

Minima

Technical Reference



Table of Contents

At a Glance

Network Overview

Layer 1: Minima

Maxima

Layer 2: Omnia

Layer 3: MiniDapps

Minima

Core Concepts

Transactions

MMR Database

TxPoW Units & Blocks

The Blockchain

Mining & Consensus

Coloured Coins

Keys & Digital Signatures

Scripting

Quantum Security

Configuration

Maxima

About

Maxima Contacts

Maxima Messaging

Maxima Location Service

FAQ

MiniDapps

About

At a Glance

Below you will find key information about Minima, with links to further reading.

Protocol

Language: Java

Consensus: TxPoW - Transaction Proof-of-Work

Longest chain: GHOST Protocol (heaviest chain)

Hashing algorithm: SHA3, considered to be Quantum resistant

Transactions/Blocks

Model: UTxO - Unspent Transaction Output (same as Bitcoin)

Txns/block: 256

Txns/second: ~5 on layer 1, unlimited on layer 2

Block intervals: 50 seconds

Currency

Native currency: Minima coin

Supply: 1 billion hard-capped, deflationary via [the Burn](#)

Custom tokens: supported natively via coloured coins, minted directly on node

NFTs: supported natively via coloured coins, minted directly on node

Smart Contracts

Scripting language: KISS VM is a custom scripting language for writing Smart Contracts on Minima

Scaling Solution

Layer 2: Minima supports off-chain payment channels, similar to Bitcoin's Lightning network, for virtually unlimited transactions per second

P2P Communication

Maxima: Off-chain information transport protocol layer to support peer-to-peer messaging

Web3 DApps

MiniDapps: Supports native web applications using HTML, CSS, JavaScript, React etc

Network Overview

The Minima network consists of four distinct layers.

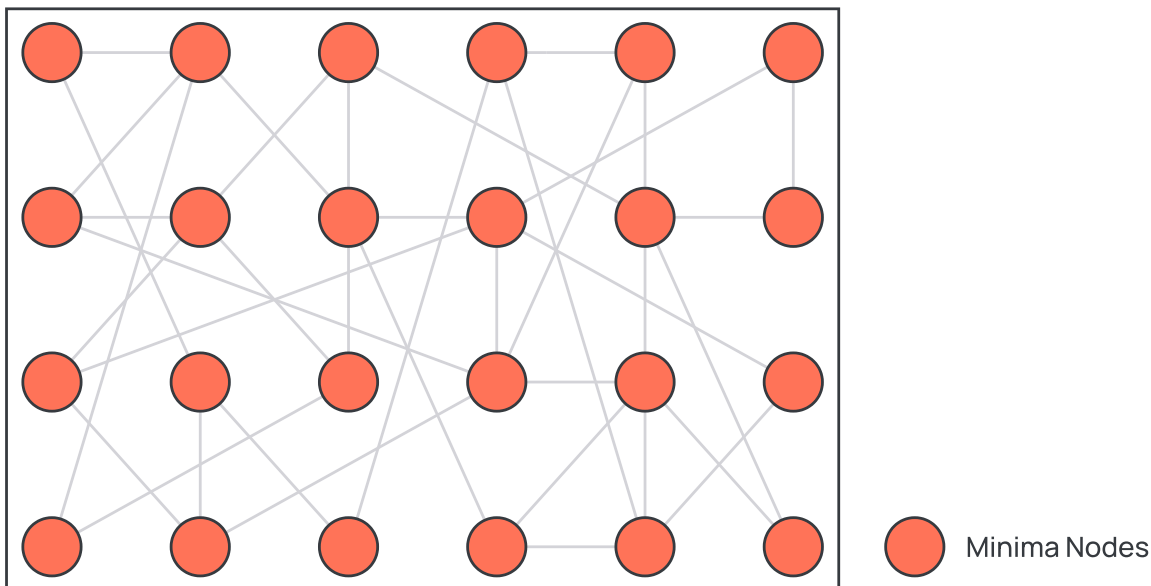
Each layer contributes to Minima being a truly decentralized, censorship resistant, peer-to-peer network for value and information exchange.

Layer 1: Minima

Value transfer

On-chain

Minima is the blockchain layer for value transfer. All transactions are processed by all nodes on the network. It is flood-fill. It uses the peer-to-peer network as its backbone for communication between nodes.



The Minima blockchain is where all on-chain transactions are processed. Every node in the network collectively comes to consensus on the state of the blockchain so all transactions are accounted for. Users initiate their transacting relationships on Layer 1, prior to moving off-chain to use Layer 2 for faster and cheaper transactions. As the trust layer of the protocol, Layer 1 is also used for settling any disputes between users on Layer 2.

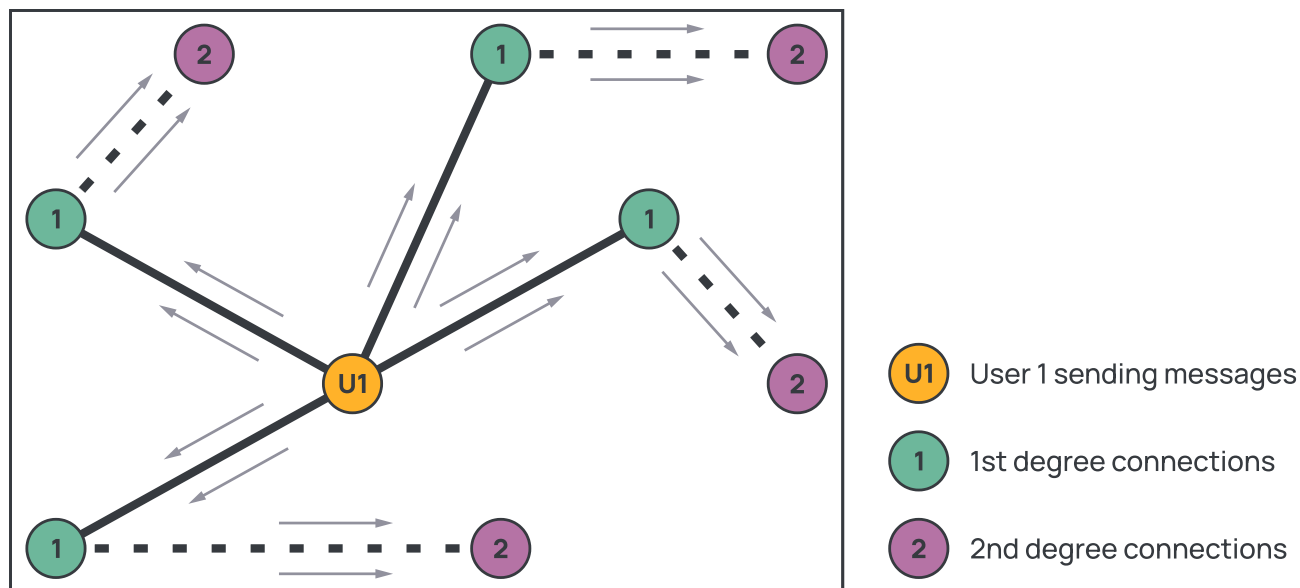
Maxima

Information transfer

Off-chain

Maxima is the information transport layer running over the Minima network.

Communication is point-to-point so that messages can be sent, off-chain, to chosen connected peers.



Maxima enables encrypted, peer-to-peer exchange of information between 1st and 2nd degree connections on the Minima network. Maxima can be used to build censorship resistant messaging applications and will be used for sending data required for Layer 2 communication on Omnia.

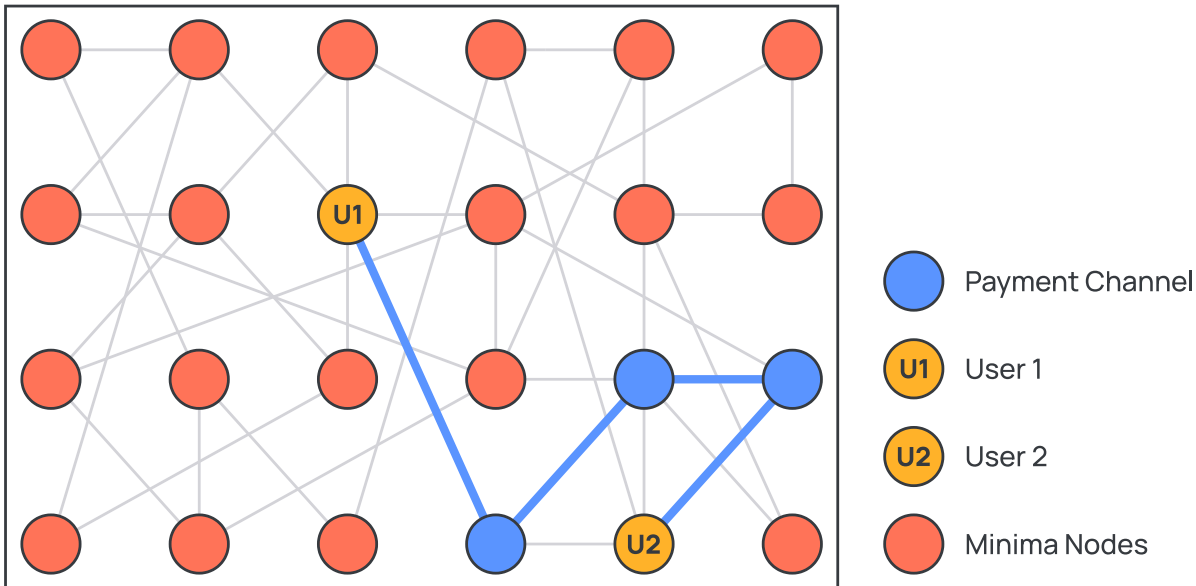
Layer 2: Omnia

Value transfer

Off-chain

Minima's Layer 2 - Omnia - is fast, cheap and scalable. It enables instant peer-to-peer payments by creating bi-directional payment channels between participants, using the latest Lightning technology - ELTOO. It is analogous to Bitcoin's Lightning Network but more advanced.

Minima can also support alternative scaling solutions such as state chains and side chains with the potential for further solutions in the future.



Once users have set up their transacting relationship on Minima, they can perform **all transactions off-chain**, on Omnia. This is where the bulk of peer-to-peer exchange between users of the network takes place. It is **affordable and fast**, as each payment is not settled on the blockchain and transactions are only processed by the relevant users rather than the entire network, as occurs on Layer 1.

Based on **ELTOO** technology, Omnia can do more than simple payments. It can do any smart contract sequence with a given subset of users and subset of coins. Using hash time locked contracts (HTLCs) and payment channels, users can keep a 'tab' of their unsettled balances indefinitely, until they wish to settle on Minima (Layer 1).

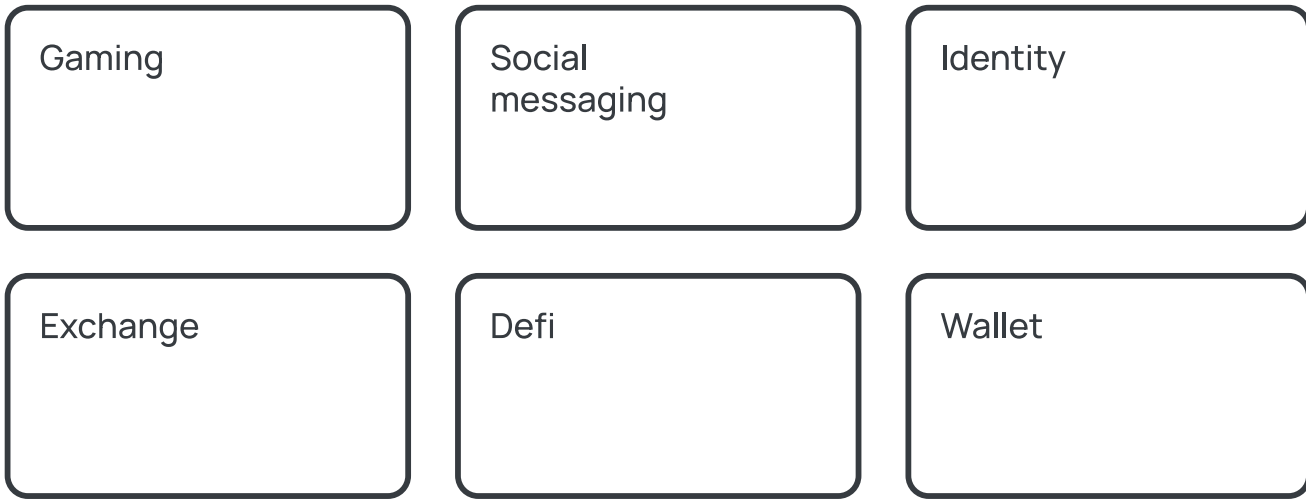
Omnia unlocks the possibility of essentially unlimited transactions per second (TPS).

Layer 3: MiniDapps

Web3 Applications

MiniDapps are truly decentralised applications built using the functionality enabled by any/all components of the Minima network:

1. Value transfer on Minima
2. Information transfer on Maxima
3. Unlimited transaction per seconds on Omnia



MiniDapps are decentralized web applications combining the utility provided by Maxima, Minima and Omnia with Minima's scripting language. The front-end for MiniDapps can be written using the widely known JavaScript, HTML and CSS. Minima's KISS scripting language is Turing-Complete, allowing for powerful smart contract driven applications. Building a MiniDapp is accessible to any web developer.

