

Blockchain Technology

(BCS613A)

MODULE 3

- Bitcoin was the first application of blockchain technology, and has started a revolution with the introduction of the very first fully decentralized digital currency.
- It has proven to be remarkably secure and valuable as a digital currency; however, it is quite unstable and highly volatile.
- Bitcoin solves a number of historically difficult problems related to electronic cash and distributed systems, including:
 1. The Byzantine generals problem
 2. The double-spending problem
 3. Sybil attacks

Bitcoin definition

- Bitcoin can be defined in various ways; it's a protocol, a digital currency, and a platform.
- It is a combination of a peer-to-peer network, protocols, and software that facilitates the creation and usage of the digital currency.
- Nodes in this peer-to-peer network talk to each other using the Bitcoin protocol.

Bitcoin—A user's perspective

Main components of a Bitcoin network are:

1. Digital keys
2. Addresses
3. Transactions
4. Blockchain
5. Miners
6. The Bitcoin network
7. Wallets (client software)

Sending a payment

There are several steps that are involved in this process. For example, using the Blockchain wallet for mobile devices.

1. The sender can also initiate a transfer to send money to another user. In both cases, the address of the beneficiary is required.



2. The sender either enters the receiver's address or scans the QR code that has the Bitcoin address, amount, and an optional description encoded in it. The wallet application recognizes this QR code and decodes it into something like:

"Please send <amount> BTC to address <receiver's Bitcoin address>".

3. With actual values, this will look like the following:

"Please send 0.00033324 BTC to address 1JzouJCVmMQBmTcd8K4Y5BP36gEFNn1ZJ3"

4. This is also shown in the following screenshot:



5. This transaction is digitally signed using the private key of the sender before broadcasting it. Once the QR code is decoded, the transaction will appear as follows:

Bitcoin

Ether

From

My Bitcoin Wallet

To

1JzouJCVmMQBmTcd8K4Y5BP36gEF...

BTC

0.00033324

GBP

1.53

Use total available minus fee:

0.00251933 BTC

Fee

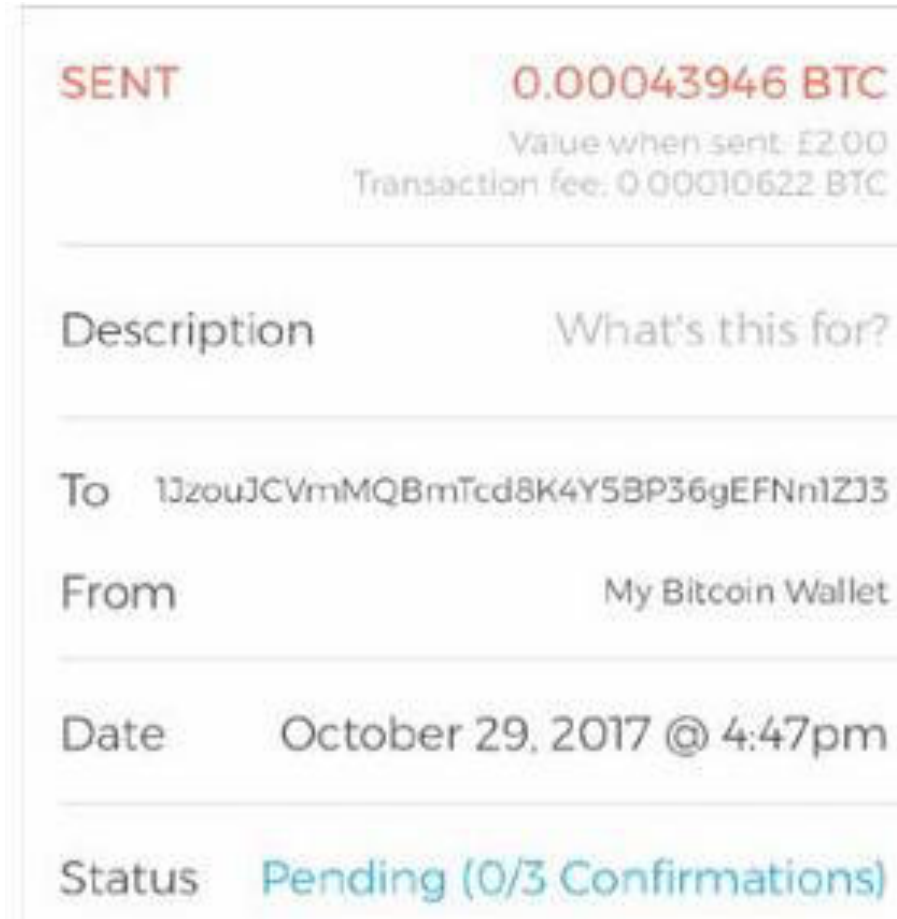
Regular
1+ hour

0.00010622 BTC (£0.49)

>

Continue

Once the transaction is sent, it will appear in the wallet software as shown in the following screenshot:



6. At this stage, the transaction has been constructed, signed, and sent out to the Bitcoin network. This transaction will be picked up by miners to be verified and included in the block. The appropriate fee will be deducted from the original value to be transferred and will be paid to the miner who has included it in the block for mining.

A summary view of various attributes of the transaction is shown in the following screenshot:

- Size: This is the size of the transaction in bytes.
- Weight: This is the new metric given for the block and transaction sizes since the introduction of the segregated witness (SegWit) soft fork version of Bitcoin.
- Received time: This is the time when the transaction was received.
- Included in blocks: This shows the block number on the blockchain in which the transaction is included.
- Confirmations: This is the number of confirmations completed by miners for this transaction.
- Total input: This is the number of total inputs in the transaction.

- Total output: This is the number of total outputs from the transaction.
- Fees: This is the total fee charged.
- Fee per byte: This field represents the total fee divided by the number of bytes in the transaction; for example, 10 Satoshis per byte.
- Fee per weight unit: For legacy transactions, this is calculated using the total number of bytes * 4. For SegWit transactions, it is calculated by combining a SegWit marker, flag, and witness field as one weight unit, and each byte of the other fields as four weight units

In summary, a payment transaction can be divided into the following steps:

1. The transaction starts with a sender signing the transaction with their private key.
2. The transaction is serialized so that it can be transmitted over the network.
3. The transaction is broadcast to the network.
4. Miners listening for transactions pick up the transaction.
5. The transaction is verified for its legitimacy by the miners.
6. The transaction is added to the candidate/proposed block for mining.
7. Once mined, the result is broadcast to all nodes on the Bitcoin network.
8. Usually, at this point, users wait for up to six confirmations to be received before a transaction is considered final; Confirmations serve as an additional mechanism to ensure that there is probabilistically a very low chance for a transaction to be reverted.

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.
- 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

The lifecycle of a Bitcoin transaction:

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.

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. 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.

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:

$$\text{fee} = \text{sum}(\text{inputs}) - \text{sum}(\text{outputs})$$

- 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.
- 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.
- 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.
- A transaction on the Bitcoin network is represented by a data structure that consists of several fields.

