept will be explained in more detail shortly.

  • An improved address format, also called a "bc1 address," which is encoded using the Bech32 mechanism instead of base58. This improvement allows for better error detection and correction. Also, all characters are lowercase, which helps with readability. Moreover, this helps with distinguishing between legacy transactions and SegWit transactions.

- SegWit was proposed in BIP 141, BIP 143, BIP 144, and BIP 145. It was activated on Bitcoin's main network on August 24, 2017 at block number 481824. The key idea behind SegWit is the separation of signature data from transaction data (that is, a transaction Merkle tree), which results in the size of the transaction being reduced. This change allows the block size to increase up to 4 MB in size. However, the practical limit is between 1.6 MB and 2 MB. Instead of a hard size limit of 1 MB blocks,
- SegWit introduced a new concept of a block weight limit. Block weight is a new restriction mechanism where each transaction has a weight associated with it. This weight is calculated on a per-transaction basis. The formula used to calculate it is:

$$Weight = (Transaction\ size\ without\ witness\ data)\ X\ 3 + (Transaction\ size)$$

- Blocks can have a maximum of four million weight units. As a comparison, a byte in a legacy 1 MB block is equivalent to 4 weight units, but a byte in a SegWit block weighs only 1 weight unit. This modification immediately results in increased block capacity.
- To spend an unspent transaction output (UTXO) in Bitcoin, a valid signature needs to be provided. In the pre-SegWit scenario, this signature is provided within the locking script, whereas in SegWit this signature is not part of the transaction and is provided separately.
- ***There are four types of transactions introduced by SegWit.*** These types are:

  1. Pay to Witness Public Key Hash (P2WPKH): This type of script is similar to the usual P2PKH, but the crucial difference is that the transaction signature used as a proof of ownership in ScriptSig is moved to a separate structure known as the "witness" of the input. The signature is the same as P2PKH but is no longer part of ScriptSig; it is simply empty. The PubKey is also moved to the witness field. This script is identified by a 20 byte hash. The ScriptPubKey is modified to a simpler format, as shown here:

  - P2PKH ScriptPubKey:

    ```
    OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG
    ```

  - P2WPKH ScriptPubKey:

    ```
    OP_0 <pubKeyHash>
    ```

  2. Pay to Script Hash - Pay to Witness PubKey Hash (P2SH-P2WPKH): This is a mechanism introduced to make SegWit transactions backward-compatible. This is made possible by nesting the P2WPKH inside the usual P2SH.

  3. Pay to Witness Script Hash (P2WSH): This script is similar to legacy P2SH but the signature and redeem script are moved to the separate witness field. This means that ScriptSig is simply empty. This script is identified by a 32-byte SHA-256 hash. P2WSH is a simpler script compared to P2SH and has just two fields. The ScriptPubKey is modified as follows:

  - P2SH ScriptPubKey:

    ```
    OP_HASH160 <pubKeyHash> OP_EQUAL
    ```

  - P2WSH ScriptPubKey:

    ```
    OP_0 <pubKeyHash>
    ```

  4. Pay to Script Hash - Pay to Witness Script Hash (P2SH-P2WSH): Similar to P2SH P2WPKH, this is a mechanism that allows backward-

compatibility with legacy Bitcoin nodes. SegWit adoption is still in progress as not all users of the network agree or have started to use SegWit.

Bitcoin Cash

Bitcoin Cash (BCH) increases the block limit to 8 MB. This change immediately increases the number of transactions that can be processed in one block to a much larger number compared to the 1 MB limit in the original Bitcoin protocol. It uses Proof of Work (PoW) as a consensus algorithm, and mining hardware is still ASIC-based. The block interval is changed from 10 minutes to 10 seconds and up to 2 hours. It also provides replay protection and wipe-out protection, which means that because BCH uses a different hashing algorithm, it prevents it being replayed on the Bitcoin blockchain. It also has a different type of signature compared to Bitcoin to differentiate between two blockchains.

Bitcoin Unlimited

Bitcoin Unlimited increases the size of the block without setting a hard limit. Instead, miners come to a consensus on the block size cap over a period of time. Other concepts such as extremely thin blocks and parallel validation have also been proposed in Bitcoin Unlimited.

Extremely thin blocks allow for faster block propagation between Bitcoin nodes. In this scheme, the node requesting blocks sends a getdata request, along with a bloom filter, to another node. The purpose of this bloom filter is to filter out the transactions that already exist in the memory pool (mempool) of the requesting node. The node then sends back a thin block only containing the missing transactions. This fixes an inefficiency in Bitcoin whereby transactions are regularly received twice – once at the time of broadcast by the sender and then again when a mined block is broadcasted with the confirmed transaction. Parallel validation allows nodes to validate more than one block, along with new incoming transactions, in parallel. This mechanism is in contrast to Bitcoin, where a node, during its validation period after receiving a new block, cannot relay new transactions or validate any blocks until it has accepted or rejected the block