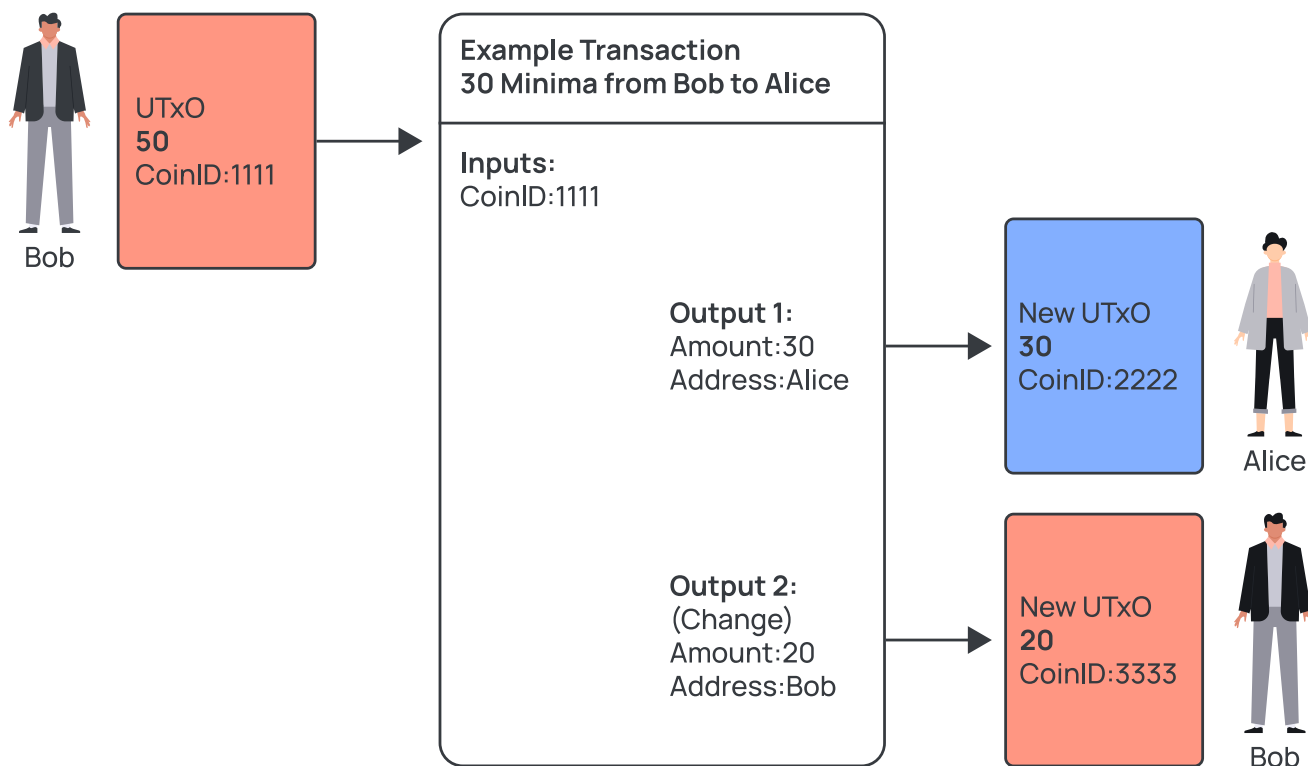
Core Concepts

UTxO Model

Minima uses the UTxO (Unspent Transaction Output) Model, like Bitcoin. A Transaction Output is a specific amount of Minima, identifiable through a unique ID called a **Coin ID**. Each Transaction Output can be considered analogous to a physical coin in that they can represent different amounts of currency and must be spent in whole.

The Minima blockchain keeps track of the UTxO set on the network and who can spend them. The UTxO set circulating in the network fluctuates as users spend coins and create new ones as outputs from transactions. The sum of the value of all the UTxOs in the network will equal at most 1 billion Minima. One or more UTxOs are used as inputs into transactions and one or more new UTxOs will be created as outputs.

The example below shows a transaction of 30 Minima from Bob to Alice. A UTxO worth 50 Minima is used as an input and two new UTxOs are created as outputs - one worth 30 Minima which is sent to the recipient and one of 20 Minima which is returned to the sender as change. Just like change is received when physical coins are spent.



Transaction Proof-of-Work (TxPoW)

Minima requires users to provide work, in the form of hashing, to 'mine' their own transactions, this is **TxPoW**.

Once a user has contributed a small amount of work (~1 second of hashing) they have contributed enough for their transaction to be sent around the network.

NOTE

There are no financial rewards for mining your transactions e.g. block rewards or collection of fees, the reward is simply being able to broadcast a (valid) transaction to the network.

Transactions are held in **TxPoW units**, which are propagated around the network and may or may not become blocks. TxPoW units contain a user's transaction and the hashes of other unconfirmed transactions known to the user's node. TxPoW units become blocks if they, by chance, meet the level of difficulty required to become a block.

This ensures that all users can contribute to the construction of the chain and is in contrast to Bitcoin where users rely on other dedicated 'miners' with specific hardware to provide PoW and propagate their transactions on their behalf.

The Burn

The Burn is a small cost (fee) which is incurred when sending transactions on the Minima network during times of high demand. This cost, denominated in Minima, is 'burned' i.e. removed from Minima's hard-capped supply, making Minima a deflationary currency as the overall circulating supply slowly decreases over time.

The burn serves multiple purposes:

- **A strong incentive to propagate and process a transaction:** All users in the network will benefit from the decrease in supply as coins that are left become more scarce and therefore more valuable.
- **A method for ordering transactions and regulating on-chain traffic:** Similar to the fee model on other blockchains, the burn serves as a selection method for deciding which unconfirmed

transactions will be added to a block. The higher the burn amount in a transaction, the more likely it is that a transaction will be added to a block.

- **A mechanism for spam prevention by making Denial-of-Service (DoS) attacks expensive:**

The burn may be high during periods of heavy traffic or spam and, as it rises, traffic will decrease, self-regulating the system. The burn can be nil or very low when traffic is at manageable levels as the total amount is not important, only the relative burn amount in comparison to other transactions.

While some blockchains have implemented burns that require a central entity to actively buy back and burn coins or tokens (for example by sending them to an inaccessible address, effectively destroying them); others have a burn directly integrated into the protocol.

In Minima, the ability to burn coins is directly coded into the protocol, meaning it does not rely on any entity, but rather is enforced by the network as a function of demand for block space.

MMR (Merkle Mountain Range) Database

To ensure that all users of the network can contribute to the construction of the chain, the chain needs to be small enough to run on a mobile device. This would not be possible if the entire history of the chain was required as this would be too much of an overhead for a mobile device. Therefore the blockchain must be constantly reduced in size to meet this requirement. This is known as **pruning**.

The impact of pruning means that the full transaction history of the chain is not kept, therefore a storage mechanism is required to keep track of coins that were created in blocks that have since been pruned. **This is the role of the MMR database.**

Hence, users must keep track of their spent and unspent coins/TxOs (Transaction Outputs) independently of the chain. Each coin is stored as a leaf node in a tree structure (a **Merkle hash-sum tree**). Then, using a collection of nodes in this tree, a proof path can be created from the coin to a peak of the tree, proving the existence of a coin even if the block that it was created in has been pruned.

All users only keep the parts of the MMR tree required to create the proofs for their own coins, which is a tiny amount of data compared to all the coins in the network. Users are also required to store the peaks and the root of the tree so that they can validate a Coin Proof that is presented to them by another user.

When a user wishes to spend their coins, they must provide the up-to-date, valid proof that it is unspent. Any other node in the network can verify this proof by calculating the peaks and root hash of the MMR tree from the proof and ensuring it matches their own values for the peak and root hash.

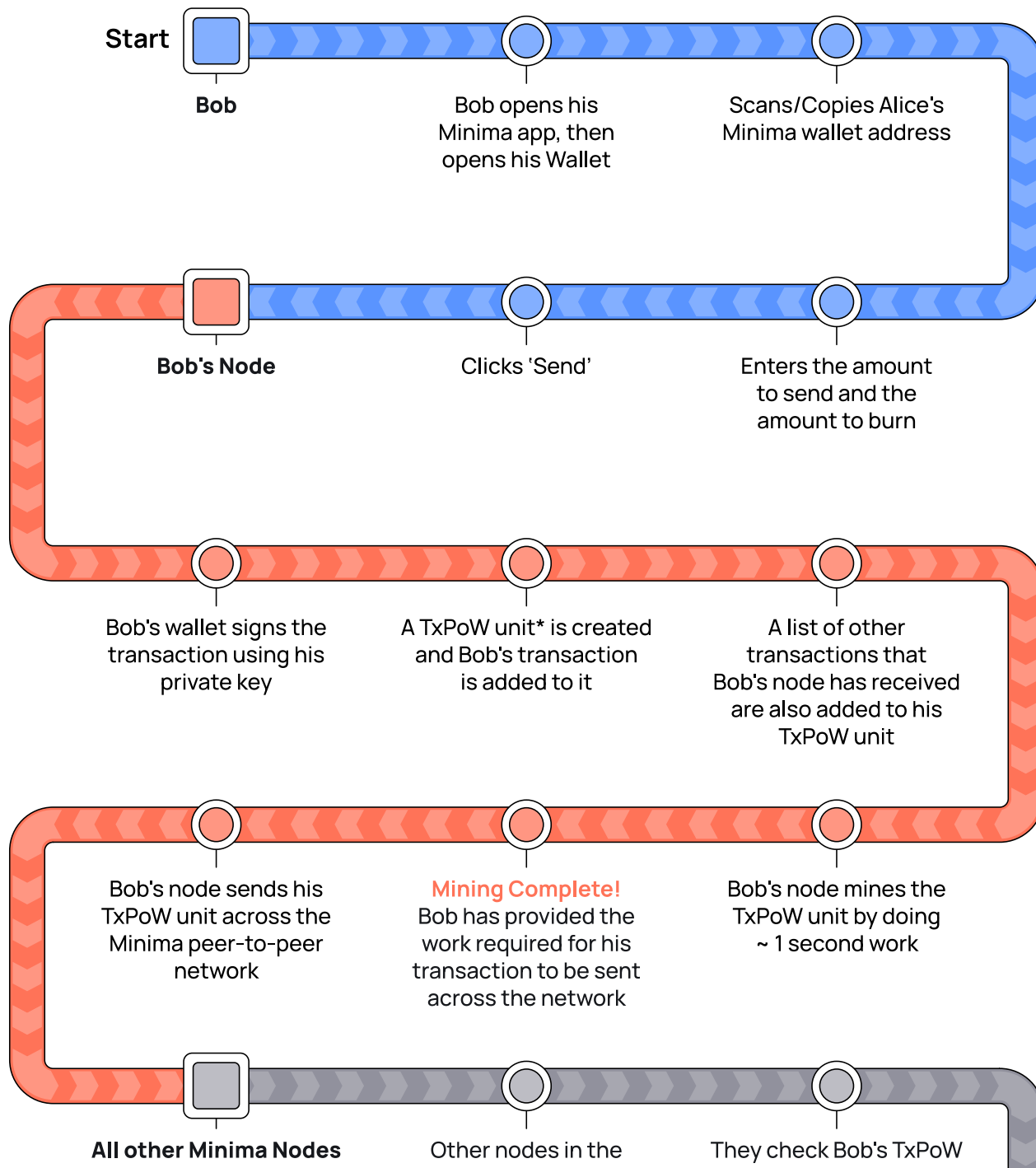
For further information, see [MMR Database](#).

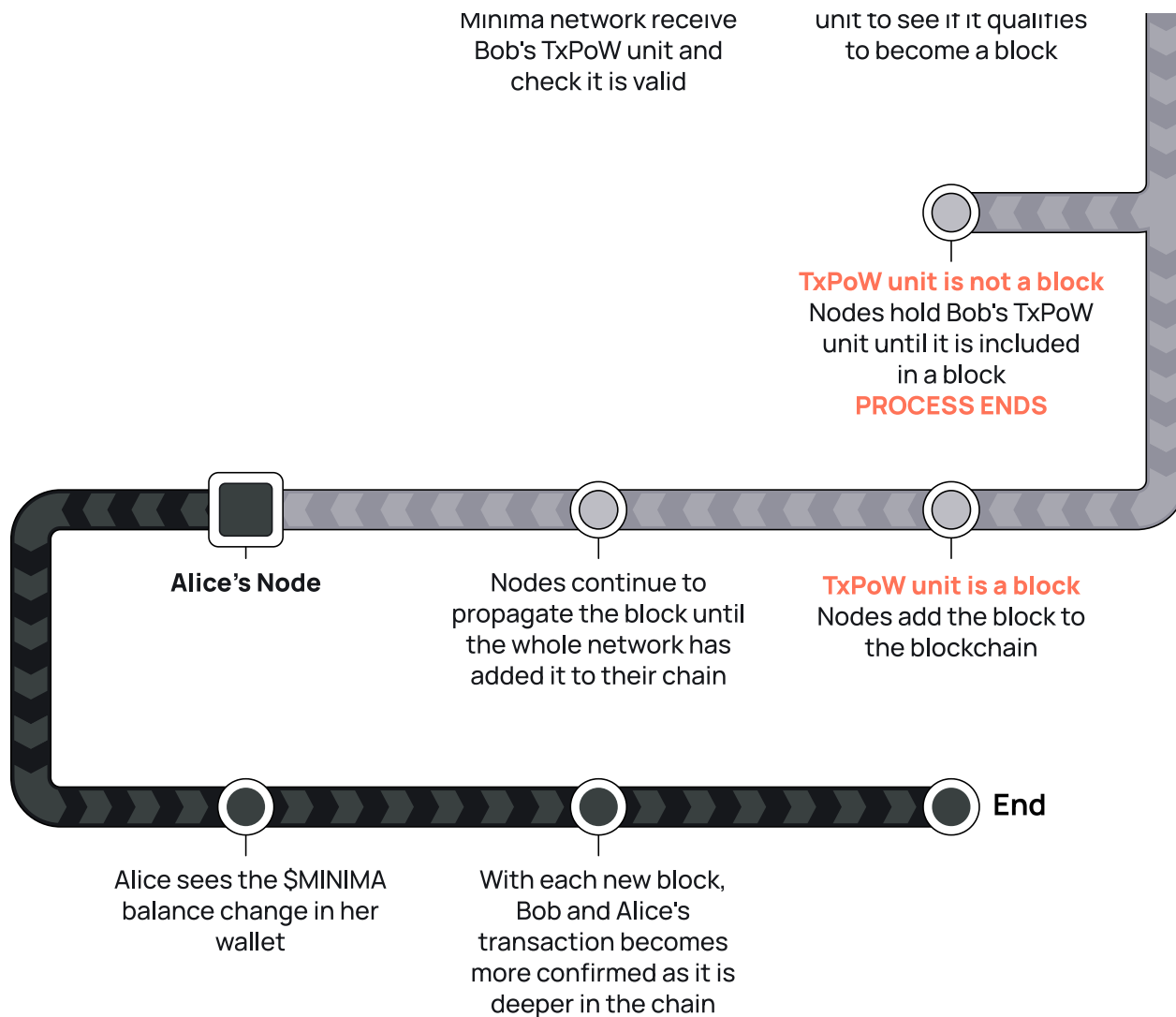
Transactions

Transaction Lifecycle

The following diagram explains the basic process from a Transaction to TxPoW unit to Block.

Bob to Alice





*TxPoW stands for Transaction Proof-of-Work. TxPoW units can be thought of as blocks, although not all will become blocks.

Structure

All transactions have the following structure:

- Inputs
- Outputs
- State variables
- Link hash
- Transaction ID

Inputs

Transaction inputs are coins (UTxOs) that will be spent in the transaction, identified by the Coin ID. One or more coins may be required depending on the value of the coins owned by the user and the amount they wish to spend.

Example:

If a user wishes to spend 100 Minima but they only have two coins worth 40 and 90 Minima, they must use both coins as inputs to the transaction and receive change of 30 as a new coin.