The transaction data structure

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

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.

There are a number of structures that make up the transaction.

- Metadata: This part of the transaction contains 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. A UTXO is an unspent transaction output that can be spent as an input to a new transaction.

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 (<code>ScriptSig</code>), which satisfies the requirements of the locking script
Sequence number	4 bytes	Usually disabled or contains lock time – disabled is represented by <code>0xFFFFFFFF</code>

Outputs: Outputs have three fields, and they contain instructions for sending bitcoins. The first field contains the amount of Satoshis, whereas the second field contains the size of the locking script. Finally, the third field contains a locking script that holds 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)

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.
- 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.
- 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.

- 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

Commonly used opcodes:

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 afterward.
OP_CHECKMULTISIG	This instruction 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.
OP_HASH256	The input is hashed twice with SHA-256.
OP_MAX	This returns the larger value of two inputs.

Types of scripts

- There are various standard scripts available in Bitcoin to handle the verification and 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 transactions are evaluated using the `IsStandard()` and `IsStandardTx()` tests and only those transactions that pass the test are allowed to be broadcasted or mined on the Bitcoin network.
- However, nonstandard transactions are also allowed on the network, as long as they pass the validity checks.

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

```
ScriptPubKey: OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG  
ScriptSig: <sig> <pubKey>
```