Bitcoin Gold

This proposal has been implemented as a hard fork since block 491407 of the original Bitcoin blockchain. Being a hard fork, it resulted in a new blockchain, named Bitcoin Gold. The core idea behind this concept is to address the issue of mining centralization,

which has hurt the original Bitcoin idea of decentralized digital cash, whereby more hash power has resulted in a power shift toward miners with more hashing power. Bitcoin Gold uses the Equihash algorithm as its mining algorithm instead of PoW; hence, it is inherently ASIC resistant and uses GPUs for mining

Bitcoin investment and buying and selling Bitcoin

There are many online exchanges where users can buy and sell Bitcoin. 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 Bitcoin or trade by opening long or short positions to make a profit when the price of Bitcoin goes up or down. Several other features, such as exchanging Bitcoin 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 (https://cex.io) here. Other exchanges offer similar types of services

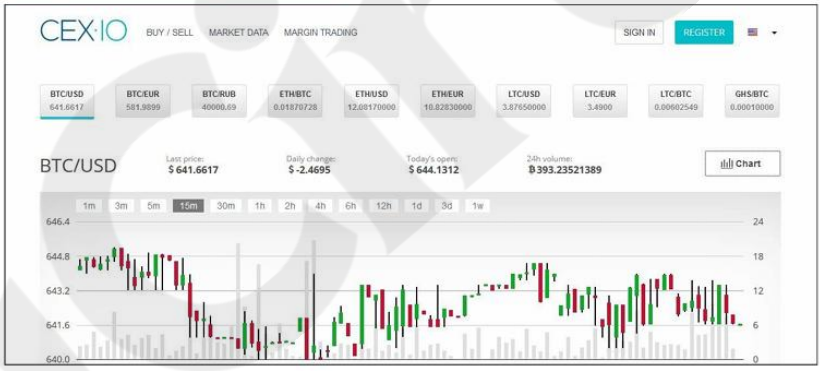


Figure 7.10: Example of the Bitcoin exchange on cex.io

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

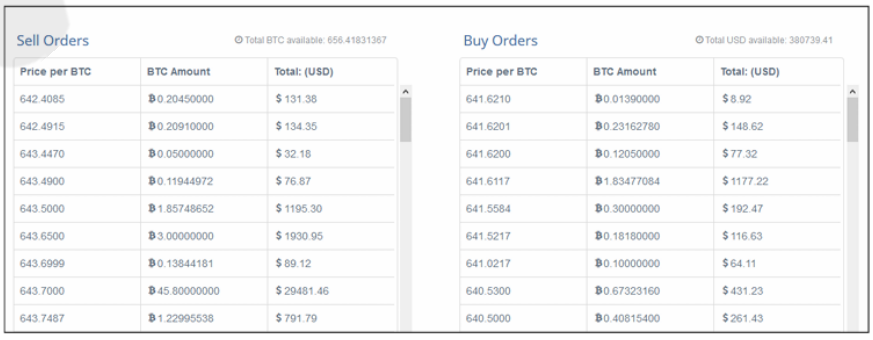


Figure 7.11: Example of a Bitcoin order book at the exchange of cex.io

The order book shown here displays sell and buy orders. Sell orders are also called ask orders, while buy orders are also called bid orders. This means that the ask price is what the seller is willing to sell the bitcoin at, whereas the bid price is what the buyer is willing to pay. If the bid and ask prices match, then a trade can occur. The

most common order types are market orders and limit orders. Market orders mean that as soon as the prices match, the order will be fulfilled immediately. Limit orders allow for buying and selling a set number of bitcoins at a specified price or better. Also, a period of time can be set, during which the order can be left open.

Smart Contracts

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

A smart contract is a secure and unstoppable computer program representing an agreement that is automatically executable and enforceable.

- Dissecting this definition reveals that a smart contract is, fundamentally, 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 fundamental idea is that smart contracts are automatically executed according to the instruction that is coded in, for example, when certain conditions satisfy. They are enforceable, which means that all contractual terms perform as specified and expected, even in the presence of adversaries. Enforcement is a broader term that encompasses traditional enforcement in the form of a law, along with the implementation of specific measures and controls that

make it possible to execute contract terms without requiring any intervention.

- Preferably, smart contracts should not rely on any traditional methods of enforcement. Instead, they should work on the principle that code is the law, which means that there is no need for an arbitrator or a third party to enforce, control, or influence the execution of a smart contract.
- Smart contracts are self-enforcing as opposed to legally enforceable. This idea may sound like a libertarian's dream, but it is entirely possible and is in line with the true spirit of smart contracts. Moreover, they are secure and unstoppable, which means that these computer programs are fault-tolerant and executable in a reasonable (finite) amount of time. These programs should be able to execute and maintain a healthy internal state, even if external factors are unfavorable. For example, imagine a typical computer program that is encoded with some logic and executes according to the instruction coded within it. However, if the environment it is running in or the external factors it relies on deviate from the usual or expected state, the program may react arbitrarily or abort. Smart contracts must be immune to this type of issue

A smart contract has the following properties:

• **Automatically executable:** It is self-executable on a blockchain without requiring any intervention.

• **Enforceable:** This means that all contract conditions are enforced automatically.