Each coin is identifiable with a unique CoinID has the following attributes:

Coin Attribute	Description	Type
CoinID	The globally unique identifier for a coin. The coin id of a new UTxO is hash(first input coin in txn output_num_in_txn)	64 byte hash (MiniData)
Amount	The amount of 'Minima'. Even custom token transactions are just amounts of coloured Minima (see Coloured Coins)	MiniNumber
Address	The hash of a script. If no custom script is provided, the script will simply be RETURN(SIGNEDBY([PublicKey of coin owner])) i.e. the transaction must be signed by the coin owner before it can be spent. All addresses are P2SH (Pay to Script Hash).	64 byte hash (MiniData)
Token ID	The Token ID. Minima is 0x00. Everything else has a full 64 byte hash. Tokens are created by colouring Minima.	64 byte hash (MiniData)
Token	Token Details (see Coloured Coins)	
Floating	True or False. Set when you create a transaction. If True, the Coin ID is ignored so that any valid coin that has the same amount, address and Token ID can be used.	Boolean

Coin Attribute	Description	Type
Store State	True or False, depending on whether the state is stored for this coin	Boolean
State	The state variables (0-255) of the transaction this coin was created in. You can access this data from scripts.	Integer
MMR Entry	The MMR database entry number for this coin	MMREntryNumber
Spent	True or False, depending on whether this coin has been spent or not.	Boolean
Created	The block number this coin was created in.	Integer

Example coin details:

```
coins relevant:true
{
  "command":"coins",
  "params":{"
    "relevant":"true"
  },
  "status":true,
  "response":[{"
    "coinid":"0xB76A17B0444D40A90697FCC7EFFF1917F7B4AC7FD70D7E70323DC560B6A3CF2",
    "amount":"98",
    "address":"0x0B00C23C8B4DFBDEC76FEE908ADF1BD396A5E92DCC826D3ADD26E4140CFA1DC0",
    "tokenid":"0x00",
    "token":null,
    "floating":false,
    "storestate":true,
    "state":[],
    "mmrentry":"439",
    "spent":false,
    "created":"167764"
```



```
}]  
}
```

Outputs

Transaction outputs are new coins (UTxOs) created as a result of a transaction and include:

1. **Amount to send to recipients:** One or more amounts with recipients' addresses specified.
2. **Change to the sender:** When the amount being sent is less than the value of the coin inputs, change needs to be returned to the sender's address.

BURN

Any difference between inputs and outputs will be burned, reducing the circulating supply of Minima.

State Variables

Transactions include State Variables for storing public data and previous transaction states to retrieve information for scripts.

256 state variables are allowed per transaction, ([see Scripting](#)).

Link Hash

Each transaction has an associated Burn transaction, which may or may not have a value.

A burn transaction uses the transaction ID of the transaction it relates to, as its link hash.

This ensures the burn transaction can only be spent with the transaction it is linked to.

For main transactions, the link hash is set to `0x00`.

Transaction ID

The transaction ID is initially set to `0x00` and is then calculated as the hash of the transaction object, including its inputs, outputs, state variables and linkhash.

Burn Transactions

Burn transactions are created automatically and have the following structure compared to a main transaction:

Main transaction structure

Transaction ID: <i>Hash(Transaction object)</i>
<i>Inputs</i>
<i>Outputs</i>
<i>State Variables (0-255)</i>
<i>Linkhash (0x00)</i>
<i>Transaction ID</i>

Burn transaction structure (automatically set)

Transaction ID: <i>Hash(Transaction object)</i>
<i>Inputs (matching the main txn)</i>
<i>Output (amount to burn)</i>
<i>State Variables (matching the main txn)</i>
<i>Linkhash (main txn ID)</i>
<i>Transaction ID</i>

Transaction Validity

For a transaction to be valid:

1. The sum of inputs must be greater than or equal to the sum of the outputs for each Token ID
2. All inputs and outputs must be valid Minima amounts (between 0-1 billion)

3. All inputs must have unique coin IDs
4. It must have no more than 256 state variables

When checking the validity of transactions, the *monotonicity* is checked to determine whether a transaction needs to be checked more than once.

Scripts of **Monotonic** transactions only need to be checked once i.e. they are either valid or not e.g. a simple RETURN(SIGNEDBY(..))

Scripts of **Non Monotonic** transactions need to be checked multiple times. Any script in a transaction that references global variables @BLKNUM, @BLKDIFF or @INBLKNUM is not monotonic as its validity will change depending on these variables.

MMR Database

Overview

As the blockchain is heavily pruned, users must store proof that their coins are unspent.

This is the role of the Merkle Mountain Range (MMR) Proof database. The MMR is a **hash sum tree** containing the proofs for all Transaction Outputs (TxOs) i.e. coins in the system.

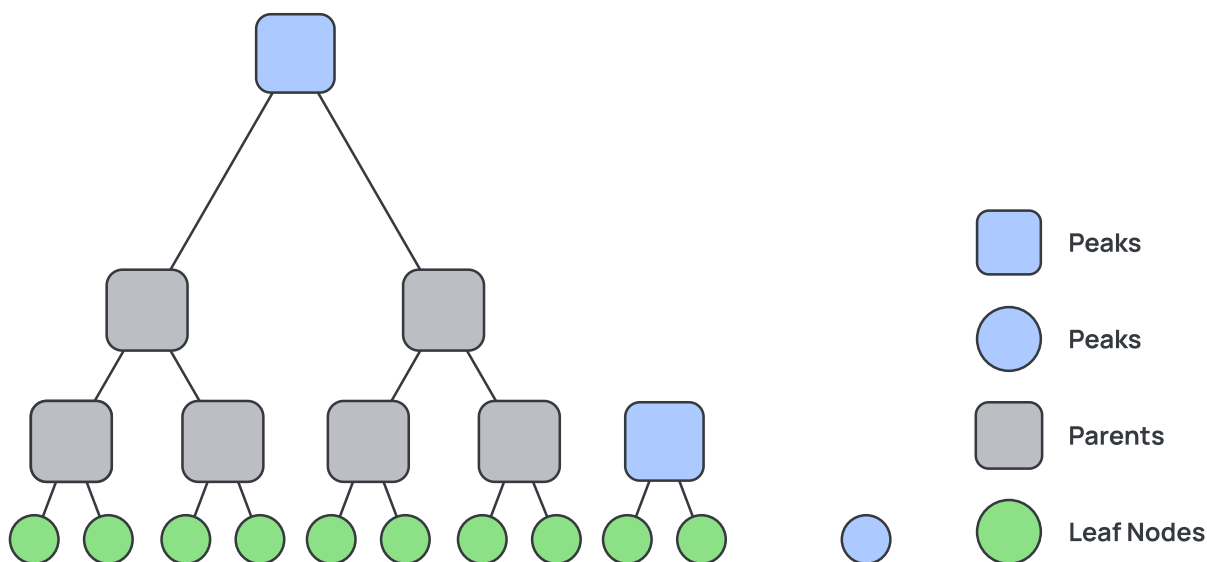
Note: these coins may be Spent Transaction Outputs (STxOs) or Unspent Transaction Outputs (UTxOs).

The tree is **append-only** and is updated as coins are spent and created. For each new UTxO created from a transaction output, a new leaf node is created in the MMR.

Coins are hashed in pairs, building up the largest **binary tree** possible until a new tree is required. As new trees are required, they start to look like a range of mountains - giving the MMR its name.

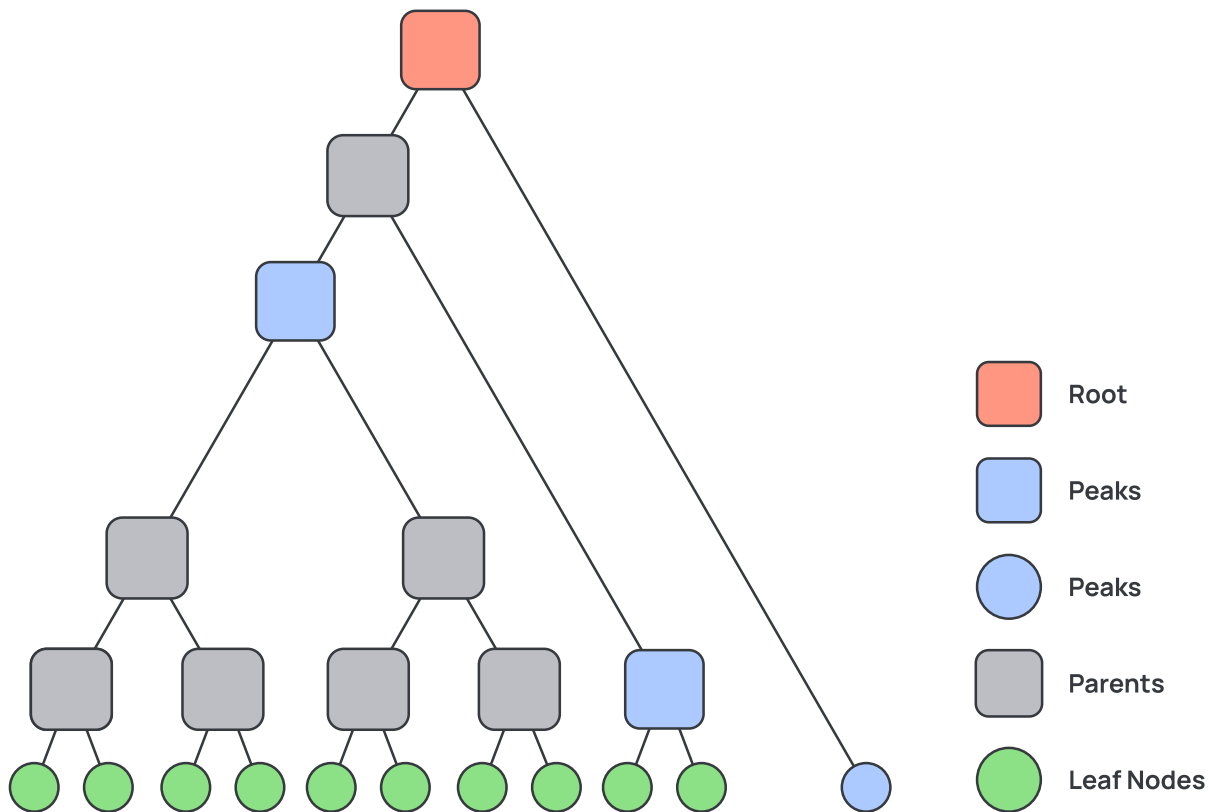
When the total number of leaf nodes (Coins/TxOs) are not equal to 2^n where $n = \text{int}\{0, \dots, 256\}$, there will be multiple trees of different heights, creating multiple peak nodes as shown below.

Diagram: Merkle Mountain Range (MMR) with 11 coins (green) and three peaks (blue)



To create a single MMR tree, the peaks must be collected (or 'bagged') starting from left to right. Until a single root hash is found.

Diagram: A complete Merkle Mountain Range (MMR) with three peaks and root



Each node in the tree will have a globally unique reference to it by combining the row/level it is in and its entry number on the row. Using a hash table to track entries, each node can be identified through a reference [R,E] where R is the row number and E is the Entry number.

Diagram: A complete MMR with hash table references [row, entry number]



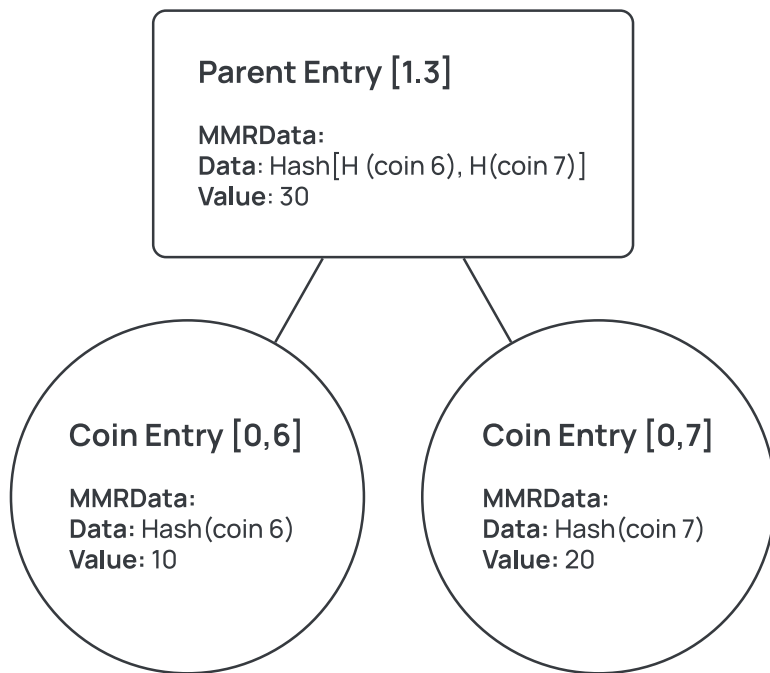
The MMR can be thought of as a book, where all users keep a copy of the spine (root and peaks) and their own page in the book (their CoinProofs). When a user wishes to spend their coins, they provide their page and the spine. Provided their page fits the spine and the spine matches that of the other nodes in the network, the user's coins can be proved to be valid.

MMR Data

Each node in the MMR has unique MMR Data consisting of a hash and a value, defined as follows:

MMRData Object	Leaf Nodes - TxOs (Row 0)	Parent Nodes (inc Peaks & Root)	Type
Data (Hash)	Hash(coin object) The coin could be spent or unspent.	Hash[Hash(left child data object), Hash(right child data object), value object]	MiniData (32 byte hash)
Value	Minima Value of coin, if unspent, Or 0, if spent	Sum of the value of child nodes	MiniNumber

Diagram: Example MMR Data for two coins and a parent node in the MMR.