Pay-to-Script Hash (P2SH): P2SH is used in order to send transactions to a script hash (that is, 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): The M of N multisignature transaction script is a complex type of script where it is possible to construct a script that requires 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>]
```

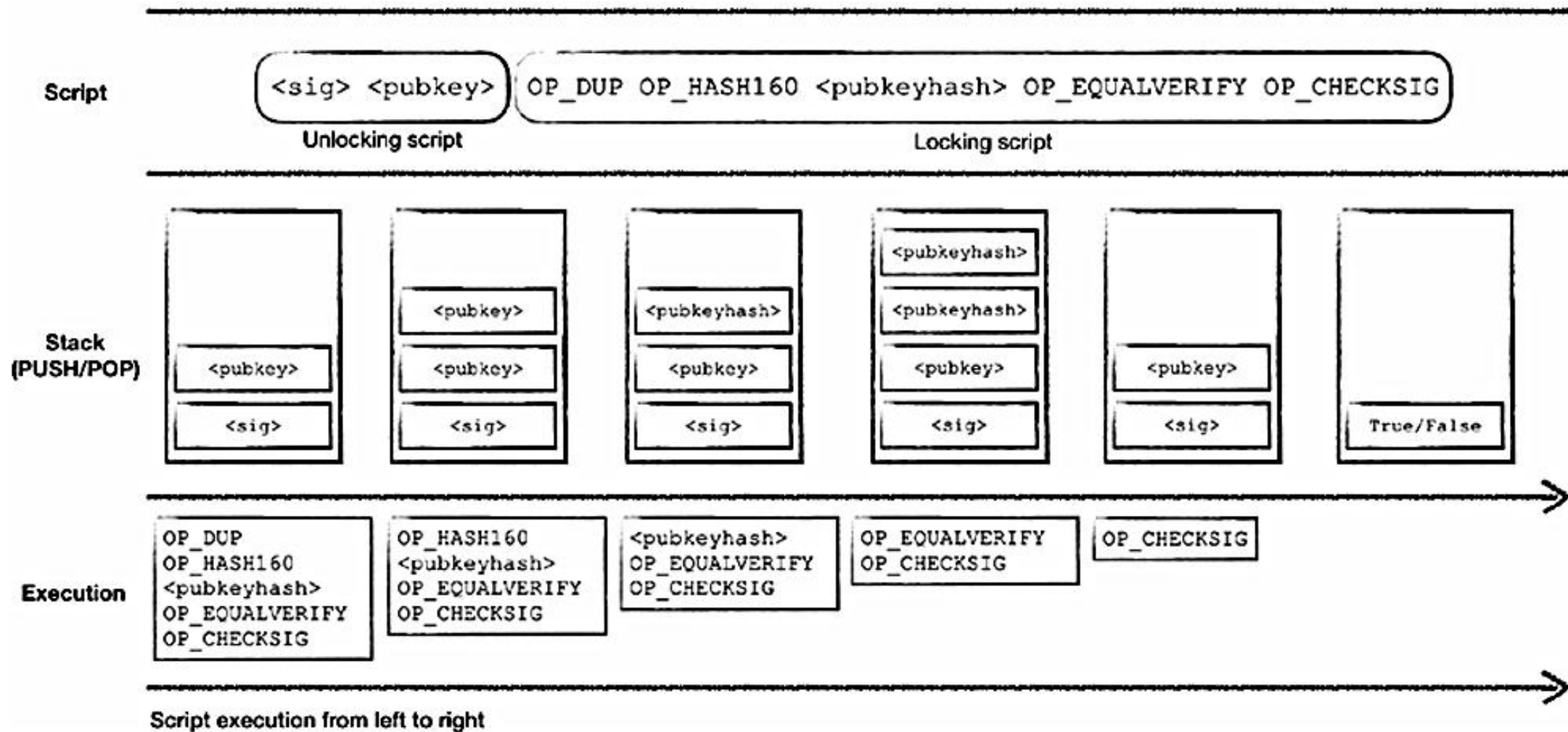
Pay to Pubkey: This script is a very simple script that is commonly used in coinbase transactions. It is now obsolete and was used in an old version of Bitcoin. The public key is stored within the script in this case, and the unlocking script is required to sign the transaction with the private key.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

Commonly used opcodes:



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. It 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.

Contracts

- Contracts are Bitcoin scripts that use the Bitcoin blockchain to enforce a financial agreement. It allows users to programmatically create complex contracts that can be used in many real-world scenarios.
- Contracts allow the development of completely decentralized, independent, and reduced risk platforms by programmatically enforcing different conditions for unlocking the money (bitcoins).

- Various contracts, such as escrow, arbitration, and micropayment channels, can be built using the Bitcoin scripting language.
- Even though the scripting language in Bitcoin can be used to create complex contracts, it is quite limited and is not at all on par with the smart contracts, which are Turing complete constructs and allow arbitrary program development.

Miniscript

- This language allows for a more structured approach to writing Bitcoin scripts.
- Even though Bitcoin Script supports combinations of different spending conditions, such as time and hash locks, it is not easy to analyze existing scripts or build new complex scripts.
- Miniscript makes it easier to write complex spending rules. It also makes it easier to ascertain the correctness of the script.

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 the coinbase, which acts as an input to the coinbase transaction. This transaction also allows up to 100 bytes of arbitrary data storage.
- A coinbase transaction input has the same number of fields as a usual transaction input, but the structure contains the coinbase data size and fields instead of the unlocking script size and fields.
- Also, it does not have a reference pointer to the previous transaction.

Field	Size	Description
Transaction hash	32 bytes	Set to all zeroes as no hash reference is used
Output index	4 bytes	Set to 0xFFFFFFFF
Coinbase data length	1-9 bytes	2-100 bytes
Data	Variable	Any data
Sequence number	4 bytes	Set to 0xFFFFFFFF

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 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.

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. 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.

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

- From Bitcoin's perspective, the blockchain can be defined as a public, distributed ledger holding a timestamped, ordered, and immutable record of all transactions on the Bitcoin network.
- Transactions are picked up by miners and bundled into blocks for mining. Each block is identified by a hash and is linked to its previous block by referencing the previous block's hash in its header.
- This linking makes sure that no transaction can be modified unless the block that records it and all blocks that follow it are also modified.