• **Secure:** This means that smart contracts are tamper-proof (or tamper-resistant) and run with security guarantees. The underlying blockchain usually provides these security guarantees; however, the smart contract programing language and the smart contract code themselves must be correct, valid, and verified.

• **Deterministic:** The deterministic feature ensures that smart contracts always produce the same output for a specific input. Even though it can be considered to be part of the secure property, defining it here separately ensures that the deterministic property is considered one of the important properties.

• **Semantically sound**: This means that they are complete and meaningful to both people and computers.

• **Unstoppable:** This means that adversaries or unfavorable conditions cannot negatively affect the execution of a smart contract. When the smart contracts execute, they complete their performance deterministically in a finite amount of time.

Ricardian contracts

Ricardian contracts were initially proposed in the paper, Financial Cryptography in 7 Layers, by Ian Grigg, in the late 1990s. This paper is available at https://iang.org/papers/fc7.html.

- Ricardian contracts were initially used in a bond trading and payment system called Ricardo. The fundamental idea behind this contract is to write a document that is understood and accepted by both a court of law and computer software. Ricardian contracts address the challenge of the issuance of value over the internet. A Ricardian contract identifies the issuer and captures all the terms and clauses of the contract in a document to make it acceptable as a legally binding contract.
- A Ricardian contract is a document that has several of the following properties:
  - It is a contract offered by an issuer to holders
  - It is a valuable right held by holders and managed by the issuer
  - It can be easily read by people (like a contract on paper)
  - It can be read by programs (parsable, like a database)
  - It is digitally signed
  - It carries the keys and server information
  - It is allied with a unique and secure identifier

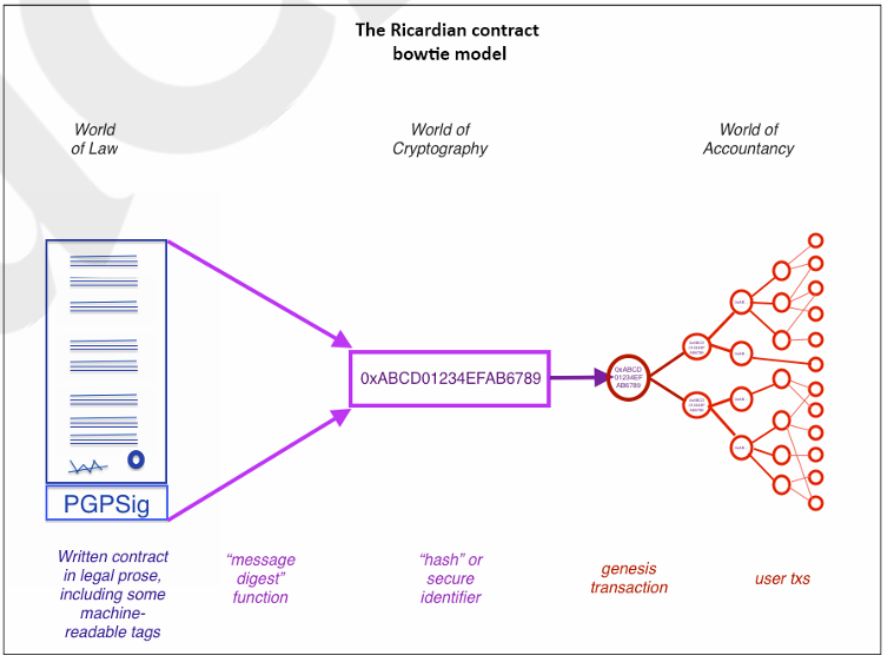


Figure 10.1: The Ricardian Contract bowtie diagram

The diagram shows a number of elements:

- The **World of Law** is on the left-hand side from where the document originates. This document is a written contract in legal prose with some machine-readable tags.

• This document is then hashed.
• The resultant message digest is used as an identifier throughout the **World of Accountancy**, as shown on the right-hand side of the diagram.
The World of Accountancy element represents any accounting, trading, and information systems that are being used in the 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 it is used in every transaction as an identifier throughout the operational execution of the contract. This way, a secure link is created between the original written contract and every transaction in the World of Accounting:

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. The second type encompasses both the denotational and operational semantics of a legal agreement. It perhaps makes sense to categorize smart contracts based on the difference between the semantics, but it is better to consider a smart contract as a standalone entity that is capable of encoding legal prose and code (business logic).

In Bitcoin, a straightforward implementation of basic smart contracts (conditional logic) can be observed, which is entirely oriented toward the execution and performance of the contract, whereas a Ricardian contract is more geared toward producing a document that is understood by humans with some parts that a computer program can understand. This can be viewed as legal semantics versus operational performance (semantics versus performance), as shown in the following diagram. The diagram shows that Ricardian contracts are more semantically-rich, whereas smart contracts are more performance-rich. This concept was

initially proposed by Ian Grigg in his paper, On the intersection of Ricardian and Smart Contracts.