Each entry in an MMR is defined by its attributes:

MMREntry Attribute	Description	Type
Row	The hash table row representing its level in the tree (where coins are Level 0)	String
Entry Number	The index of the Entry on a specific row from left to right, starting from 0	MMREntryNumber
Data	The MMR data (Hash and Value) of the entry	MMRData

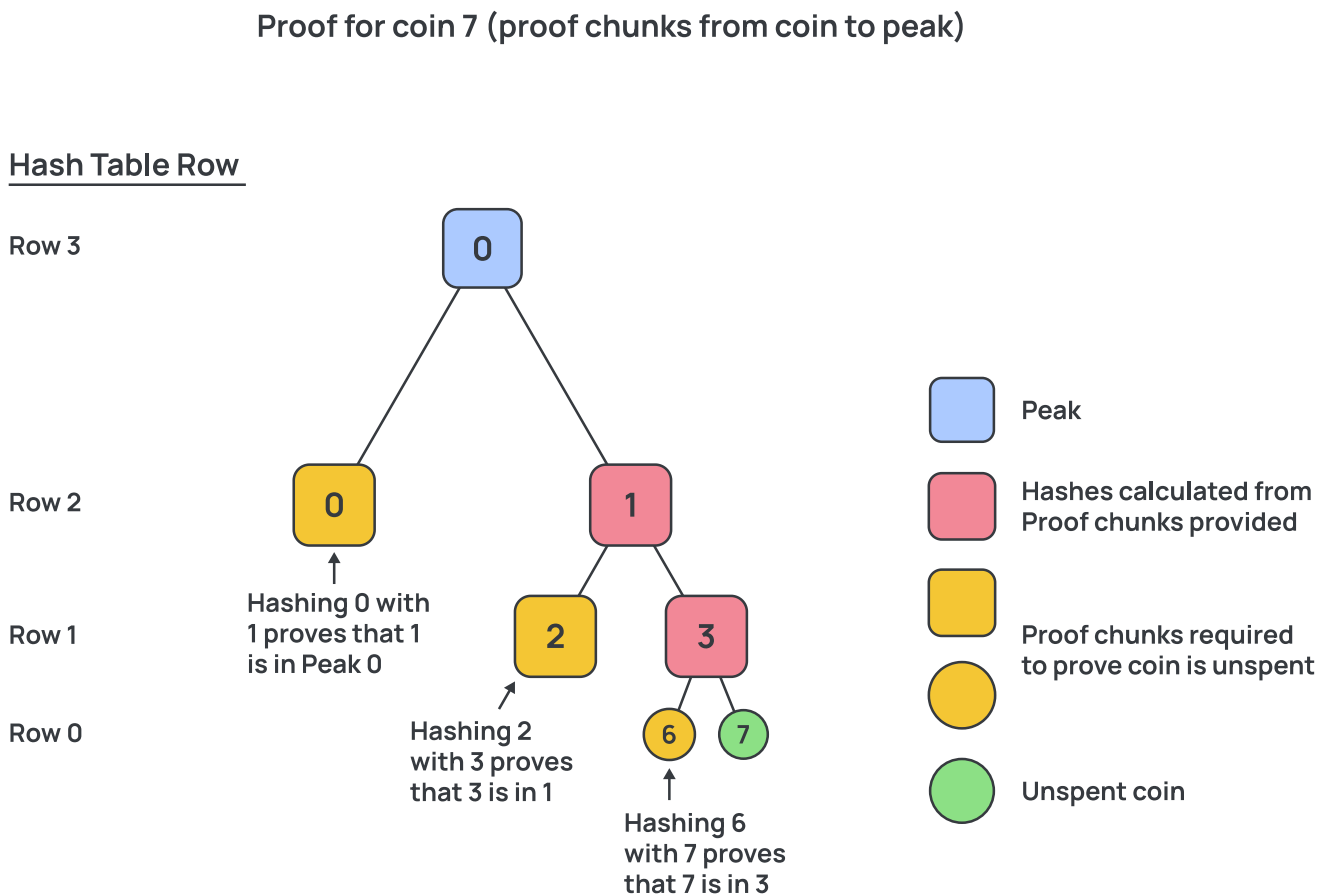
Proofs

When a user wishes to spend their coins, they must provide proof that their coins are unspent by providing a **CoinProof** for each coin they wish to spend. **A CoinProof is a list of Proof Chunks** that any other node can use to independently verify that someone else's coin exists and is unspent, without having to store the proofs for every coin in the network.

Given a CoinProof, any node verifying a transaction can calculate the path (i.e. the intermediate parent hashes), from another user's coin to a peak in the MMR. If the calculated peak hash matches the peak hash from their own MMR, the CoinProof is valid, otherwise the CoinProof and transaction are not valid.

Proof Chunks consist of the MMR Data (hash and value) for an MMR entry and a True/False flag indicating whether it is a left sibling or not.

Diagram: Example CoinProof for coin 7 (coin to peak)



The CoinProof for coin 7 consists of the coin and the yellow Proof Chunks, i.e. entries $\{[0,6], [0,7], [1,2], [2,0]\}$, so that:

- Hashing $[0,6]$ with $[0,7]$ calculates parent $[1,3]$
- Hashing $[1,2]$ with $[1,3]$ calculates parent $[2,1]$
- Hashing $[2,0]$ with $[2,1]$ calculates peak $[3,0]$

Any node receiving this CoinProof is able to calculate the parents and the peak node, and by comparing it to their own peaks, proving that Coin 7 is valid.

MMR Sets

When a coin is **spent**, the Spent flag of the coin changes from false to true, changing the hash of the coin.

When a new coin is **created** (as an output to a transaction), a new coin is added as a leaf node to the tree.

In both cases, the intermediate parent nodes, peaks and root will need to be calculated for the new hash of the coin. Therefore the CoinProofs for all coins in the system change with each new block and it is therefore essential for users to keep up to date with the latest block.

Each block contains an **MMR Set** containing updated and new MMR Entries for all the spent and new coins of the main transaction in the block.

Each block has its own set of MMR entries with the following attributes:

MMR Attribute	Description	Type
Block Time	The blocktime for the MMR set	MiniNumber
Elen	How many entries in this MMR set	MiniNumber
Entry Number	The latest entry number	MMREntryNumber
Set Entries	The hash table elements for all the MMR Entries in this set. HashTable<0,1> is the entry on Row 0, Entry index 1	Enumeration of MMR Entries

Traversing the MMR

Starting from any node in the MMR, we can navigate through it as follows:

Operation	Row Operation	Entry Number Operation
Right Sibling	-	Add 1
Left Sibling	-	Minus 1
Parent	Add 1	Divide by 2, Round down to nearest integer
Left Child	Minus 1	Multiply by 2
Right Child	Minus 1	Multiply by 2, Add 1

Separate MMRs are also used to store Signature Proofs and Script Proofs.

TxPoW Units & Blocks

The **TxPoW unit** is the main building block of the Minima Blockchain.

Transactions are included in TxPoW units which have the potential to become blocks added to the chain.

TxPoW Units

Before a transaction can be posted to the network, it must be added to a **TxPoW unit** with other essential data and a small amount of 'work' must be done i.e. it must be 'mined'.

During the TxPoW creation process, in addition to the main transaction, unconfirmed transactions in the mempool will also be added to the body of the TxPoW unit, serving to further propagate mempool transactions to known peers.

After being mined, a TxPoW unit will be propagated to the network either as a block if it meets the network block difficulty level, or as a basic TxPoW unit which serves only to propagate known unconfirmed transactions. This ensures block creation is a 'chance' encounter.

A node can carry out several activities with a TxPoW unit depending on the situation:

Generate: When a user wishes to send a transaction, or is required to send a **Pulse** to the network, their node will generate (construct) a TxPoW unit containing their transaction and the hashes of other unconfirmed transactions it knows about in the **mempool**. TxPoW units are analogous to compact blocks in Bitcoin ([see Constructing TxPoW Units](#)).

Mine: Before a node can propagate its TxPoW unit to the network, it must mine the TxPoW unit by cycling through different **nonces** (numbers), adding the nonce to the TxPoW header and hashing the result. Once the resulting hash meets the level of difficulty required by the network (~1 second work), they can propagate their TxPoW unit to other nodes in the network.

NOTE

The **transaction difficulty** sets the minimum amount of work a node must provide before their TxPoW can be propagated across the network (~1 second work). This ensures the network has

received the transactions in their TxPoW unit, **however this does not automatically mean that their transactions are in a block.**

TxPoW units only become blocks if, by chance, the **block difficulty target** is met in the process of meeting the transaction difficulty target.

Check: When a node receives a TxPoW unit from another node on the network, it must check it to ensure its validity before processing and forwarding it on to its peers.

Process: When the node has checked that a TxPoW unit is valid, it will process it, creating a **TxBLOCK** and adding it to the chain as a node in the **TxPoW tree** if it meets the required difficulty level to become a block.

Diagram: The structure of a TxPoW Unit

TxPoW ID (Hash[Header])
Header <i>Nonce</i> <i>ChainID</i> <i>TimeMilli</i> <i>Block Number</i> <i>Block Difficulty</i> <i>Cascade Levels</i> <i>Parent Blocks</i> <i>MMR Root</i> <i>MMR Total</i> <i>Magic</i> <i>Body Hash</i>

TxPoW ID (Hash[Header])
Body <i>Random Number</i> <i>Txn Difficulty</i> <i>A Transaction</i> <i>A Witness</i> <i>Burn Txn</i> <i>Burn Witness</i> <i>Txn List</i>

Header

Header Field	Description
Nonce	The final nonce (number) that was included in the TxPoW header so that, when hashed, the required difficulty was achieved.
Chain ID	The Chain ID - This defines the rules this block was made under, MUST be 0x01
TimeMilli	Time this TxPoW was created in milliseconds since the epoch of 1970-01-01T00:00:00Z
Block Number	Block height to be used if this TxPoW unit becomes a block
Block Difficulty	The Difficulty required for this unit to be considered a valid block
Cascade Levels	The maximum number of levels in the Cascade (32)
Super Parents	Pointer to its immediate previous block and to the most recent block at each Super Parent level for cascading.

Header Field	Description
MMR Root	The root hash of the MMR (to prove coins existed using a proof and TxPoW header)
MMR Total	The sum of all coins in the system (using a hash sum tree, the total amount of Mir every block removing the possibility of inflation bugs)
Magic	Chain parameters - the magic numbers: CurrentMaxTxPoWSize,CurrentMaxTxnPerBlock,CurrentMinTxPoWWork,CurrentM
TxBodyHash	The hash of the TxPoW body

Body

Body Field	Description	Type
Random Number	A Random number so that everyone is working on a different TxPoW in the pulse	MiniData
Txn Difficulty	The Difficulty required for this unit to be a valid TxPoW unit. The value that all users try to achieve when cycling through nonce values.	MiniData
Transaction	Transaction ID for the main transaction. UTxO (coin) inputs, outputs, state variables, linkhash	Transaction
Witness	Signature Proofs;Coin Proofs (pointing to a valid unspent MMR entry in the past 24 hours for each input coin used in the txn); Script Proofs (for the various P2SH addresses used)	Witness
Burn Txn	Inputs, outputs, state variables and linkhash for the Burn transaction paying for the transaction the user is trying to send. Can be empty.	Transaction

Body Field	Description	Type
Burn Witness	The Witness data for the Burn. Signatures, MMR Proofs and scripts. Can be empty.	Witness
Txn List	List of the hashes of mempool transactions to propagate. These will become confirmed if this TxPoW unit becomes a block. Only the hash of transactions are added since transactions have already been sent across the network.	MiniData array

The default maximum size of a TxPoW unit is 64MB.

Witness

A Witness provides three proofs that prove a transaction is valid. Each proof is stored in an MMR tree. It has the following attributes:

Witness Attribute	Description	Type
SignatureProofs	The MMR Proofs for the Signatures	ArrayList<Signature>
CoinProofs	The MMR Proofs that each input Coin is valid and unspent	ArrayList<CoinProof>
ScriptProofs	The MMR Proofs for Scripts used in the transactions	ArrayList<ScriptProof>

TxBlocks (Blocks)

TxBlocks are TxPoW that become blocks and get added to the blockchain. TxBlocks can also be referred to as SyncBlocks as they are required for syncing when a new user joins (or existing users rejoin) the network.

They include the following details:

TxBlock Attribute	Description	Type
TxPoW	The TxPoW object that became this block	TxPoW
Previous Peaks	The MMR Peaks from the previous block	ArrayList<MMREntry>
Spent Coins	The CoinProofs of all the input (spent) coins, unspent as of the last block	ArrayList<CoinProof>
New Coins	A list of all the newly created coins	ArrayList<Coin>

The Blockchain

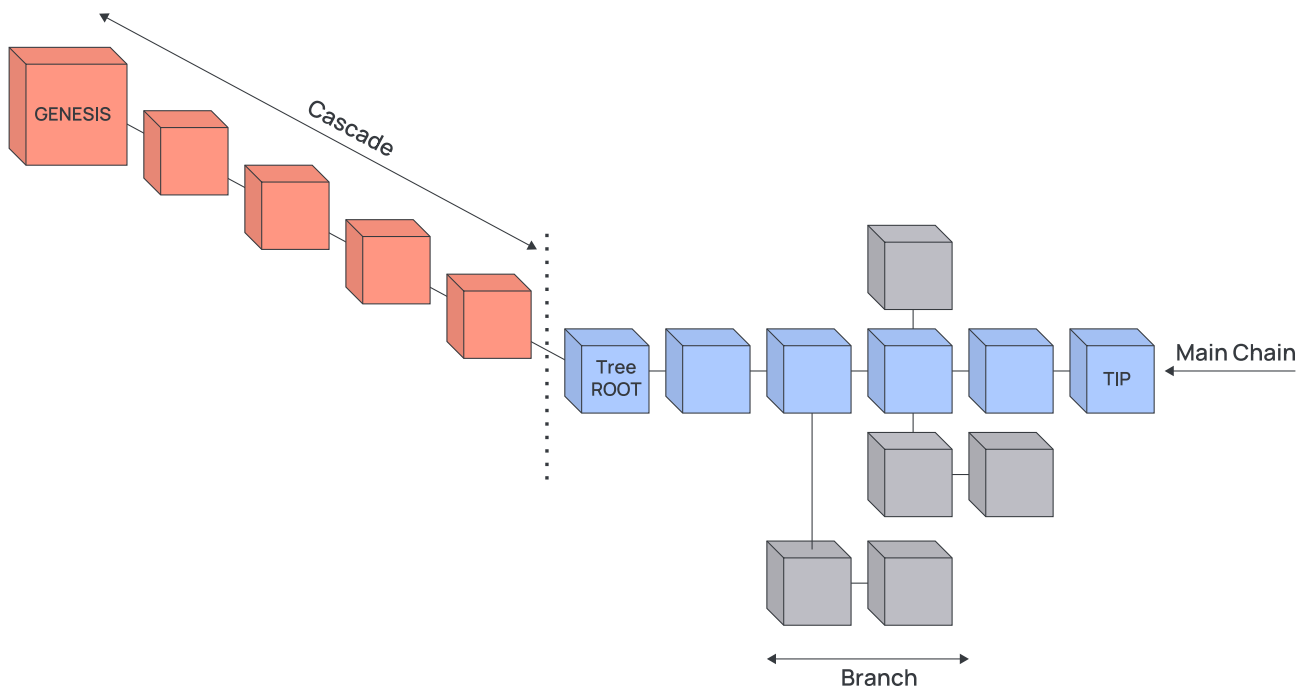
Overview

The Minima blockchain is structured to be compact and 'lightweight', it is therefore heavily pruned to meet this requirement. The chain itself is a TxPoW Tree containing TxBlocks.

The **TxPoW Tree** is the main blockchain consisting of the most recent TxBlocks.