- The first block is not linked to any previous block and is known as the genesis block:

The data structure of a Bitcoin block

Field	Size	Description
Block size	4 bytes	The size of the block.
Block header	80 bytes	This includes fields from the block header described in the next section.
Transaction counter	Variable	The field contains the total number of transactions in the block, including the coinbase transaction. Size ranges from 1-9 bytes.
Transactions	Variable	All transactions in the block.

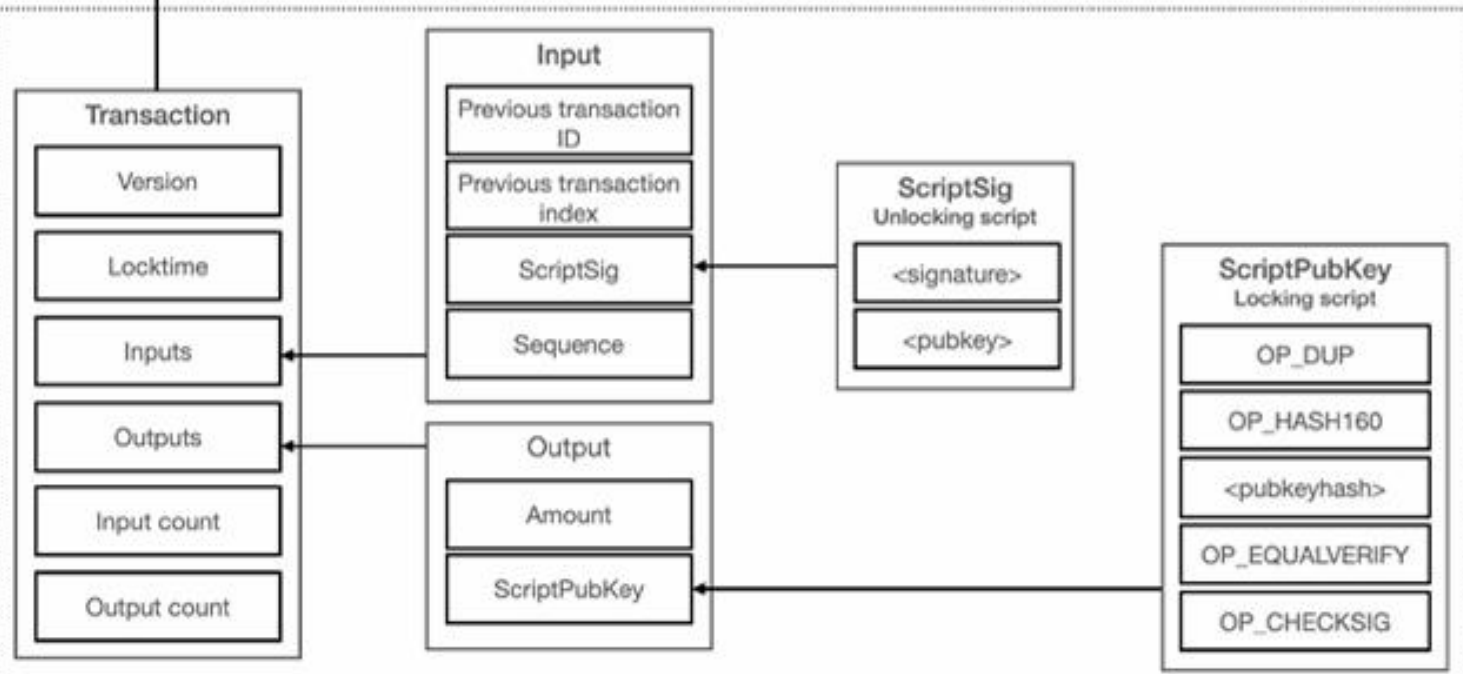
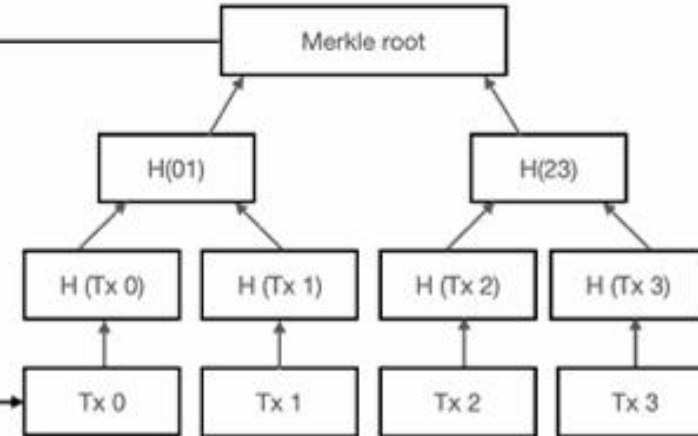
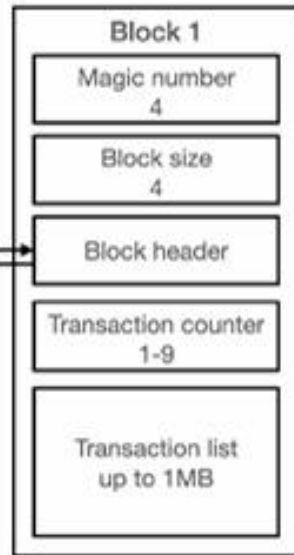
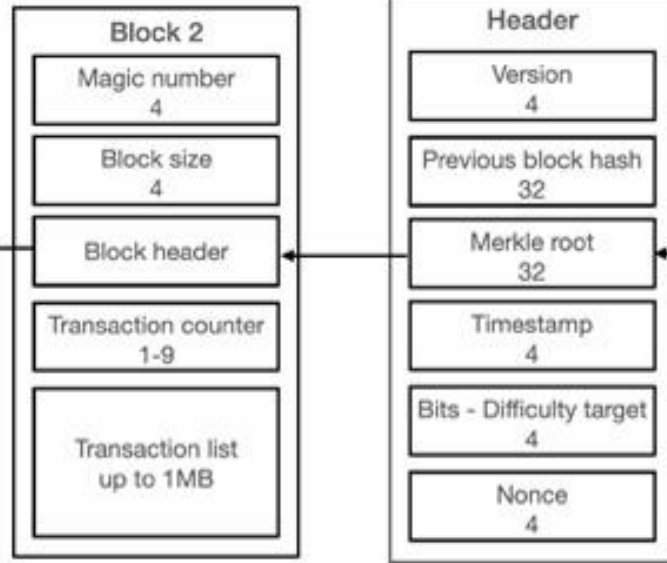
The block header

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.

Blockchain height

A visualization of blockchain, block, block header, transactions, and scripts

Previous block link



- 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.
- Further down the diagram, transactions are also magnified to show the structure of a transaction and the elements that it contains.
- Transactions are then further elaborated to show what locking and unlocking scripts look like. The size (in bytes) of each field of block, header and transaction is also shown as a number under the name of the field.

The genesis block

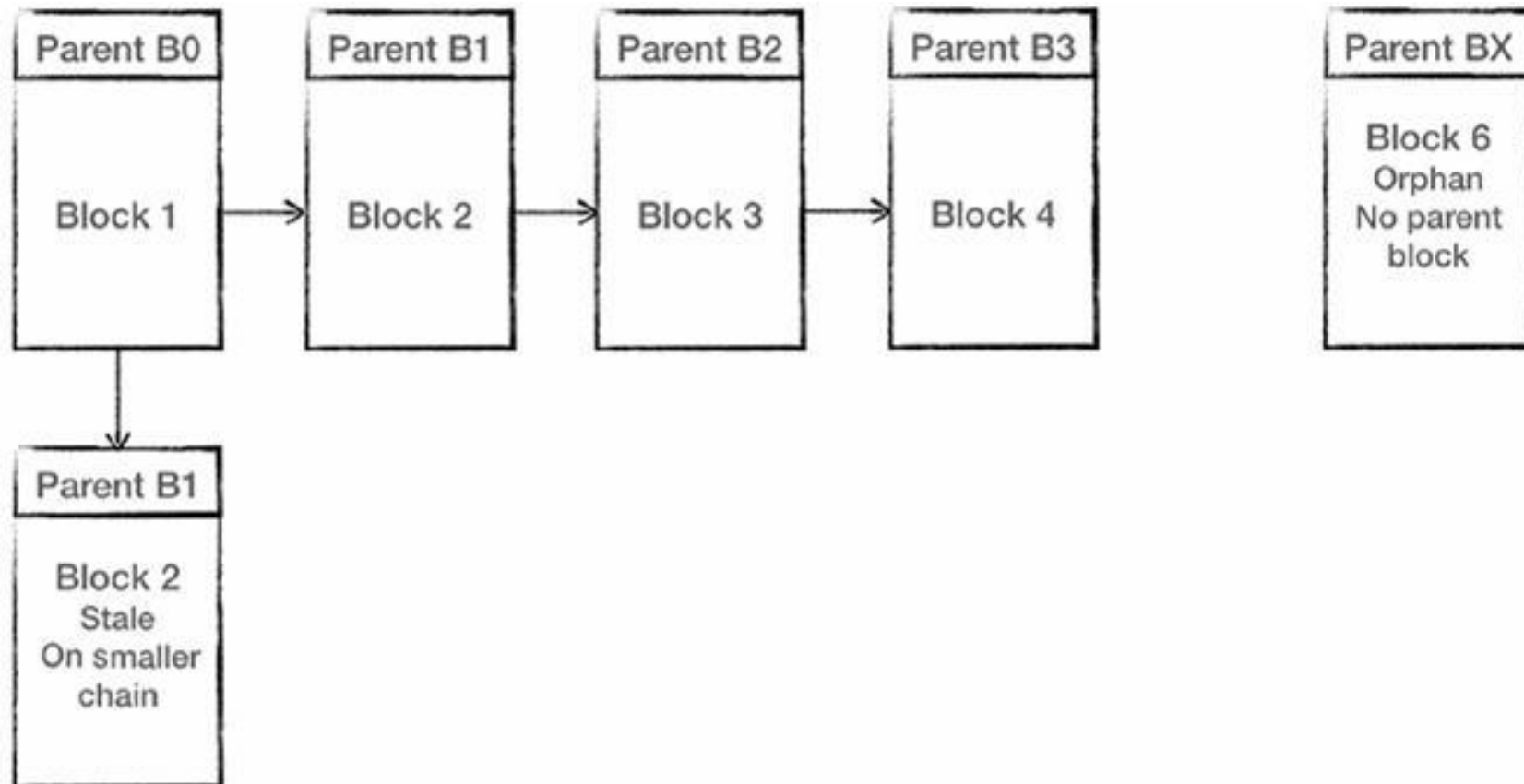
- This is the first block in the Bitcoin blockchain. The genesis block was hardcoded in the Bitcoin core software.
- The first Bitcoin block (genesis block) was not mined earlier than January 3rd, 2009.
- Bitcoin provides protection against double-spending by enforcing strict rules on transaction verification and via mining.
- Transactions and blocks are added to the blockchain only after the strict rule-checking 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.
- For example, a miner found a valid block, but the receiving node did not have the parent block (due to network delays).
- As their parents are unknown, they cannot be validated.
- As a result, the node temporarily marked the block as an orphan until it received the missing parent.
- Similar to stale blocks, they do not provide any reward to their miners.



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

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

- 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.
- As Bitcoin evolves and new upgrades and innovations are introduced in it, the version associated with blocks also changes.
- These versions introduce various security parameters and new features.

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.

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:

$$\text{Target} = \text{Previous target} * \text{Time}/2016 * 10 \text{ minutes}$$

- 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.

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.

- Some nodes are full blockchain nodes with a complete blockchain as they are more secure and play a vital role in block propagation, while some nodes perform network routing functions only but do not perform mining or store private keys (the wallet function).
- 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.

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

- A full Bitcoin node performs four functions. These are wallet, miner, blockchain, and network routing.
- There are 27 types of protocol messages in total. The most commonly used protocol messages are:

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.

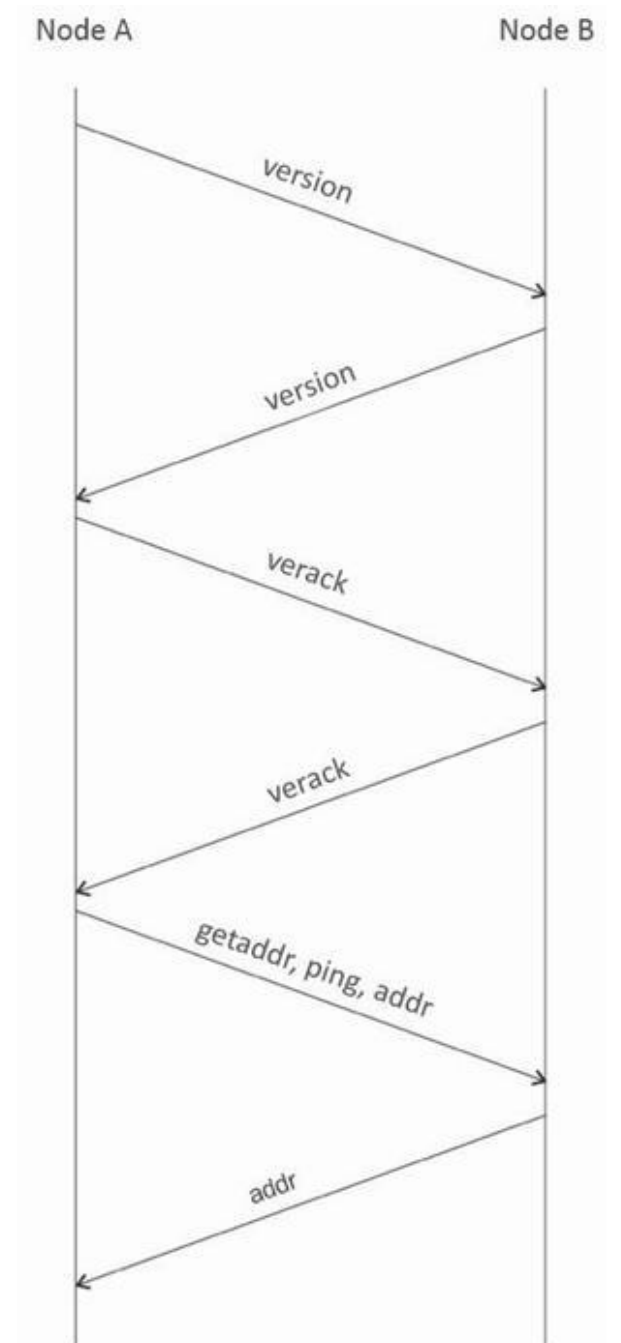
Ping: This message is used to confirm if the TCP/IP network connection is active.

Pong: This message is the response to a ping message confirming that the network connection is live.