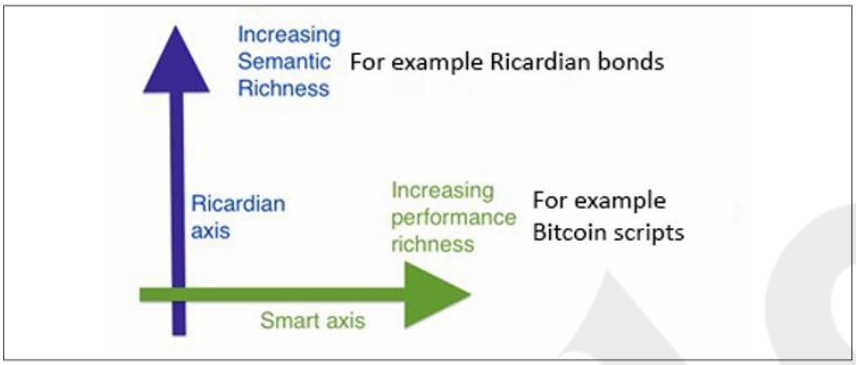


Figure 10.2: Diagram explaining that performance versus semantics is an orthogonal issue, as described by Ian Grigg; it is slightly modified to show examples of different types of contracts on both axes

A smart contract is made up of both of these elements (performance and semantics) embedded together, which completes the 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 natural language; code represents the program that is a computer-understandable representation of legal prose; and the parameters join the appropriate parts of the legal contract to the equivalent code.

Smart contract templates

Smart contracts can be implemented in any industry where they are required, but the most popular use cases relate to the financial sector. This is because blockchain first found many use cases in the finance industry and, therefore, sparked enormous research interest in the financial industry long before other areas. Recent work in the smart contract space specific to the financial sector 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.

Christopher D. Clack et al. proposed this idea in their paper published in 2016, named Smart Contract Templates: Foundations, design landscape and research directions.

The paper also suggested that domain-specific languages (DSLs) should be built to support the design and implementation of smart contract templates. A language named **common language for augmented contract knowledge (CLACK)** has been proposed, and research has started to develop this language. This language is intended to be very rich and is expected to provide a large variety

of functions ranging from supporting legal prose to the ability to be executed on multiple platforms and cryptographic functions.

The main aim of this paper is to investigate how legal prose could be linked with code using a markup language. It also covers how smart legal agreements can be created, formatted, executed, and serialized for storage and transmission. This work is ongoing and remains an open area for further research and development. Contracts in the finance industry are not a new concept, and various DSLs are already in use in the financial services industry to provide a specific language for a particular domain. For example, there are DSLs available that support the development of insurance products, represent energy derivatives, or are being used to build trading strategies.

It is also essential to understand the concept of DSLs, as this type of programming language can be developed to program smart contracts. **DSLs** are different from general-purpose programming languages (**GPLs).** DSLs have limited expressiveness for a particular application or area of interest. These languages possess a small set of features that are sufficient and optimized for a specific domain only. Unlike GPLs, they are not suitable for building large general-purpose application programs. Based on the design philosophy of DSLs, it can be envisaged that such languages will be developed specifically to write smart contracts. Some work has already been done, and Solidity is one such language that has been introduced with the Ethereum blockchain to write smart contracts.

Vyper is another language that has been recently introduced for Ethereum smart contract development. This idea of DSLs for smart contract programming can be further extended to a GPL. A smart contract modeling platform can be developed where a domain expert (not a programmer but a front desk dealer, for example) can use a graphical user interface and a canvas (drawing area) to define and illustrate the definition and execution of a financial contract. Once the flow is drawn and completed, it can be emulated first to test it and then be deployed from the same system to the target platform, which can be a smart contract on a blockchain or even a complete decentralized application (DApp). This concept is also not new, and a similar approach is already used in a non-blockchain domain, in the Tibco StreamBase product, which is a Java based system used for building event-driven, high-frequency trading systems. It has been proposed that research should also be conducted in the area of developing high-level DSLs that can be used to program a smart contract in a user-friendly graphical user interface, thus allowing a non-programmer domain expert (for example, a lawyer) to design smart contracts. Apart from DSLs,

there is also a growing interest in using general-purpose, already established programming languages like Java, Go, and C++ to be used for smart contract programming. This idea is appealing, especially from a usability point of view, where a programmer who is already familiar with, for example, Java, can use their skills to write Java code instead of learning a new language. The high-level language code can then be compiled into a low level bytecode for execution on the target platform. There are already some examples of such systems, such as in EOSIO blockchains, where C++ can be used to write smart contracts, which are compiled down to the web assembly for execution. An inherent limitation with smart contracts is that they are unable to access any external data. The concept of oracles was introduced to address this issue. An oracle is an off-chain source of information that provides the required information to the smart contracts on the blockchain.

Oracles

Oracles are an essential component of the smart contract and blockchain ecosystem. The limitation with smart contracts is that they cannot access external data because blockchains are closed systems without any direct access to the real world. This external data might be required to control the execution of some business logic in the smart contract; for example, the stock price of a security product that is required by the contract to release dividend payments. In such situations, oracles can be used to provide external data to smart contracts. An oracle can be defined as an interface that delivers data from an external source to smart contracts. Oracles are trusted entities that use a secure channel to transfer off-chain data to a smart contract. The following diagram shows a generic model of an oracle and smart contract ecosystem.