The **Cascading chain** is an unbroken chain of the headers of so called 'Super Blocks', designed to record and prove, in an immutable way, the total cumulative Proof-of-Work input into the network without having to keep a record of all individual blocks. Blocks are added to the Cascading chain on a periodic basis and the root of the TxPoW Tree always remains connected to the tip of the Cascading chain.

Diagram: Cascading Chain + TxPoW tree (Blockchain)



TxPoW Tree Node

If a TxPoW unit becomes a block (TxBlock), it will be added to the blockchain and become a node in the TxPoW tree. The TxPoW Tree Node provides the structure required to hold the TxBlock in the tree.

Every TxPoW Tree Node has the following attributes:

TxPoW Tree Node Attribute	Description	Type/Size
TxBlock	A syncblock representing this node	TxBlock
MMR Set	The MMR (with blocktime, entry number, and entries) can be constructed from the TxBlock	MMR
Coins	ALL The Coins - both spent and unspent	ArrayList<Coins>
RelevantMMRCoins	The Coins that are relevant to THIS USER	ArrayList<MMREntryNumber>

TxPoW Tree

The TxPoW tree is the main blockchain and has a **root** - the start of the chain, **tip** - end of the chain, **blocks** (tree nodes) and **length** (number of blocks).

The root of the TxPoW tree meets the tip of the Cascade.

Branching in the tree can occur if multiple blocks with the same height (block number) are found simultaneously. Branching is a natural occurrence and will resolve over time as a heavier branch continues to be built on.

The Cascading Chain

The Cascading Chain is a component of the Minima Protocol designed to record and prove, in an immutable way, the total cumulative Proof-of-Work input into the network without having to keep a

record of all individual blocks.

By identifying multiple levels of difficulty over and above the required block difficulty, so-called 'Super Blocks' emerge which, by chance, provide orders of magnitude more PoW to the network than the typical (Level 0) block. The Cascading Chain uses 'Levels' to store these Super Blocks as a representation of the Proof-of-Work input into the network, allowing for heavy pruning of Level 0 blocks, without losing any PoW.

The Cascading Chain provides an objective proof of the current 'heaviest', and hence valid, chain.

Proof-of-Work is provided, in the form of electric energy, by all users running the Minima Protocol through the process of hashing. A user's device must perform a, pre-determined, minimum amount of hashing each time a user:

1. Mines their transaction before forwarding to peers
2. Mines a 'Pulse' TxPoW before forwarding to peers

Attributes of the Cascading Chain:

1. The Cascading Chain consists of 32 levels (0-31), with a maximum of 128 blocks at each level.
2. The Cascading Chain grows logarithmically, as each level is twice as difficult to achieve as the previous level.
3. Over time, the cumulative sum of the PoW (the 'weight') recorded in the Cascading Chain will tend towards the weight of the chain that would have existed had no blocks been pruned.
4. The Cascading Chain is unbroken. Each block in the Cascade references its previous super parent block in the Cascade.
5. At 100 block intervals, the heaviest chain (consisting of all levels in the Cascade and the heaviest branch) is processed and the Cascading Chain is updated.

Definitions:

Block Difficulty Target: A system set parameter influencing the average number of hashes required for the network to mine a block every 50 seconds (or as close to). The higher the difficulty, the more PoW (energy) required to mine a block.

Cascade Levels: The Cascading Chain consists of 32 levels, where each level consists of blocks which, by chance, exceeded the block difficulty target of the previous level by a factor of 2. e.g. A

block in level 3 of the Cascading Chain achieved twice the difficulty of a block in level 2.

The Cascade: The chronological, unbroken chain of blocks consisting of a maximum of 128 blocks at each level. The block at the root of the Cascade (after Genesis) will be the block which satisfies both 1) the earliest timestamp and 2) the highest (most difficult) level. The Cascade does not include the TxPoW tree and has no branches.

Super Block: Any block which achieves the difficulty required to take a position on the Cascade.

Current level: The level representing how deep in the Cascading Chain a particular Super Block is currently positioned.

Super Level: Also the Maximum Level. The level representing the furthest depth a Super Block could reach on the Cascade (determined at random by the difficulty level achieved during the process of mining the TxPoW unit).

Base Weight: A block's base weight is equal to its difficulty value. This is the average number of hashes that would be required to meet this Block's difficulty target.

Current Weight: The base weight multiplied by a factor dependant on the level in the Cascade the block is currently positioned in, such that $\text{Current Weight} = \text{Base weight} * 2^{\text{Current level no.}}$.

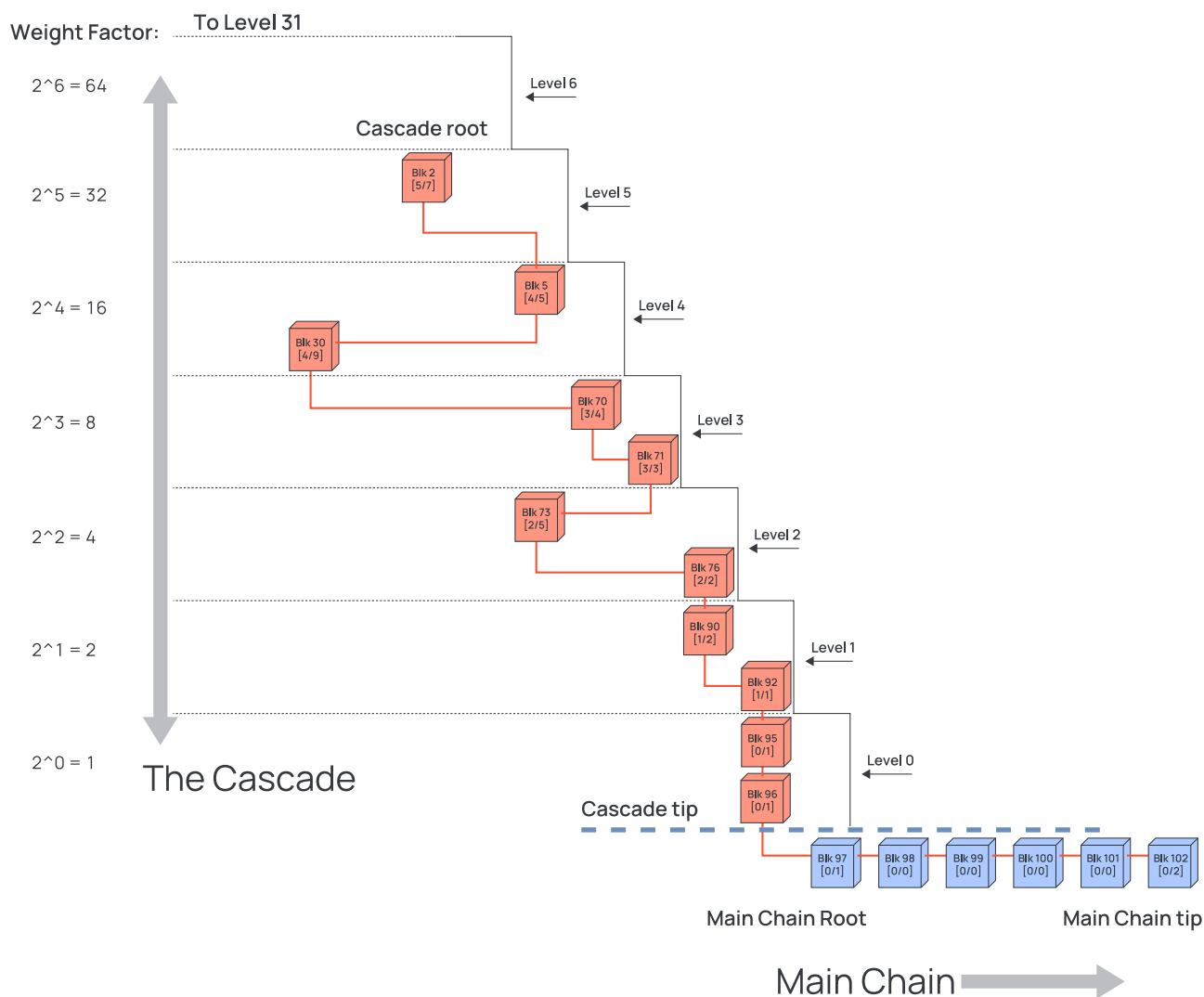
Branch: The main chain (TxPoW Tree) starts at the tip of the Cascade and consists of the most recent 1024 blocks which have not yet been committed to the Cascade. If, due to network latency, two blocks with the same block number are found, there may be multiple branches off the main chain.

Diagram of the Cascading Chain at a point in time

Note: *this diagram illustrates a Cascade and Main chain with test parameters (only 2 blocks at each level). In practice there are a maximum of 128 blocks at each level.*

Diagram: Cascading Chain and Main chain

Cascading Chain with SuperBlock references



The corresponding Cascading Chain in the Minima Terminal. Actual output of **printtree cascade:true** (using Test parameters of 2 blocks at each level and a default weight of 1000)

```

    "command": "printtree",
    "status": true,
    "message": "Printed to stdout"
}
printtree
[5/7] 0x0000436C7690D35E6C42FD5FAD92AD86E3397F2FASAAD5DAD50AF6721F0962EB@ 2
weight:32000.00
[4/5] 0x000184FC2C1674CA3F1822A74BDA265898428337867F9CEC17A30F10F90F886F@ 5
weight:16000.00
[4/9] 0x00001218A849EB1C0922202077563AE57125AF1A490444437157B45005587E088@ 30
weight:16000.00

```

```

[3/4] 0x0003304740533814A703ASE06079DA3EFED964A17908514403FBFE1ED389473E@ 70
weight:8000.000
[3/3] 0x000548332F9278833812CD8508760A8436188847F4FA08BF5C3019626F55872C@ 71
weight:8000.000
[2/5] 0x000144F2E92068E6EOCDD361A942CEAB9DD25AC9E0ADD3966066ACF99FBCB02E@ 73
weight:4000.000
[2/2] 0x000A140060789CF6DA6652F7185FC99E0791E3FF423A028928DC34C4065081DA@ 76
weight:4000.000
[1/2] 0x00004062539C98051845207CBF486BFCF9EC1333700684A3A285FEE2E0BA3CFE@ 90
weight:2000.000
[1/1] 0x0019BCD30CDC83704188816388036DA11C7C1E0BACSE72E62AD286736AD0E6SB@ 92
weight:2000.000
[0/1] 0x001880031AC428632321043BE43FFF6988284A91085408770AC4180200739101@ 95
weight:1000.000
[0/1] 0x001833219F9048291BDAC6634151788878499579C091DS7870F18A087CS83A84@ 96
weight:1000.000
----
97 [0/1] 0x0018488783BFD4FA8F6AEADS8E7063857858E3023095100C51885867124F2F60 txns:0
weight:1000.000/16000.000
--> 98 [0/0] 0x0024C76EBC2F0FF3811960177E94B9DCCF0072590A3E0A0704CB958ABCS33FFC
txns:0 weight:1000.000/15000.000
--> 99 [0/0] 0x003B8CF257AAFEA852ASECF329F07863EB7ED4C2AA0A1820EBBC11EEEAB271DD
txns:0 weight:1000.000/14000.000
--> 100 [0/0]
0x0038A2F83A1300E0F293A688281C42A0C94F13377F1958E9580863933EB0A089 txns:0
weight:1000.000/13000.000
--> 101 [0/0]
0x002CB758E57E6805E3432FDFAB450FCFDE40A700BCB17COA8204FD3995989D02 txns:0
weight:1000.000/12000.000
--> 102 [0/2]
0x00087C8559EE3571E3939DECD5774F82698618704F5149513265CC4C230AF21F txns:0
weight:1000.000/11000.000
--> 103 [0/2]
0x0009FC18DE1881A20107288640A3628280814F310E026896338F0087A13AEC3E txns:0
weight:1000.000/10000.000
--> 104 [0/2]
0x000CC2CA1224EFCF33EE158192468AF1F105C4CEA561EE146EA9206088031807 txns:0
weight:1000.000/9000.000
--> 105 [0/0]
0x00247633330E06AA30AC7440F131583823E955F7F49ACC8468844FC590047656 txns:0
weight:1000.000/8000.000
--> 106 [0/0]
0x003E9FOC22C8F2CF3EEOC51AFAC8477E4C0820CC66C5800221688A9ED9A15F14 txns:0

```

```

weight:1000.000/7000.000
                                --> 107 [0/0]
0x002F368390850EB0C21E0F06A8863E27FC60554A770202688A06BD4C9EBB6448 txns:0
weight:1000.000/6000.000
                                --> 108 [0/1]
0x00118C1FDE884615937AS68788456995F37817360150053C04A5204F113E7472 txns:0
weight:1000.000/5000.000
                                --> 109 [0/0]
0x003CA183002013808BE0CS086081870207C1C44BFA035463A7SE99366FFS63EE txns:0
weight:1000.000/4000.000
                                --> 110 [0/0]
0x003156208808A2CB774681A29C70A08C5549677828E30F2AA138B2A2C23FDA98 txns:0
weight:1000.000/3000.000
                                --> 111 [0/0]
0x002500503020AE21001083A7EAC2C3018780376AFF04373333E428452SEDD310 txns:0
weight:1000.000/2000.000
                                --> 112 [0/1]
0x001088A860FC123879BFC98484ES0E385748104A277E878E9C26DCD9AS3FSBE7 txns:0
weight:1000.000/1000.000
Cascade Weight :94000.000
Chain Weight :16000.000
Total Weight :110000.000

```

In the minima terminal, a block in the **Cascade** (top section) has the following format:

- **[L_{Curr} , L_{max}]** TxPowID @ blocknumber **weight:** weight of block at L_{Curr} @ timestamp

Where:

- L_{Curr} is the level in the Cascade that the Super Block is currently positioned i.e. its **Current Level**
- L_{max} is the maximum level in the Cascade that the Super Block can reach i.e. its **Super Level**
- **TxPowID:** the hash of the Super Block
- **blocknumber:** the number (height) of the Super Block
- **weight:** The Super Block's weight at its current position i.e. its **Current Weight**

Example:


```
[1/4] 0x0000001062CF82B7735998368D982BD0DAC6B158D596507F6A4FF46E40F946F0 @ 118598
```

```
weight:2.654886E+7 @ Tue Jan 25 23:18:32 GMT 2022
```

Whereas a **(level 0) block on the chain** has the following format:

blocknumber [0, L_{\max}] TxPowID **txns**: number of txns in the block **weight**: *block weight at L_0 /Total weight @ timestamp*

Where:

- **Total weight** is the sum of the weights of the current block and all its child blocks

Example:

```
119839 [0/0] 0x0000012767305A327C2F1B4E8F729B64AACFEFA932443156604E7B6EC845BA3C txns:0
```

```
weight:1.412993E+7/1.412993E+7 @ Wed Jan 26 16:24:40 GMT 2022
```

Attributes 1-3:

The Cascading Chain consists of 32 levels (0-31), with a maximum of 128 blocks at each level.