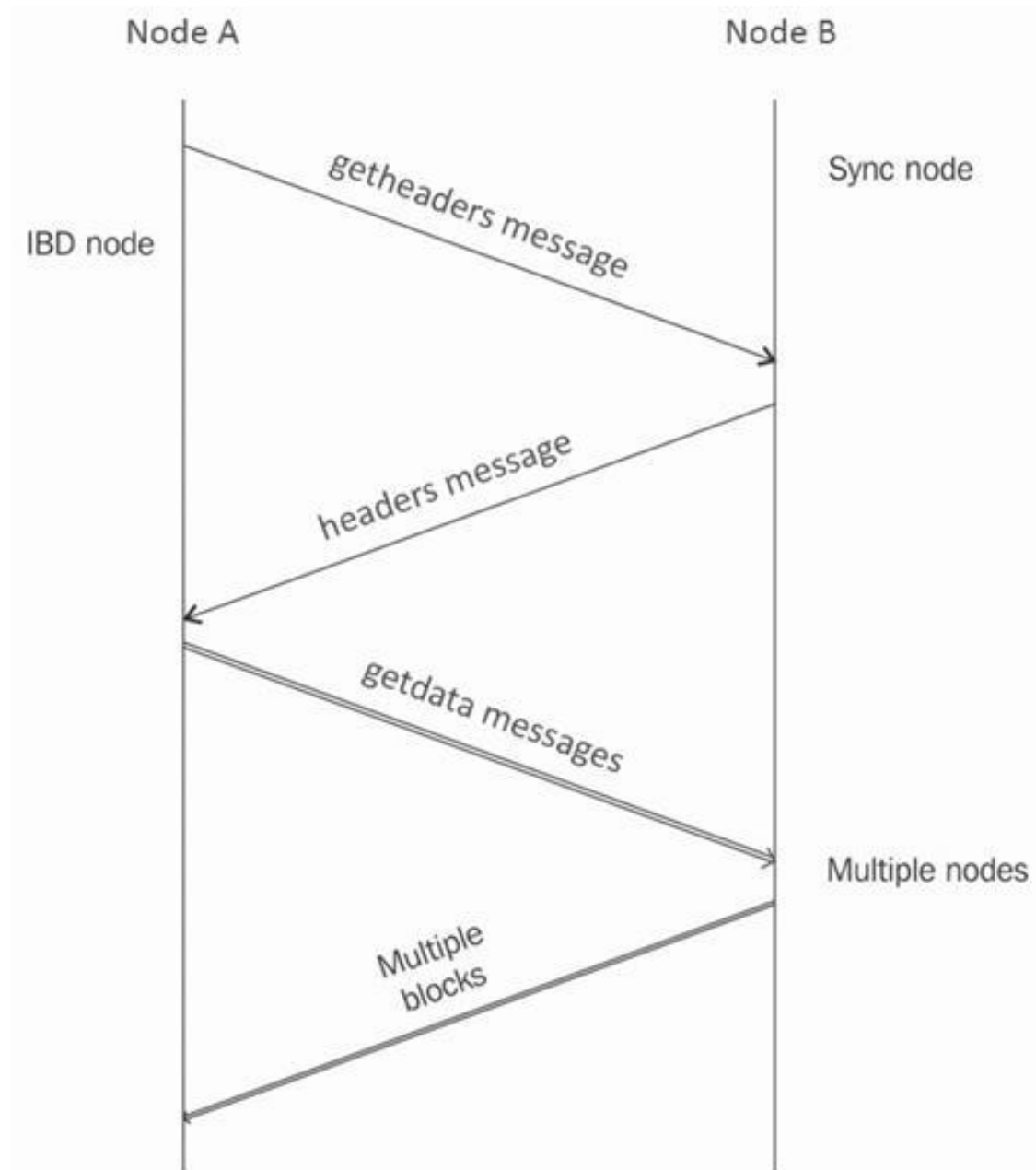
- When a Bitcoin Core node starts up, first, it initiates the discovery of all peers.
- This is achieved by DNS query. This lookup returns a number of DNS A records.
- First, the client sends a protocol message, version, which contains various fields, such as version, services, timestamp, network address, 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.

Visualization of node discovery protocol

- 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.
- 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.



- An Remote Procedure Call (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.

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 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.

Bloom filters

- A bloom filter is a data structure 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.

- 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.
- 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:

1. filterload: This is used to set the bloom filter on the connection.
2. filteradd: This adds a new data element to the current filter.
3. 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.
- 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

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.
- These phrases can be used to recover all keys and make private key management comparatively easier

Hierarchical deterministic wallets

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

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

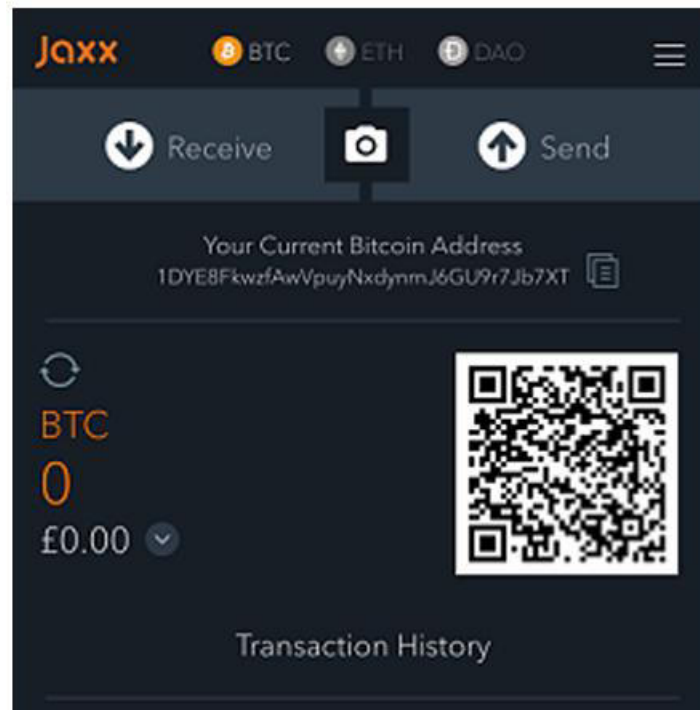
- Online wallets, as the name implies, are stored entirely online and are provided as a service usually via the cloud.
- They provide a web interface to the users to manage their wallets and perform various functions, such as making and receiving payments.
- They are easy to use but imply that the user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

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.

Smart Contracts

History

- Nick Szabo first theorized smart contracts in the 1990s, in an article called Formalizing and Securing Relationships on Public Networks.
- This theory was presented almost 20 years before the real potential and benefits of smart contracts were appreciated, that is, before the invention of Bitcoin and the subsequent development of other more advanced blockchain platforms, such as Ethereum.

- Smart contract functionality was implemented in a limited fashion in Bitcoin in 2009.
- Bitcoin supports a restricted scripting language called script, which allows the transfer of bitcoins between users.
- However, this is not a Turing complete language and does not support arbitrary program development.
- It can be regarded as a limited function calculator with only simple arithmetic operations, whereas smart contracts can be considered general-purpose computers that support writing any program.

Definition

- A smart contract is a secure and unstoppable computer program representing an agreement that is automatically executable and enforceable.
- It is 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.
- smart contracts are automatically executed according to the instruction that is coded in.
- They are enforceable, which means that all contractual terms perform as specified and expected, even in the presence of adversaries.
- 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.
- These computer programs are fault-tolerant and executable in a reasonable (finite) amount of time.