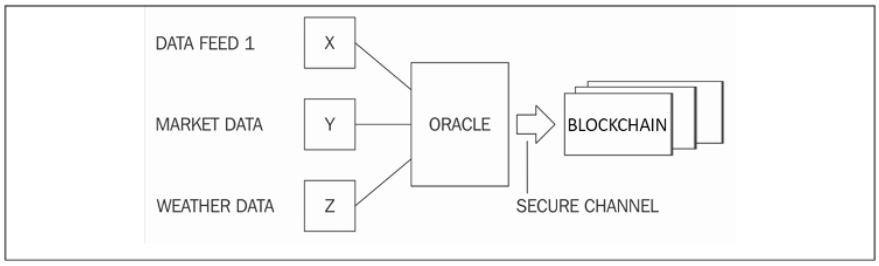


Figure 10.3: A generic model of an oracle and smart contract ecosystem

Depending on the industry and use case requirements, oracles can deliver different types of data ranging from weather reports, real-world news, and corporate actions to data coming from an Internet of Things (IoT) device. A list of some of the common use cases of oracles is shown here.

Type of data	Examples	Use case
Market data	Live price feeds of financial instruments. Exchange rates, performance, pricing, and historic data of commodities, indices, equities, bonds, and currencies.	DApps related to financial services, for example, decentralized exchanges and **decentralized finance (DeFi)**
Political events	Election results	Prediction markets
Travel information	Flight schedules and delays	Insurance DApps
Weather information	Flooding, temperature, and rain data	Insurance DApps
Sports	Results of football, cricket, and rugby matches	Prediction markets
Telemetry	Hardware IoT devices, sensor data, vehicle location, and vehicle tracker data	Insurance DApps Vehicle fleet management DApps

There are different methods used by oracles to write data into a blockchain, depending on the type of blockchain used. For example, in a Bitcoin blockchain, an oracle can write data to a specific transaction, and a smart contract can monitor that transaction in the blockchain and read the data. Other methods include storing the fetched data in a smart contract's storage, which can then be accessed by other smart contracts on the blockchain via requests between smart contracts depending on the platform. For example, in Ethereum, this can be achieved by using message calls. **The standard mechanics of how oracles work is presented here:**

1. A smart contract sends a request for data to an oracle.
2. The request is executed and the required data is requested from the source. There are various methods of requesting data from the source. These methods usually involve invoking APIs provided by the data provider, calling a web service, reading from a database (for example, in enterprise integration use cases where the required data may exist on a local enterprise legacy system), or requesting data from another blockchain. Sources can be any external off-chain data provider on the internet or in an internal enterprise network.
3. The data is sent to a notary to generate cryptographic proof (usually a digital signature) of the requested data to prove its validity (authenticity).

Usually, TLSNotary is used for this purpose (https://tlsnotary.org). Other techniques include Android proofs, Ledger proofs, and trusted hardware-assisted proofs, which we will explain shortly.

4. The data with the proof of validity is sent to the oracle.
5. The requested data with its proof of authenticity can be optionally saved on a decentralized storage system such as Swarm or IPFS and can be used by the smart contract/blockchain for verification. This is especially useful when the proofs of authenticity are of a large size and sending them to the requesting smart contracts (storing them on the chain) is not feasible.

6. Finally, the data, with the proof of validity, is sent to the smart contract.

This process can be visualized in the following diagram

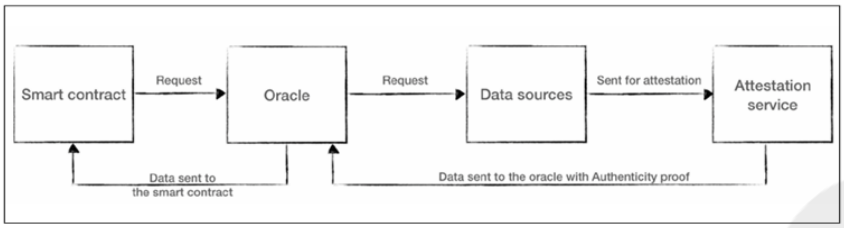


Figure 10.4: A generic oracle data flow

The preceding diagram shows the generic data flow of a data request from a smart contract to the oracle. The oracle then requests the data from the data source, which is then sent to the attestation service for notarization. The data is sent to the oracle with proof of authenticity.

Finally, the data is sent to the smart contract with cryptographic proof (authenticity proof) that the data is valid. Due to security requirements, oracles should also be capable of digitally signing or digitally attesting the data to prove that the data is authentic. This proof is called **proof of validity or proof of authenticity**.

Smart contracts subscribe to oracles. Smart contracts can either pull data from oracles, or oracles can push data to smart contracts. It is also necessary that oracles should not be able to manipulate the data they provide and must be able to provide factual data. Even though oracles are trusted (due to the associated proof of authenticity of data), it may still be possible that, in some cases, the data is incorrect due to manipulation or a fault in the system. Therefore, oracles must not be able to modify the data. This validation can be provided by using various cryptographic proofing schemes.