The Cascading Chain grows logarithmically, as each level is twice as difficult to achieve as the previous level.

Over time, the cumulative sum of the PoW (the 'weight') recorded in the Cascading Chain will tend towards the weight of the chain that would have existed had no blocks been pruned.

Difficulty levels in the Cascading chain

Level 0 is the actual block difficulty. Set to 1 block every 50 seconds,

Level 1 is 2x harder than Level 0

Level 2 is 4x harder than Level 0

Level 3 is 8x harder than Level 0

up to Level 31...

such that **Level L** is 2^L harder than Level 0

Given that each level in the Cascading Chain consists of blocks which are twice as difficult to find as blocks in the previous level, it gets exponentially harder to find a block as the levels increase.

We can calculate the probabilities of finding a block of each level as follows:

Let a block at level L be denoted as $B_L(x)$ where L are the levels $\{0, \dots, 31\}$ and x is the block number (height), then the probability of finding a block at each level is:

Level 0: $P(B_0(x)) = 1/(2^0) = 1$

Level 1: $P(B_1(x)) = 1/(2^1) = 1/2$ i.e. the probability of finding a level 1 block is 1 in 2

Level 2: $P(B_2(x)) = 1/(2^2) = 1/4$ i.e. the probability of finding a level 2 block is 1 in 4

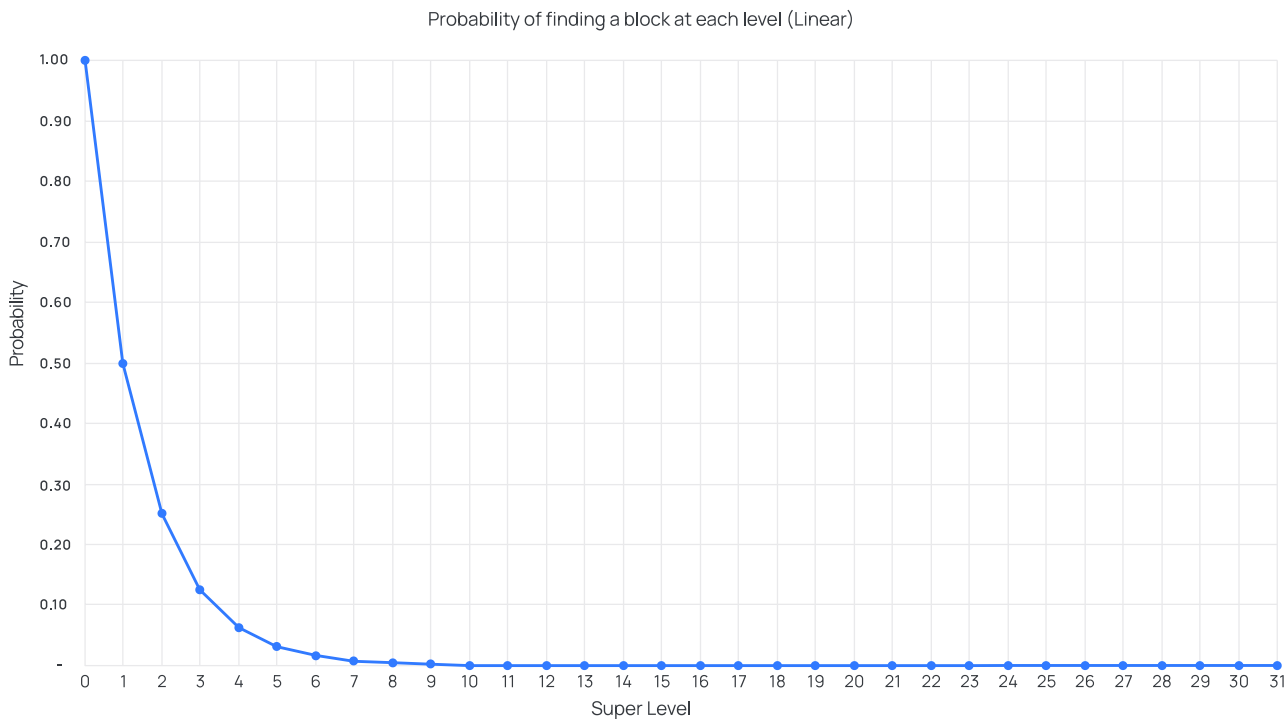
Level 3: $P(B_3(x)) = 1/(2^3) = 1/8$ i.e. the probability of finding a level 3 block is 1 in 8

Level 4: $P(B_4(x)) = 1/(2^4) = 1/16$ i.e. the probability of finding a level 4 block is 1 in 16

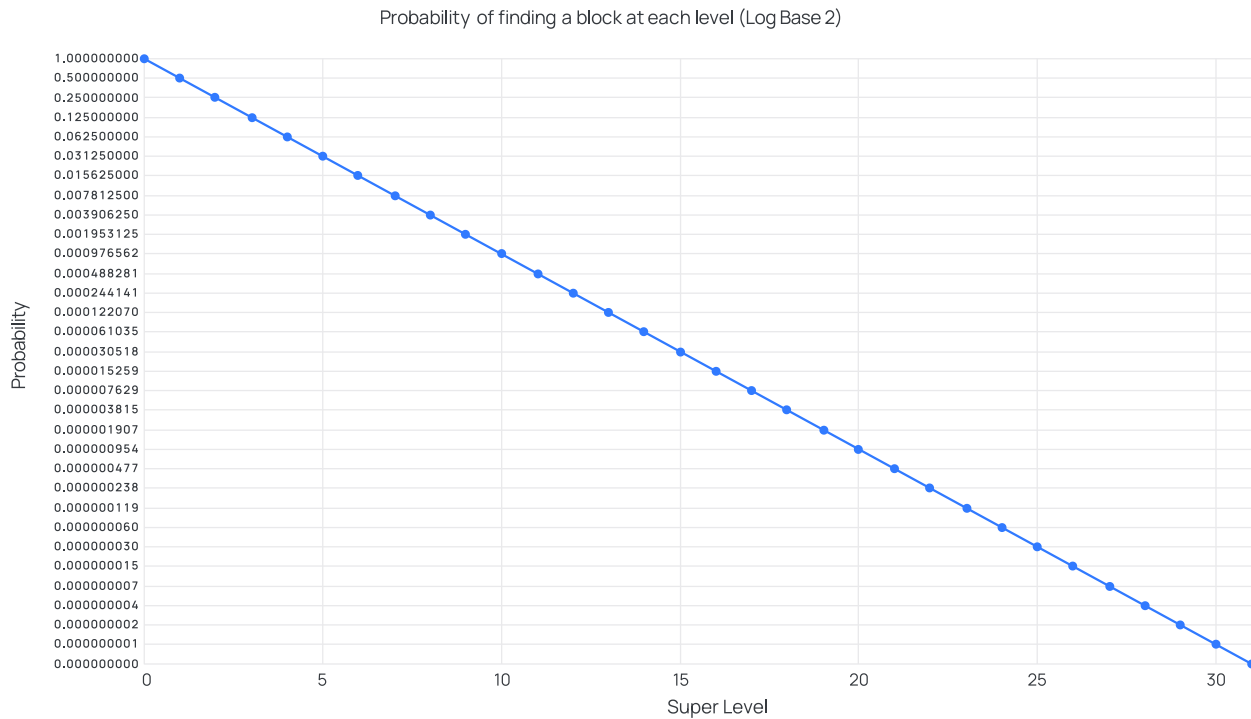
up to Level 31...

such that **Level L:** $P(B_L(x)) = 1/(2^L)$

Probability of finding a block at each level, plotted on a linear chart:



Probability of finding a block at each level, plotted on a (base 2) logarithmic chart:



Relating this to block times means that, on average, it would take the network twice as long to find a level L+1 Super Block compared to a level L Super Block. Therefore, on average, we would expect the amount of time taken to find a block to double with each incremental level.

Expected Super Block times:

Level 0: every 50 seconds (set by the network block difficulty)

Level 1: every 100 seconds (i.e. 50×2^1 or 50×2) (equivalent to finding 2 level 0 blocks)

Level 2: every 200 seconds (i.e. 50×2^2 or 50×4) (equivalent to finding 4 level 0 blocks)

Level 3: every 400 seconds (i.e. 50×2^3 or 50×8) (equivalent to finding 8 level 0 blocks)

Level 4: every 800 seconds (i.e. 50×2^4 or 50×16) (equivalent to finding 16 level 0 blocks)

up to Level 31...

such that the **average time taken to find a Level L Super Block = 50×2^L seconds.**

Hence, on average,

For Levels 0-31:

- One Level L block would contribute the same difficulty (PoW) to the chain as 2^L Level 0 blocks, e.g. One Level 4 Super Block provides the PoW of sixteen Level 0 blocks.

For Levels 0-30:

- There are half as many Level L+1 Super Blocks as Level L
- The sum of the difficulty of Level L+1 Super Blocks = The sum of the difficulty of Level L Super Blocks

Attribute 4:

The Cascading Chain is unbroken. Each block in the Cascade references its previous super parent block in the Cascade.

In every block, there exists a header which contains a reference (hash) to a parent at every existing Super level in the Cascade.

Note: this header is created at the time the block is created and before it has been mined, therefore the maximum level of each new block is unknown when the references to its Super Parents are created.

Each new block in the chain $B_0(n)$ where $n = \text{latest block number}$ contains a reference to the following set of blocks:

- **Its previous block:** $B_0(n-1)$
- **Its set of Super Parents $\{B_s(x_{\max})\}$** where
 x_{\max} is the highest block number (most recent) at any Super level
 s is the Super level of the block $B(x_{\max})$

In other words, when there are multiple blocks at any Super level L, a new block will only reference the most recent level L Super Parent in its header.

The resulting output is that all super blocks in the Cascade will have an immutable link to:

1. Their immediate previous block in the Cascade
2. Previous super blocks in the Cascade which have different maximum levels to their immediate previous super parent.

Terminal output showing Super Parent references

Minima command: **txpow txpowid:**[insertTxPoWID]

The above command will show for the given block, the full details of the TxPoW including its Super Parents.

Example (cropped) showing 5 Super Parents of this block at levels 4, 11, 14, 16, 18 and the genesis block.

```
txpow txpowid:0x0000001062CF8287735998368D9828D0DAC6B158D596507F6A41F46E40F946F0
{
  "command":"txpow",
  "params":{
    "txpowid":"0x0000001062CF8287735998368D982BD0DAC6B158D596507F6A4FF46E40F946F0
  },
  "status":true,
  "response":{
    "txpowid":"0x0000001062CF8287735998368D982BD0DAC6B158D596507F6A4FF46E40F946F0
    "isblock":true,
    "istransaction":false,
    "superblock":4,
    "size":553,
    "header":{
      "block":"118598",
      "blkdiff":"0x01438D45457A6D43AC2268F797C34CEASCFE6AC11986A25821D63DE6806A
      "cascadelevels":32,
      "superparents":[{
        "difficulty":4,
        "count":5,

"parent":"0x0000000F93468B23600B57E89C9C2090A37B8FAA032AE74C7C9992CB4E820249"
      },
      {
        "difficulty":11,
        "count":7,

"parent":"0x0000000028270485D5F55ABEF34A05E4E4CC8ADC18E8B5C6D9235AD7A2C25AF9"
      },
      {
        "difficulty":14,
        count":3,

"parent":"0x00000000035747E54BDA4A1C8FBA709AAFD2488D81FCD0C91E5D14E0F54E459D"
      },
      {
        "difficulty":16,
```

[illegible]

Attribute 5:

At 100 block intervals, the heaviest chain (consisting of all levels in the Cascade and the heaviest branch in the TxPoW Tree) is processed and the Cascading Chain is updated.

Once the main chain (heaviest branch) reaches 1124 ($1024 + 100$) blocks in length, the cascading process begins.

The new Cascade will include a subset of blocks from the existing (previous) Cascade and all of the first 100 blocks from the main chain i.e. the 100 blocks closest to the tip of the existing Cascade. Once added to the Cascade, these first 100 blocks will be pruned from the main chain, leaving 1024 blocks unpruned.

The 101st block in the main chain will become the new root of the TxPoW Tree whose parent will be the tip of the new Cascade.

Before these 100 blocks are pruned, the new root block's MMR Set is updated with entries for all the CoinProofs from these blocks for unspent coins that the node is tracking. Therefore CoinProofs are not lost once the blocks are pruned.

The Cascading process is as follows:

1. **Level 0:** Working backwards through the first 100 blocks in the main chain, each block is checked to see if it meets the difficulty of a Level 0 block. By definition, all blocks are Level 0 so these 100 blocks are all added to Level 0 in the new Cascade. 28 Level 0 blocks from the previous Cascade will remain in Level 0 of the new Cascade, filling the 128 spaces at this level.
2. **Level 1:** After 128 blocks have been added to Level 0 of the Cascade, continuing to work backwards through the remaining Level 0 blocks in the previous Cascade, these will only be kept and added to Level 1 in the new Cascade if they meet the difficulty required to be a Level 1 Super Block or above, otherwise they are pruned.
3. **Level 2:** Once, and if, 128 Super Blocks have been added to Level 1, the next Super Blocks must meet the Level 2 difficulty to remain in the Cascade, otherwise they are pruned.
4. **Level 3:** Continuing to work backwards through Super Blocks the previous Cascade, the next 128 blocks added to the Cascade must meet the difficulty required for at least Level 3, otherwise they are pruned.
5. The process continues until all blocks in the chain have been processed.

This results in a new Cascading Chain and the remaining (most recent) 1024 blocks kept, in full, on the main chain.

The weight of the Super Blocks on the Cascade are also recalculated as follows:

A Super Block's Current Weight (CW) is its 'Base Weight' (BW) multiplied by a factor proportional to the current level the Super Block is positioned (irrespective of its maximum Super Level):

CW of Super Blocks on Level 0 = $BW * 2^0$ *i.e. Base weight*1*

CW of Super Blocks on Level 1 = $BW * 2^1$ *i.e. Base weight*2*

CW of Super Blocks on Level 2 = $BW * 2^2$ *i.e. Base weight*4*

CW of Super Blocks on Level 3 = $BW * 2^3$ *i.e. Base weight*8*

CW of Super Blocks on Level 4 = $BW * 2^4$ *i.e. Base weight*16*

up to Level 31...

such that $CW(B_L(x)) = BW * 2^L$, where L is the current level a Super Block is positioned on.

Example: if a block has a base weight of 1000, and it currently sits at Level 4, the block would weigh $1000 * 2^4 = 16000$

Mining and Consensus

Magic Numbers

The Magic numbers provide a mechanism for future-proofing the network. As technology improves over time, increasing the storage and processing capacity on mobile devices, the Magic numbers allow Minima to adapt or grow simultaneously.