- Smart contracts usually operate by managing their internal state using a state machine model provided by the underlying blockchain.
- The legality of smart contracts is uncertain in many jurisdictions. If humans and machines can both understand the code written in a smart contract, it might become acceptable in legal situations.
- Smart contracts do what they have been programmed to do, thus smart contracts produce the same output every time they are executed. The deterministic nature of smart contracts is highly desirable in blockchain platforms due to consistency requirements.

In summary, 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 smart contract programming language and the smart contract code must be correct, valid, and verified.
- **Deterministic:** The deterministic feature ensures that smart contracts always produce the same output for a specific input.

- **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

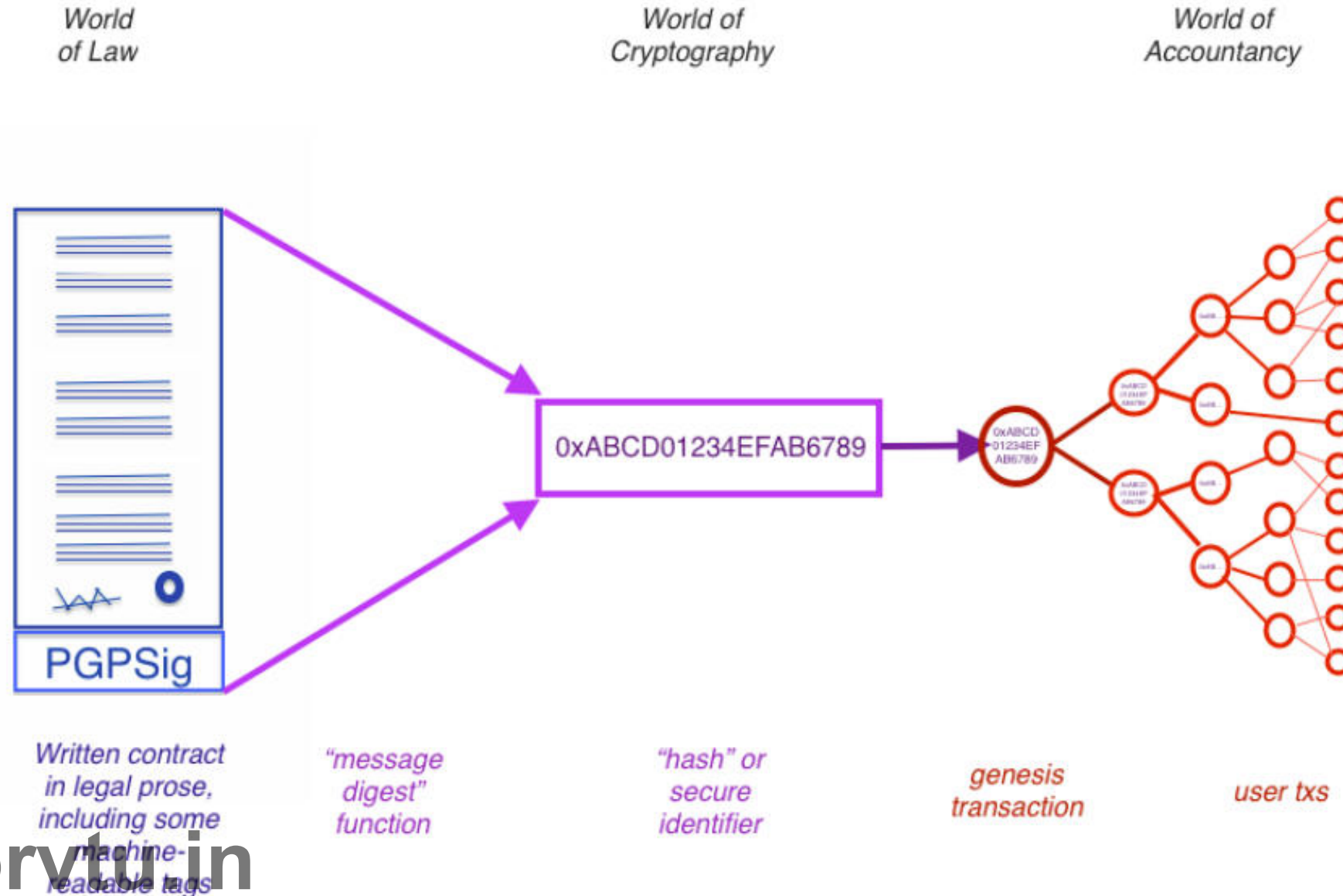
- Ricardian contracts were initially proposed in the paper, Financial Cryptography in 7 Layers, by Ian Grigg, in the late 1990s.
- The fundamental idea behind this contract is to write a document that is understood and accepted by both a court of law and computer software.
- Ricardian contracts address the challenge of the issuance of value over the internet.

A Ricardian contract is a document that has several of the following properties:

1. It is a contract offered by an issuer to holders
2. It is a valuable right held by holders and managed by the issuer
3. It can be easily read by people (like a contract on paper)
4. It can be read by programs (parsable, like a database)
5. It is digitally signed
6. It carries the keys and server information
7. It is allied with a unique and secure identifier

- In practice, the contracts contains the terms of the contract in legal prose and the required machine-readable tags.
- This document is digitally signed by the issuer using their private key.
- This document is then hashed using a message digest function to produce a hash by which the document can be identified.
- This hash is then further signed by parties therefore serving as evidence of intent.

The Ricardian contract bowtie model



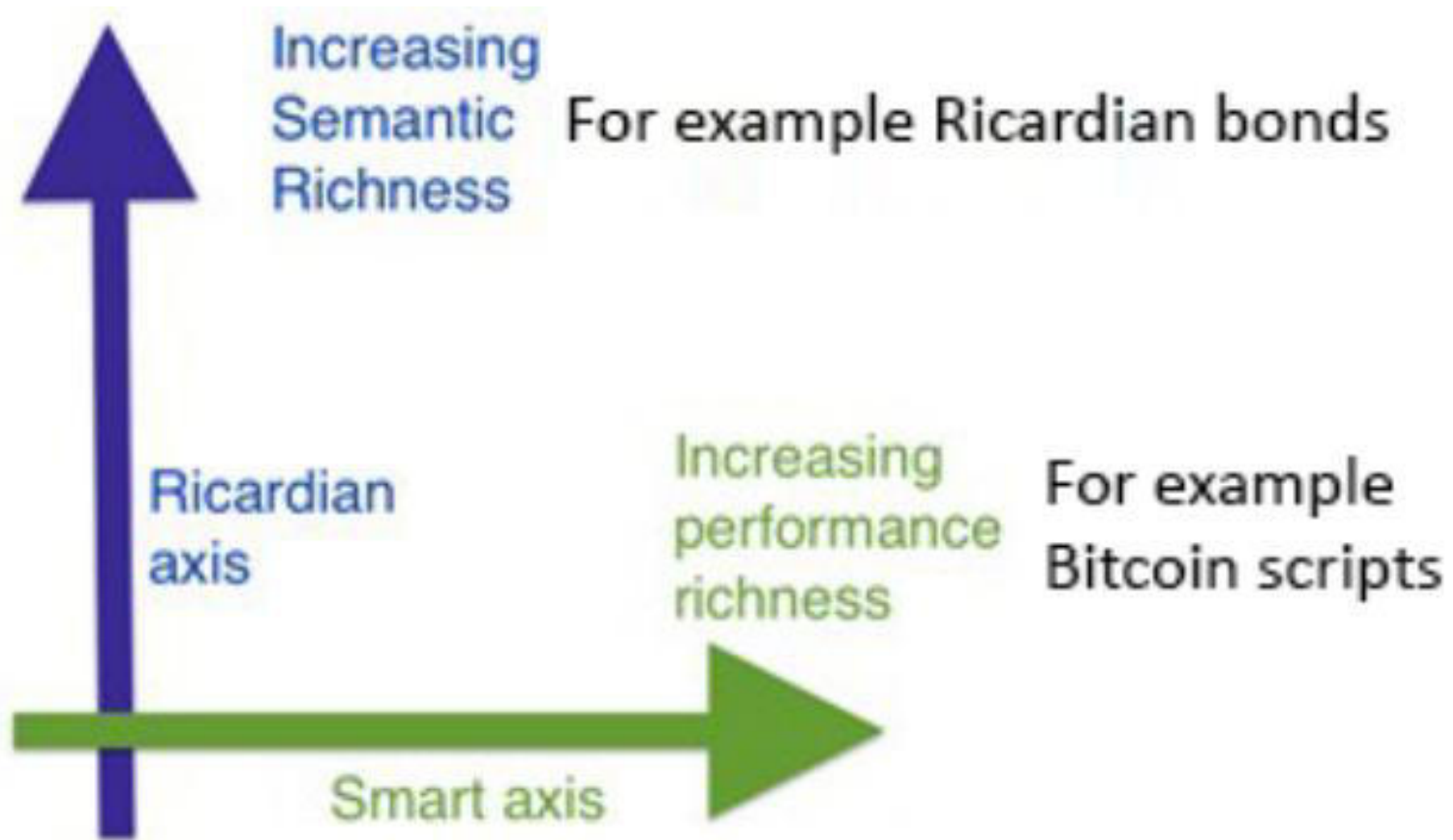
The diagram shows a number of elements:

1. 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.
2. This document is then hashed.
3. 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.
- Genesis transaction, or first transaction is used 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.

- In Bitcoin, a straightforward implementation of basic smart contracts can be observed, which is entirely oriented toward the execution and performance of the contract.
- 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.



The diagram shows that Ricardian contracts are more semantically-rich, whereas smart contracts are more performance-rich.

- 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.
- Domain-specific languages (DSLs) should be built to support the design and implementation of smart contract templates.
- 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.

- 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.
- 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.

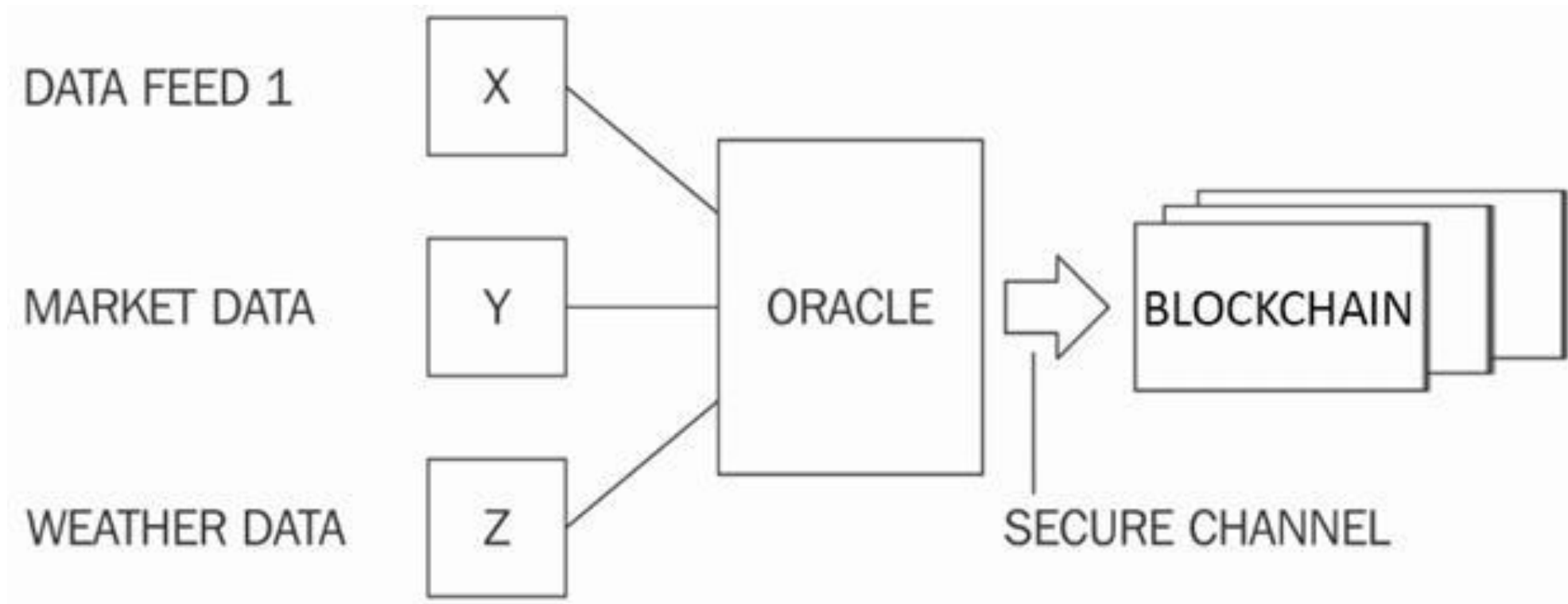
- This idea of DSLs for smart contract programming can be further extended to a GPL.
- A smart contract modeling platform can be developed where a domain expert can use a graphical user interface and a canvas (drawing area) to define and illustrate the definition and execution of a financial contract.
- 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.