Software and network-assisted proofs

As the name suggests, these types of proofs make use of software, network protocols, or a combination of both to provide validity proofs. One of the prime examples of such proofs is TLSNotary, which is a technology developed to be primarily used in the PageSigner project to provide web page notarization. This mechanism can also be used to provide the required security services to oracles.

TLSNotary This protocol provides a piece of irrefutable evidence to an auditor that specific web traffic has occurred between a client and a server. It is based on Transport Layer Security (TLS), which is

a standard security mechanism that enables secure, bidirectional communication between hosts. It is extensively used on the internet to secure websites and allow HTTPS traffic.

The link provided is only for TLS version 1.0 as TLSNotary only supports TLS version 1.0 or 1.1. The key idea behind using TLSNotary is to utilize the TLS handshake protocol's feature, which allows the splitting of the TLS master key into three parts. Each part is allocated to the server, the auditee, and the auditor. The oracle service provider (https://provable.xyz) becomes the auditee, whereas an Amazon Web Services (AWS) instance, which is secure and locked down, serves as the auditor. In summary, to prove the authenticity of the data retrieved by oracles from external sources, attestation mechanisms such as TLSNotary are used, which produce verifiable and auditable proofs of communication between the data source and the oracle. This proof of authenticity ensures that the data fed back to the smart contract is indeed retrieved from the source.

TLS-N based mechanism

This mechanism is one of the latest developments in this space. TLS-N is a TLS extension that provides secure non-repudiation guarantees. This protocol allows you to create privacy preserving and non-interactive proofs of the content of a TLS session. TLS-N based oracles do not need to trust any third-party hardware such as Intel SGX or TLSNotary type service to provide authenticity proofs of data web content (data) to the blockchain. In contrast to TLSNotary, this scheme works on the latest TLS 1.3 standard, which allows for improved security.

Hardware device-assisted proofs

As the name suggests, these proofs rely on some hardware elements to provide proof of authenticity. In other words, they require specific hardware to work.

Android proof

This proof relies on Android's SafetyNet software attestation and hardware attestation to create a provably secure and auditable device.

SafetyNet validates that a genuine Android application is being executed on a secure, safe, and untampered hardware device. Hardware attestation validates that the device has the latest version of the OS, which helps to prevent any exploits that existed due to vulnerabilities in the previous versions of the OS. This secure device is then used to fetch data from third-party sources, ensuring tamper-proof HTTPS connections. The very use of a provably secure

device provides the guarantee and confidence (that is, a proof of authenticity) that the data is authentic.

Ledger proof

Ledger proof relies on the hardware cryptocurrency wallets built by the ledger company (https://www.ledger.com). Two hardware wallets, Ledger Nano S and Ledger Blue, can be used for these proofs. The primary purpose of these devices is as secure hardware cryptocurrency wallets. However, due to the security and flexibility provided by these devices, they also allow developers to build other applications for this hardware. These devices run a particular OS called **Blockchain Open Ledger Operating System (BOLOS)**, which, via several kernel-level APIs, allows device and code attestation to provide a provably secure environment.

The secure environment provided by the device can also prove that the applications that may have been developed by oracle service providers and are running on the device are valid, authentic, and are indeed executing on the **Trusted Execution Environment (TEE)** of the ledger device. This environment, supported by both code and device attestation, provides an environment that allows you to run a third-party application in a secure and verifiable ledger environment to provide proof of data authenticity. Currently, this service is used by Provable, an oracle service, to provide untampered random numbers to smart contracts.

Currently, as these devices do not connect to the internet directly, the ledger devices cannot be used to fetch data from the internet.

Trusted hardware-assisted proofs This type of proof makes use of trusted hardware, such as TEEs. A prime example of such a hardware device is **Intel SGX.** A general approach that is used in this scenario is to rely on the security guarantees of a secure and trusted execution provided by the secure element or enclave of the TEE device. A prime example of a trusted hardware-assisted proof is Town Crier (https://www.town-crier.org), which provides an authenticated data feed for smart contracts. It uses Intel SGX to provide a security guarantee that the requested data has come from an existing trustworthy resource.

Town Crier also provides a confidentiality service, which allows you to run confidential queries. The query for the data request is processed inside SGX Enclave, which provides a trusted execution guarantee, and the requested data is transmitted using a TLS-secured network connection, which provides additional data integrity guarantees

An issue can already be seen here, and that is the issue of trust. With oracles, we are effectively trusting a third party to provide us with the correct data.

- What if these data sources turn malicious, or simply due to a fault start provide incorrect data to the oracles?
- What if the oracle itself fails or the data source stops sending data? This issue can then damage the whole blockchain trust model. This phenomenon is called the **Blockchain oracle problem**.
- How do you trust a third party about the quality and authenticity of the data they provide?
- This question is especially real in the financial world, where market data must be accurate and reliable. There are several proposed ways to overcome this issue. These solutions range from merely trusting a reputable third party to decentralized oracles. Even if it is attested later on, the actual data itself is not guaranteed to be accurate. It might be acceptable for a smart contract designer in a use case to accept data for an oracle that is provided by a large, reputable, and trusted third party. For example, the source of the data can be from a reputable weather reporting agency or airport information system directly relaying the flight delays, which can give some level of confidence. However, the issue of centralization remains.