There are two sets of four variables that define the network, for each variable there is the **Current** network value and the user's **Desired** value. The Current value dictates the network at that point in time, the value that the whole network is currently working to. Desired values can be specified by users to reflect the capability of their node. Note: Desired values must take a value that is at least half of the corresponding Current value, and not more than double the corresponding Current value.

The Current Magic numbers are recalculated every block by taking a heavily weighted average of 16383:1 in favour of the Current network value over the node's Desired value. Provided the entire network agrees on a new desired value, over a period of approximately 50 days, the Current Magic value will converge to the Desired value.

Magic Number	Description	Default Value	Type
CurrentMaxTxPoWSize	The maximum size of a TxPoW unit in bytes.	The default and minimum TxPoW size is 64KB	MiniNumber
CurrentMaxKISSVMOps	The maximum number of KISS VM operations in a TxPoW (script complexity)	The default and minimum is 1024	MiniNumber

Magic Number	Description	Default Value	Type
CurrentMaxTxnPerBlock	The maximum number of transactions per block	The default and minimum transactions per TxPoW unit is 256	MiniNumber
CurrentMinTxPoWWork	The target value for the hash of a TxPoW header, that must be met before for a TxPoW unit to be sent across the network	The minimum is equivalent to 1 million hashes/second	MiniNumber

Difficulty

Difficulty is a dynamic value which determines how hard it is to mine a TxPoW unit. The difficulty value is used to calculate a target value which the hash value of the TxPoW header (the TxPoW ID) needs to be less than.

Target Value = Max Value/Difficulty Value

Where

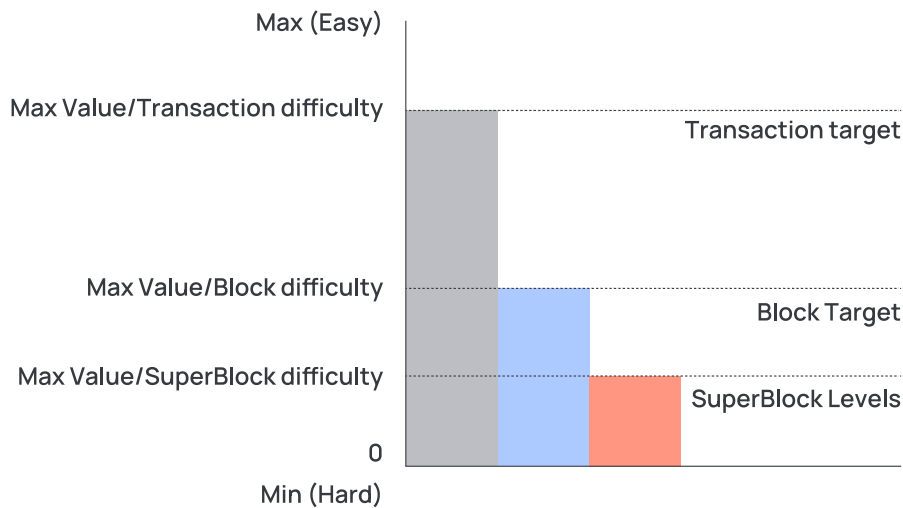
Max value (represented in Hex) =

FF

Max value (represented in Decimal) = 1.15792089237316E+77

In Minima, there are multiple difficulty levels:

Diagram: Basic representation of the different difficulty levels, where transaction target is the easiest to achieve (not to scale)



NOTE

The higher the difficulty, the lower the target value, the harder it is to achieve.

Transaction (TxPoW) Difficulty

The default and minimum difficulty value is 1 million, meaning that a device with a hash rate of 1m hashes/second will need to perform 1 second of 'work' before the target is met and their TxPoW can be propagated across the network, entering the mempool of unconfirmed transactions.

The default and easiest TxPoW Target = $\text{Max Value} / 1,000,000$

Target represented in Hex =

0x10C6F7A0B5ED853E638E9803F452E932EB00EAD38965A800000000000000

Target represented in Decimal = $1.15792089237316E+71$

This is the magic number **CurrentMinTxPoWWork**.

The actual transaction difficulty for a node considers the achievable hash rate of the node and is equal to the higher value of 1 million or the device's hash rate in hashes/second, so that each device performs roughly 1 second of 'work' to 'mine' their transaction (TxPoW unit).

Block Difficulty

The purpose of the block difficulty level is to regulate the frequency at which blocks are found on the network ensuring block intervals remain as close to 50 seconds as possible. TxPoW units only require the minimum amount of work before they are sent across the network, however these TxPoW units can become blocks and get added to the chain if, by chance, the TxPoW ID hash is also less than the network block difficulty target.

The block target is much lower (harder) than the transaction target, and fluctuates with the number of nodes on the network and their hash rate, so that 50 block intervals are maintained.

If block times deviate from 50 second intervals, the difficulty will be adjusted proportionally, by a maximum of 10% up or down.

The block difficulty will always be at least as hard as the transaction difficulty.

Adjusting the block difficulty:

Example 1: Block times too slow

If the time interval between blocks slows down to 1 block every 54 seconds, the current block difficulty will be decreased by a factor of $50/54 = 0.926$.

This lower difficulty increases the block target value, making it easier for nodes to achieve and acting to speed up the interval between blocks.

Example 2: Block times too fast

If the time interval between blocks speeds up to 1 block every 44 seconds, the current speed ratio (required block time/actual chain speed) is $50/44 = 1.25$, however since this is over 10%, the block difficulty will only be increased by the maximum 11.

This higher difficulty decreases the block target value, making it harder for nodes to achieve and acting to slow down the interval between blocks.

Once the transaction (not block) difficulty target has been met, the mining process finishes. If the block difficulty target has by chance also been achieved, then the TxPoW unit will become a block and be added to the main chain.

Superblock Difficulty

[\(see Cascading Chain\)](#)

Block Weight

A block's base weight is equal to the difficulty value that was required for it to become a block. This also represents the average number of hashes that would be required to meet this block target value.

For a specific block:

Base Weight = Max Target Value/Block Target Value

Note that this is not the actual weight, which would be the actual number of hashes it took to find a value less than the target.

If the actual weight, by chance, exceeds its base weight by at least a factor of 2, the block will be considered a Super Block [\(see Cascading Chain\)](#).

Constructing TxPoW Units

TxPoW units are created after a transaction and its witness has been created. Once created, the TxPoW will be 'mined' and propagated across the network.

The following Terminal commands will result in creation of a TxPoW unit:

send - send a transaction

tokencreate - create a custom token

txnpost - posting a manually constructed transaction

A TxPoW unit is constructed as follows:

1. Create the TxPoW

Set details

- Block height (if this TxPoW becomes a block)
- Timemilli (the node's current time)
- Main transaction & Witness (the transaction the user is sending)

- Burn transaction & Witness
- Calculate & set the magic numbers

2. Set the Cascade Super Parents

If the current tip block is Super Level i , this TxPoW will have:

Super Parents for Levels 0- i : Current tip block

Super Parents for Levels i -31: same Super Parents as the current tip

3. Set the Transaction difficulty

The Hex value that the hash of the TxPoW must be less than to be valid, is set based on the hash rate of the node. It must be no easier than the Magic number **CurrentMinTxPoWWork**.

```
Minimum TxPoW hash = Max Value / user's hash rate
```

```
If
  Minimum TxPoW hash > CurrentMinTxPoWWork i.e. if the Minimum TxPoW hash is easier
than CurrentMinTxPoWWork
then
  Minimum TxPoW hash = CurrentMinTxPoWWork
```

4. Calculate Chain Speed and Block Difficulty

Using the latest 256 blocks in the main chain, the average time and block difficulty for each block is calculated and the block difficulty target is adjusted proportionally by a maximum of 10% up or down.

```
Initial Start Position = tip block, Initial End Position = tip - 256
```

The two blocks with the median time from the 32 blocks prior to the initial start and initial end positions are determined. Those blocks become the final start and end positions to calculate the current Chain Speed and Block Difficulty. Using the median smooths out any anomalies in the system time of nodes on the network.

```
Chain speed (secs) = Avg time interval per block between the final start and end
position
```

Speed Ratio = Required block speed (50 secs)/Chain speed

Average block difficulty = Avg difficulty for blocks between the start and end position

Then,

New block difficulty = Avg block difficulty * Speed Ratio
(if $0.9 \leq \text{Speed Ratio} \leq 1.1$)

or

New block difficulty = Avg block difficulty * 0.9
(if Speed Ratio < 0.9)

or

New block difficulty = Avg block difficulty * 1.1
(if Speed Ratio > 1.1)

or

New block difficulty = CurrentMinTxPoWWork
(if Avg block difficulty * Speed Ratio < CurrentMinTxPoWWork and $0.9 \leq \text{Speed Ratio} \leq 1.1$)

i.e. the Magic number **CurrentMinTxPoWWork** is used as a lower bound for the block difficulty

5. Order Mempool Transactions

Sort the unconfirmed mempool TxPoW units by the amount they burn.

6. Check Mempool Transactions

Cycle through the mempool transactions checking the following:

- Coins are not already added to this TxPoW
- Transaction size is less than or equal to the magic number **CurrentMaxTxPoWSize**
- Transaction difficulty must be harder than or equal to the magic number **CurrentMinTxPoWWork**
- MMR Proofs are valid
- Script Proofs are valid

7. Add valid Mempool TxPoWs

Create a list of valid mempool transactions, including the maximum allowed by the magic number **CurrentMaxTxnPerBlock**. This list is added to the body of the TxPoW unit. Invalid TxPoW are removed from the mempool.

NOTE

Only the TxPoW IDs of mempool transactions are added to this list.

8. Calculate hashes for main and burn transactions

Calculate the hash of the main transaction and the burn transaction, add these to the TxPoW body. These are required for creating the Coin IDs of the transaction outputs and the MMR.

9. Construct the MMR

Construct an updated MMR set with proofs for the newly created coins (UTxOs). Calculate the new MMR root hash and the root value.

10. Add the MMR Root data

Add the MMR root hash and value (sum of all coins in the network) to the TxPoW header.

The TxPoW generation process is complete and is ready to be 'mined' before being sent on to peer nodes for propagation across the network.

Mining TxPoW units

Once a new TxPoW unit has been created, it must be 'mined' before a node can send it on to its peers.

Mining is the process of repeatedly hashing the TxPoW header, each time with a different number (known as the nonce value) set in the header. By changing the nonce each time, the header data changes, resulting in a different hash.

This process is repeated until the resulting hash is less than the transaction difficulty target. Once this is achieved, the required difficulty level has been met and the TxPoW has enough 'Proof of Work' to be propagated to other nodes in the network.

i NOTE

A node does not consider the block difficulty target during the mining process.

Only after a TxPoW has been mined, is it evaluated to see if it will become a block - if the TxPoW ID hash is also, by chance, lower than the block difficulty target.

A TxPoW unit is 'mined' as follows:

1. Calculate TxPoW body hash

Hash the TxPoW body and set in it the TxPoW header

2. Set the initial nonce value

Set the start nonce value in the TxPoW header

3. Hash - Check - Set

The TxPoW header is hashed, if it does not satisfy the transaction difficulty, the nonce is incremented by 1 and the process repeats.

Once the hashed TxPoW header satisfies the transaction difficulty, the final nonce value is set in the TxPoW header

4. Calculate the TxPoW ID & Size

The hash of the TxPoW header is the TxPoW ID. If it also meets the block difficulty target, its Block weight and Super Level are also calculated.

Once the mining process has ended, the node will continue to validate, process and send the TxPoW ID across the network.

Validating TxPoW units

Once a user has successfully mined a TxPoW unit or when they have received a TxPoW unit from a peer, it must be validated before propagating to peers.

The potential outcomes from this check are:

Outcome 1: The TxPoW is invalid - disconnect from the client who sent it and discard the TxPoW.

Outcome 2: The TxPoW is not fully valid - some check(s) did not pass but it could be valid at a future time. Remain connected to the client, keep and attempt to process the TxPoW but do not send it on to peers.

Outcome 3: The TxPoW is fully valid - remain connected to the client, keep and attempt to process the TxPoW but do not send it on to peers.

Validating a TxPoW unit includes checking:

- Does it already exist in the TxPoW database?
- *if false, request it from the peer*
- Does it have a block number before the root of the TxPoW Tree?
- *if true, outcome 1.*
- *Reason: this TxPoW is before the Cascade tip*
- Is the block difficulty at most 10% below the tip block?
- *if false, outcome 1.*
- *Reason: the block difficulty is too low*
- Does the Chain ID match the Chain ID of the current network?
- *if false, outcome 1.*
- *Reason: the Chain ID is wrong*
- Does it meet basic transaction checks?
- *if false, outcome 1.*
- *Reason: the TxPoW fails basic checks (see Basic TxPoW Checks)*
- Are the signatures valid?
- *if false, outcome 1.*
- *Reason: the signatures are invalid*
- Are the coin and token scripts valid?
- *if false & if the transaction is monotonic, outcome 1 but remain connected.*
- *Reason: monotonic TxPoWs will always be invalid if they fail script checks*

- *if false & if the transaction is non-monotonic, outcome 2.*

- Reason: non-monotonic transactions are not fully valid at this point in time but are still processed in case they rely on a certain block time to be valid

- If the TxPoW is a block, does it have a timestamp less than 2 hours in the future?
 - *if false, outcome 2.*
 - *Reason: the TxPoW is over 2 hours in the future but not discarded as could be something wrong with internal clock*
- Does it use any coins as transaction inputs that are also currently in the mempool?
 - *if true, outcome 2.*
 - *Reason: the same coins are used in another transaction but the TxPoW is not discarded as it could be valid in another branch*
- Do the CoinProofs for the main and burn transactions pass the MMR checks?
 - *if false, outcome 2.*
 - *Reason: MMR proofs are not valid, but the TxPoW is not discarded as they could be valid in another branch*
- Did it take longer than 1 second to process?
 - *if true, outcome 2.*
 - *Reason: the message took a long time to process*

Fully Valid and not Fully Valid TxPoWs are then added to the TxPoW database. Only a Fully Valid TxPoW is then forwarded onto the node's peers.

If a received TxPoW unit is a block, all the mempool TxPoW in the block's *txn list* as well as the parent block must exist in the TxPoW database before it can be added to the TxPoW Tree. If the node is missing any, these are requested from peer nodes before processing.

Basic TxPoW checks:

The Basic TxPoW checks ensure the main transaction and its burn transaction are valid.