- 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 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.
- 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.



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.

Some of the common use cases of oracles:

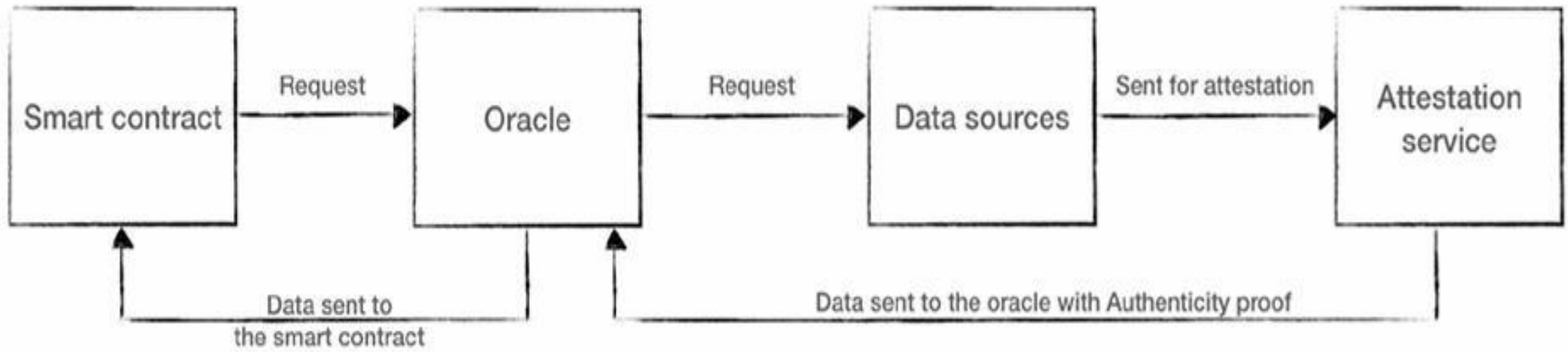
Type of data	Examples	Use case
Market data	Live price feeds of financial instruments. Exchange rates, performance, pricing, and historic data of commodities, indices, equities, bonds, and currencies.	DApps related to financial services, for example, decentralized exchanges and decentralized finance (DeFi)
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

- 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:

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, 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 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.

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.



A generic oracle data flow

- 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 (<https://tlsnotary.org/pagesigner.html>) to provide web page notarization.
- This mechanism can also be used to provide the required security services to oracles.

1. 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 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.
- 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.

2. TLS-N based mechanism

- TLS-N is a TLS extension that provides secure non-repudiation guarantees.
- This protocol allows 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

These proofs rely on some hardware elements to provide proof of authenticity. They require specific hardware to work.

1. 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 the data is authentic.

2. 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.

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

3. 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 `towncrier` service, which allows to run `towncrier` 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.

Types of blockchain oracles

- There are various types of blockchain oracles, ranging from simple software oracles to complex hardware assisted and decentralized oracles.
- 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.

Types: Software, Hardware, and several other types of inbound oracle.

1. 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, 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 oracle can also be called standard or simple oracles.

2. 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.

- 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 of IoT device's data and an anti-tampering mechanism on the IoT device, which renders the device useless in case of tampering attempts.

3. 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 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 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.

4. Aggregation based oracles

- A single value is sourced from many different feeds. 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.
- 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, which collates price data from multiple external price feed sources and provides a median ETHUSD price to MakerDAO.

5. Crowd wisdom driven oracles

- This method is used to address the problem where a single source is not trusted. Instead, multiple public sources are used to deduce the most appropriate data eventually.
- 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.
- 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.

6. Decentralized oracles