Types of blockchain oracles

There are various types of blockchain oracles, ranging from simple software oracles to complex hardware assisted and decentralized oracles. Broadly speaking, we can categorize oracles into two categories: **inbound oracles and outbound oracles**.

Inbound oracles: This class represents oracles that receive incoming data from external services, and feed it into the smart contract **Software oracles:** These oracles are responsible for acquiring information from online services on the Internet. This type of oracle is usually used to source data such as weather information, financial data (stock prices, for example), travel information and other types of data from third-party providers. The data source can also be an internal enterprise system, which may provide some enterprise specific data. These types of oracles can also be called **standard or simple oracles.**

Hardware oracles This type of oracle is used to source data from hardware sources such as IoT devices or sensors. This is useful in use cases such as insurance-related smart contracts where telemetry sensors provide certain information, for example, vehicle

speed and location. This information can be fed into the smart contract dealing with insurance claims and payouts to decide whether to accept a claim or not. Based on the information received from the source hardware sensors, the smart contract can decide whether to accept or reject the claim. These oracles are useful in any situation where real-world data from physical devices is required. However, this approach requires a mechanism in which hardware devices are tamper-proof or tamper-resistant. This level of security can be achieved by providing cryptographic evidence (non-repudiation and integrity) of IoT device's data and an anti-tampering mechanism on the IoT device, which renders the device useless in case of tampering attempts.

Computation oracles: These oracles allow computing-intensive calculations to be performed off-chain. As blockchain is not suitable for performing compute-intensive operations, a blockchain (that is, a smart contract on a blockchain) can request computations to be performed on off-chain high-performance computing infrastructure and get the verified results back via an oracle. The use of oracle, in this case, provides data integrity and authenticity guarantees. An example of such an oracle is Truebit (https://truebit.io). It allows a smart contract to submit computation tasks to oracles, which are eventually completed by miners in return for an incentive.

Aggregation based oracles: In this scenario, a single value is sourced from many different feeds. As an example, this single value can be the price of a financial instrument, and it can be risky to rely upon only one feed. To mitigate this problem, multiple data providers can be used where all of these feeds are inspected, and finally, the price value that is reported by most of the feeds can be picked up. The assumption here is that if the majority of the sources reports the same price value, then it is likely to be correct. The collation mechanism depends on the use case: sometimes it's merely an average of multiple values, sometimes a median is taken of all the values, and sometimes it is the maximum value. Regardless of the aggregation mechanism, the essential requirement here is to get the value that is valid and authentic, which eventually feeds into the system. An excellent example of price feed oracles is MakerDAO (https://makerdao.com/en/) price feed oracle (https://developer.makerdao.com/feeds/), which collates price data from multiple external price feed sources and provides a median ETHUSD price to MakerDAO. Crowd wisdom driven oracles This is another way that the blockchain oracle problem can be addressed where a single source is not trusted. Instead, multiple public sources are used to deduce the most appropriate data eventually. In other words, it solves the problem

where a single source of data may not be trustworthy or accurate as expected. If there is only one source of data, it can be unreliable and risky to rely on entirely. It may turn malicious or become genuinely faulty.

In this case, to ensure the credibility of data provided by third-party sources for oracles, the data is sourced from multiple sources. These sources can be users of the system or even members of the general public who have access to and have knowledge of some data, for example, a political event or a sporting event where members of the public know the results and can provide the required data. Similarly, this data can be sourced from multiple different news websites. This data can then be aggregated, and if a sufficiently high number of the same information is received from multiple sources, then there is an increased likelihood that the data is correct and can be trusted.

Decentralized oracles Another type of oracles, which primarily emerged due to the decentralization requirements, is called decentralized oracles. Remember that in all types of oracles discussed so far, there are some trust requirements to be placed in a trusted third party. As blockchain platforms such as Bitcoin and Ethereum are fully decentralized, it is expected that oracle services should also be decentralized. This way, we can address the Blockchain Oracle Problem. This type of oracle can be built based on a distributed mechanism. It can also be envisaged that the oracles can find themselves source data from another blockchain, which is driven by distributed consensus, thus ensuring the authenticity of data. For example, one institution running their private blockchain can publish their data feed via an oracle that can then be consumed by other blockchains. A decentralized oracle essentially allows off-chain information to be transferred to a blockchain without relying on a trusted third party.

Augur is a prime example of such type of oracles. The core idea behind Augur's oracle is that of crowd wisdom-supported oracles, in which the information about an event is acquired from multiple sources and aggregated into the most likely outcome. The sources in case of Augur are financially motivated reporters who are rewarded for correct reporting and penalized for incorrect reporting.

Smart oracles An idea of smart oracle has also been proposed by Ripple labs (codius). Smart oracles are entities just like oracles, but with the added capability of executing contract code. Smart oracles proposed by Codius run using Google Native Client, which is a sandboxed environment for running untrusted x86 native code.