- This type of oracles primarily emerged due to the decentralization requirements, is called decentralized oracles.
- 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.
- These 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 decentralized prediction market platform built on Ethereum. Augur uses a “crowdsourced truth” model, where users stake tokens (REP) to report on the outcome of an event.
- The sources in case of Augur are financially motivated reporters who are rewarded for correct reporting and penalized for incorrect reporting.

7. Smart oracles

- 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 other scenario is that an external hardware device needs to perform some physical activity in response to a transaction on-chain.
- 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.

- If the hardware device is running on an external blockchain, then to get data from the source chain to the target chain it 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.

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/>, where Ethereum, Bitcoin, and Town Crier oracles can be created.
- It allows smart contracts to connect to applications and allows to feed data into the smart contracts from off-chain sources.

Deploying smart contracts

- 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.
- 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.

- Various other blockchain platforms support smart contracts such as Monax, Lisk, Counterparty, Stellar, Hyperledger Fabric, Axoni core, Neo, EOSIO, and Tezos.
- Smart contracts can be developed in various languages, either DSLs or general-purpose languages.
- The critical requirement, however, is determinism, which is very important because it is vital that regardless of where the smart contract code executes, it produces the same result every time and everywhere.
- This requirement of the deterministic nature of smart contracts also implies that smart contract code is absolutely bug-free.
- Validation and verification of smart contracts is an active area of research.

- Various languages have been developed to build smart contracts such as Solidity, which runs on Ethereum Virtual Machine (EVM).
- There are platforms that already support mainstream languages for smart contract development, such as Lisk, which supports JavaScript.
- Another prominent example is Hyperledger Fabric, which supports Golang, Java, and JavaScript for smart contract development.
- A more recent example is EOSIO, which supports writing smart contracts in C++.

- Security is of paramount importance for smart contracts. However, there are many vulnerabilities discovered in prevalent blockchain platforms and relevant smart contract development languages.
- These vulnerabilities result in some high-profile incidents, such as the DAO attack.

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 reentrancy 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 (reentrancy 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.

- Though this hard fork was introduced for genuine reasons, it goes against the true spirit of decentralization, immutability, and the notion that code is the law.
- Subsequently, resistance against this hard fork resulted in the creation of Ethereum Classic, where a large number of users decided to keep mining on the old chain.
- This chain is the original, non-forked Ethereum blockchain that still contains the DAO. It can be said that on this chain, the code is still the law.

- 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.
- 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 auditability. 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.