Outbound oracles This type, also called **reverse oracles,** are used to send data out from the blockchain smart contracts to the outside world. There are two possible scenarios here; one is where the source blockchain is a producer of some data such as blockchain metrics, which are needed for some other blockchain. The actual data somehow needs to be sent out to another blockchain smart contract. The other scenario is that an external hardware device needs to perform some physical activity in response to a transaction on-chain. However, note that this type of scenario does not necessarily need an oracle, because the external hardware device can be sent a signal as a result of the smart contract event.

On the other hand, it can be argued that if the hardware device is running on an external blockchain, then to get data from the source chain to the target chain, undoubtedly, will need some security guarantees that oracle infrastructure can provide. Another situation is where we need to integrate legacy enterprise systems with the blockchain. In that case, the outbound oracle would be able to provide blockchain data to the existing legacy systems. An example scenario is the settlement of a trade done on a blockchain that needs to be reported to the legacy settlement and backend reporting systems. Now that we have discussed different types of oracles, we now introduce different service providers that provide these services. Several service providers provide oracle services for blockchain, some of these we introduce following.

Blockchain oracle services

Various online services are now available that provide oracle services. These can also be called **oracle-as-a-service** platforms.

All of these services aim to enable a smart contract to securely acquire the off-chain data it needs to execute and make decisions:

- Town Crier: https://www.town-crier.org
- Provable: https://provable.xyz
- Witnet: https://witnet.io
- Chainlink: https://chain.link
- The Realitio project: https://realit.io
- TrueBit: https://truebit.io
- iExec: https://iex.ec Another service at https://smartcontract.com/ is also available, where Ethereum, Bitcoin, and Town Crier oracles can be created.

It allows smart contracts to connect to applications and allows you to feed data into the smart contracts from off-chain sources.

There are many oracle services available now, and it is challenging to cover all of them here. A random selection is presented in the preceding list.

Deploying smart contracts

Smart contracts may or may not be deployed on a blockchain, but it makes sense to do so on a blockchain due to the security and decentralized consensus mechanism provided by the blockchain. Ethereum is an example of a blockchain platform that natively supports the development and deployment of smart contracts. Smart contracts on an Ethereum blockchain are typically part of a broader DApp.

In comparison, in a Bitcoin blockchain, the transaction timelocks, such as the nLocktime field, the CHECKLOCKTIMEVERIFY (CLTV), and the CHECKSEQUENCEVERIFY script operator in the Bitcoin transaction, can be seen as an enabler of a simple version of a smart contract. These timelocks enable a transaction to be locked until a specified time or until 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) are met. For example, you can implement conditions such as Pay party X, N number of bitcoins after 3 months. However, this is very limited and should only be viewed as an example of a basic smart contract. In addition to the example mentioned earlier, Bitcoin scripting language, though limited, can be used to construct basic smart contracts.

One example of a basic smart contract is to fund a Bitcoin address that can be spent by anyone who demonstrates a hash collision attack. This was a contest that was announced on the Bitcointalk forum where bitcoins were set as a reward for whoever manages to find hash collisions for hash functions. This conditional unlocking of Bitcoin solely on the demonstration of a successful attack is a basic type of smart contract.

The DAO

The Decentralized Autonomous Organization (DAO), started in April 2016, was a smart contract written to provide a platform for investment. Due to a bug, called the re-entrancy bug, in the code, it was hacked in June 2016. An equivalent of approximately 3.6 million ether (roughly 50 million US dollars) was siphoned out of the DAO into another account. Even though the term hacked is used here, it was not really hacked. The smart contract did what it was asked to do but due to the vulnerabilities in the smart contracts, the attacker was able to exploit it. It can be seen as an unintentional behavior (a bug) that programmers of the DAO did not foresee. This incident resulted in a hard fork on the Ethereum blockchain, which was introduced to recover from the attack. The DAO attack exploited a vulnerability (re-entrancy bug) in the DAO code where it was

possible to withdraw tokens from the DAO smart contract repeatedly before giving the DAO contract a chance to update its internal state to indicate that how many DAO tokens have been withdrawn. The attacker was able to withdraw DAOs. However, before the smart contract could update its state, the attacker withdrew the tokens again. This process was repeated many times, but eventually, only a single withdrawal was logged by the smart contract, and the contract also lost record of any repeated withdrawals.

The DAO attack highlights the dangers of not formally and thoroughly testing smart contracts. It also highlights the absolute need to develop a formal language for the development and verification of smart contracts. The attack also highlighted the importance of thorough testing to avoid the issues that the DAO experienced. There have been various vulnerabilities discovered in Ethereum over the last few years regarding 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, for example, an online service at https://securify.ch, which provides tools to formally verify smart contract code.

- Another example is **Michelson** (https://www.michelson.org) for writing smart contracts in the Tezos blockchain. It is a functional programming language suitable for formal verification.
- **Vyper** is another language that aims to provide a secure language for developing smart contracts for EVM. It aims for goals such as security, simplicity, and audibility. It is a strongly typed language with support for overflow checking and signed integers. All these features make Vyper a reasonable choice for writing secure smart contracts. Even though many different initiatives are aiming to explore and address the security of smart contracts, this field still requires further research to address limitations in smart contract programming languages.