The basic checks are:

- The Chain ID must be valid

For main transaction and burn transaction:

- The Link Hash connecting the main and burn transaction must be valid

- Check if the transaction is empty, empty transactions are valid (e.g. a Pulse TxPoW)
- Transaction inputs & outputs are valid
- Must be at least one input
- The number of MMR proofs must be correct

For all coins in the main transaction:

- No coins can be used more than once
- If a custom token is being spent, the Token ID must match the Token ID in the coin and in the MMR Proof
- CoinProof details must match the Coin details (amount, address & Token ID)
- The Coin ID of non-floating coins must match the Coin ID in the CoinProof
- The Coin must not be spent
- The Coin must have script proofs

Processing TxPoW & Assembling the Chain

The final step, once a TxPoW passes all the validation checks, is to push it to the end of a process stack so that it can be added to the TxPoW tree in the correct order.

The node will process the stack, attempting to add any TxPoW units that are blocks to the end of the chain. A block's parent must exist in the tree before it can be added. If the parent does not exist in the tree yet, the node may not have had all its transactions when it first attempted to process it. Since last attempting to process the parent, the node should have received any missing transactions from its peers so it is searched for in the TxPoW database and, if found, added to the process stack to be processed before the child.

If the parent block is already in the tree, the TxPoW is processed by first ensuring that all mempool transactions in the *txn list* of the TxPoW body exist in the user's TxPoW database.

- If there are any missing, the TxPoW cannot be processed any further.
- If there are none missing, the TxPoW will go through final checks before becoming a TxBlock.

A **TxBlock** includes the TxPoW and adds the MMR peaks; a list of CoinProofs for all the coins that will become spent; and a list of the new coins that will be created from all the transactions in the block. ([see TxBlocks](#))

Once the TxBlock has been created, a structure is required to attach the block to the TxPoW tree, this is the role of a TxPoW Tree Node. It adds further relevant information to the TxBlock i.e. the MMR set; a list of all the spent and newly created coins; and a list of all the MMR entry numbers of any coins in any of the transactions that are relevant to the user (i.e. coins they are tracking). ([see TxPoW Tree Nodes](#))

A child node is then added to its parent in the tree and the tree is **recalculated**.

Recalculating the tree involves:

- Calculating the weight of all blocks in the tree
- Selecting the heaviest branch as the main chain ([see Selecting the main chain](#))
- Cascading the chain if the heaviest branch has reached the required length ([see Cascading Chain - Attribute 5](#))
- Setting transactions in the main chain so they cannot be added to a new TxPoW

The TxPoW database is then checked to see if there are any children for this new block as these can then be added to the process stack for processing.

This process continues until either the process stack is empty or contains TxPoWs with transactions not yet received by the user.

Selecting the main chain (GHOST)

The TxPoW Tree consists of a main chain and branches which occur naturally due to network latency. All nodes in the network must only consider one chain to be the valid one at any point in time - the main chain. The tip of the main chain becomes the parent block that a node will attempt to build on when generating a TxPoW unit.

Minima uses the **GHOST (Greedy Heaviest Observed SubTree)** protocol to ensure that nodes can come to consensus on which chain to use as the main chain. GHOST dictates that the 'heaviest' branch should be the main chain. The 'heaviest' branch is the branch which has had the most Proof-of-Work put into it and may not necessarily be the longest chain.

An alternative to GHOST is the simple 'Longest chain' rule, where the valid chain is considered the one with the most number of blocks in it, however by using GHOST and considering the 'weight' of

blocks rather than simply the number of blocks, the chain is more resilient to attack and allows for faster block times.

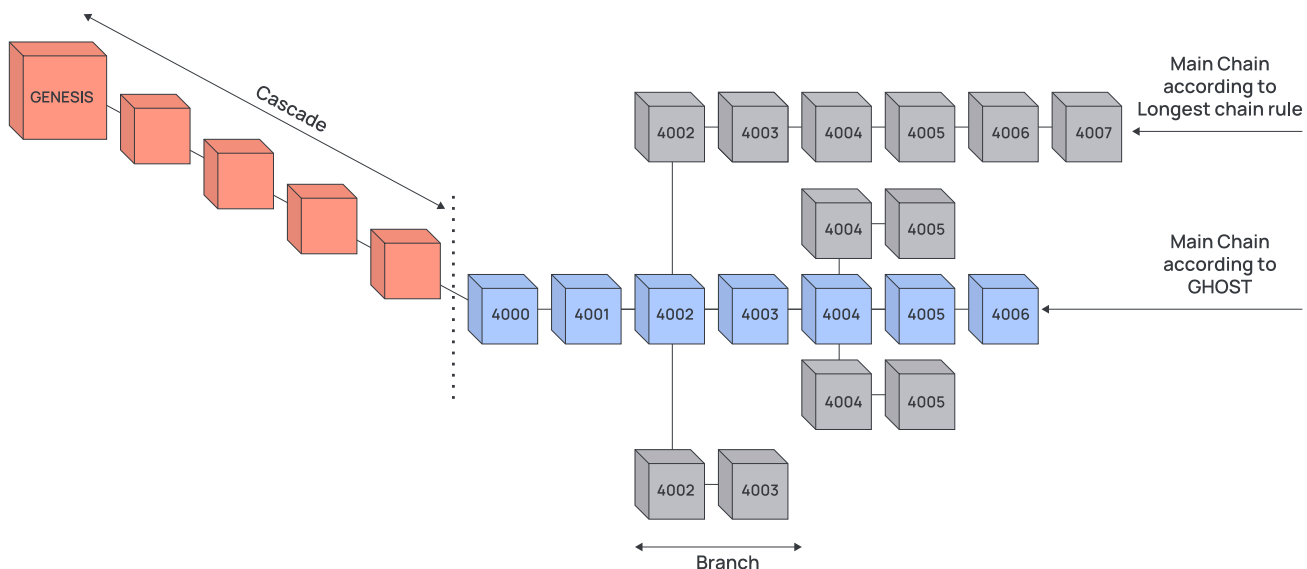
The weight of a block is equal to the sum of the base weights of its children, where the base weight = the block difficulty (see [Difficulty](#)), so where there are siblings in the tree that create two branches, their parent block will have a total weight of the sum of the siblings.

The process of selecting the main chain (i.e. recalculating the tree) occurs after receiving and processing a TxPoW unit (whether or not it is a block). During this process, the weight of all blocks in the TxPoW tree (main chain and branches) are evaluated and the heaviest branch is set as the main chain. All transactions in the blocks of this chain are then considered as 'truth' and the node continues to build from the tip of this chain. This may mean some transactions are returned to the mempool and must be added to a new TxPoW unit before being confirmed.

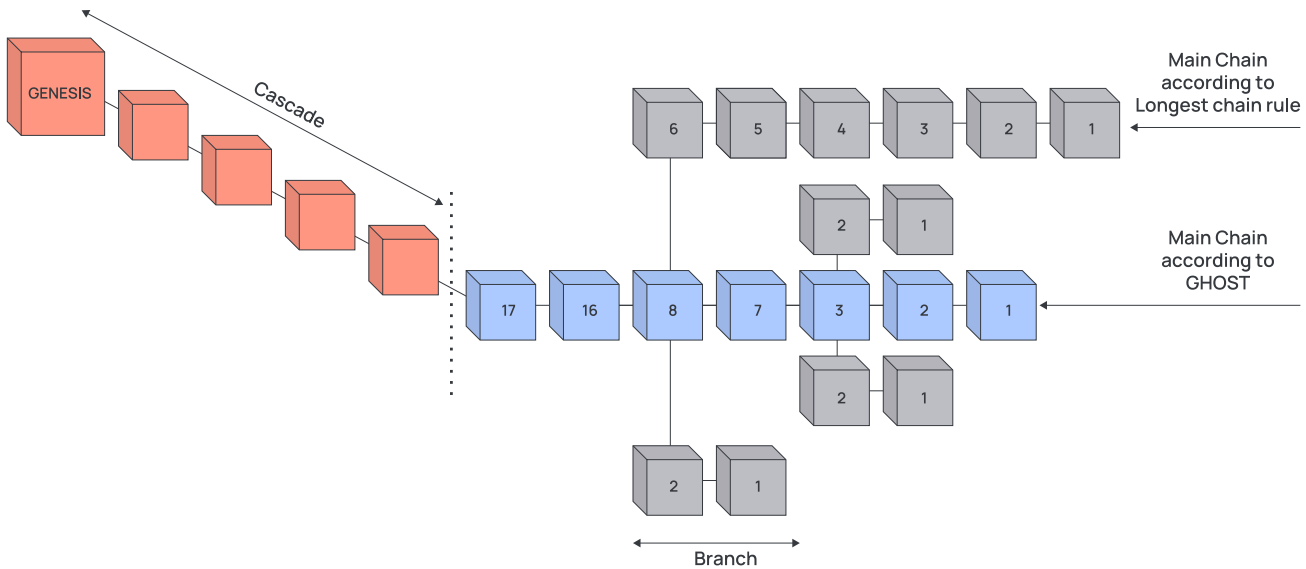
Diagram illustrating the main chain according to GHOST.

The longest chain could more easily be achieved by an attacker in secret, however with the majority of the network constructing on the heaviest chain, the attacker's secret chain would not change the main chain.

Cascading Chain & TxPoW Tree showing main chain according to GHOST (with Block numbers)



Cascading Chain & TxPoW Tree showing main chain according to GHOST (with Block weight = 1)



NOTE

A block weight of 1 is for illustrative purposes only, actual block weights vary depending on the block difficulty of the network. (see [Block Weight](#))

TIP

GHOST was originally proposed as an alternative to Longest Chain by Sompolinsky and Zohar in the paper *Secure High-Rate Transaction Processing in Bitcoin*.

Coloured Coins

Minima is the native coin for the Minima blockchain. Each UTXO is defined as a coin and therefore a coin can be worth any amount of Minima. Minima supports custom tokens (including NFTs) natively. Tokens are **Coloured coins**. Coloured coins are tiny fractions of Minima which represent the supply of a custom token or NFT.

Custom tokens are specified by the following characteristics:

[illegible]

NFTs are simply custom tokens with no decimal places i.e. they can only be spent whole.

Keys and Digital Signatures

Public-Private key pairs, combined with digital signatures, provide the mechanism required to hold coins securely and to independently prove ownership of them. Keys are held and controlled by a user's wallet.

A private key is intended to be known only to the owner of the coins, whereas the corresponding public key can be shared without risk of compromising the coins and is used to receive funds which then become secured by the associated private key.

NOTE

You can think of a public key as being your bank account number and the private key as your PIN number.

Generating Public-Private Key Pairs

An essential property of Public-Private key pairs is that the private key should not be deducible from the public key. Various cryptographic algorithms are available for generating a public key from a private key, for example RSA or Elliptic Curve Cryptography. These are one-way functions which, using complex mathematics, ensure that the private key cannot be deduced from the public key.

Digital Signature Schemes

Using a public-private key pair and a digital signature scheme, a user can digitally 'sign' some data e.g. a transaction, with their private key to create an unforgeable digital signature. Digital signatures are generated by applying the algorithm of the signature scheme to the private key and some data. Anyone can independently verify the validity of a signature knowing the user's public key, the data, and the check algorithm of the digital signature scheme.

Minima uses the **Winternitz One-Time Signature Scheme (WOTSS)** as its digital signature scheme which applies one algorithm for generating public keys from a private seed and another to compute a digital signature, with a given piece of data to be signed. **Winternitz is considered to be Quantum-resistant.**

NOTE

Not all Digital Signature Schemes are considered Quantum-resistant, for example RSA and Elliptic Curve Digital Signature Algorithm (ECDSA), used in Bitcoin and Ethereum. In the 90's, Shor published a Quantum algorithm which could be used to break these schemes, rendering them vulnerable to attack from Quantum-based computers in the future.

Merkle Signature Scheme

The **Merkle Signature Scheme (MSS)**, originally proposed by Ralph Merkle in the 70's, combines a Quantum-resistant, hash-based, but **one-time-use**, digital signature scheme with hash trees.

A **one-time-use signature scheme** means that each public-private key pair can only be used once, securely, to sign some data. Reusing the same key pair for multiple signatures increases the chances of the private key being deduced. To mitigate this inconvenience, many single-use key pairs can be stored as leaf nodes in a hash tree, with the root hash of the tree used as a **multiple-use root public key**.

Minima uses a Merkle Signature Scheme by combining the **Winternitz One Time Signature Scheme (WOTSS)** with **Merkle Mountain Range (MMR) hash trees**. Winternitz is used to generate private/public key pairs and signatures which are stored as leaf nodes in an MMR, creating a Tree of Keys.

NOTE

The cryptographic hash function used in Minima's implementation of the Merkle Signature Scheme is **SHA3-256**, which is considered Quantum-resistant. It takes an input message and produces a 256-bit message digest, from which the input message cannot be determined.

NOTE

Minima also uses MMR trees for storing a user's coins. See [MMR](#).

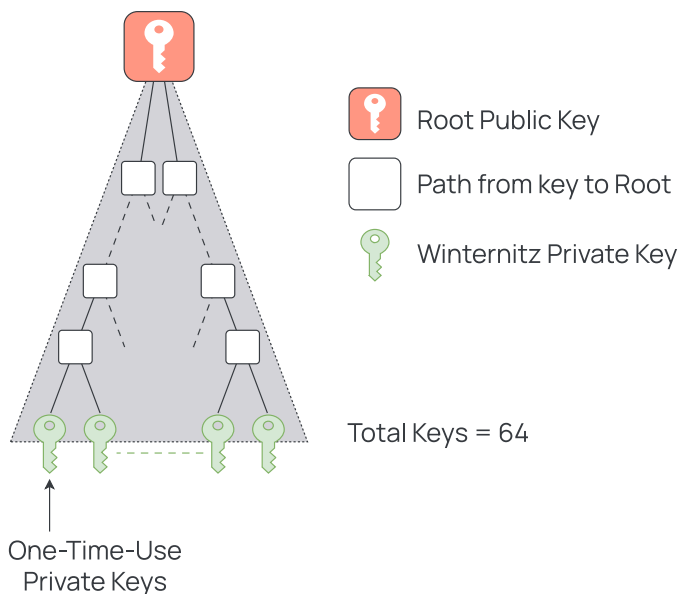
Trees of Keys

A Tree of Keys enables a user to have many **secure but one-time-use private keys** all associated with the same root public key. This is useful because it allows a user to share a single public key for receiving funds but to sign each transaction with a different private key, ensuring maximum security.

This **root public key** can be used securely, the same number of times as there are private keys associated with it, i.e. the number of leaf nodes in the tree. By signing with a different private key for each transaction, and presenting a proof path with the signature which indicates the path through the MMR tree from the private key to the tree root, any external party can validate that the signature was generated by the rightful owner of the funds.

Example of a single Tree of Keys

A TreeKeyNode is a single MMR Tree with (a default) 64 single-use Winternitz public-private key pairs and a Root public key



Each leaf node (Winternitz Key Pair & Signature) is generated using:

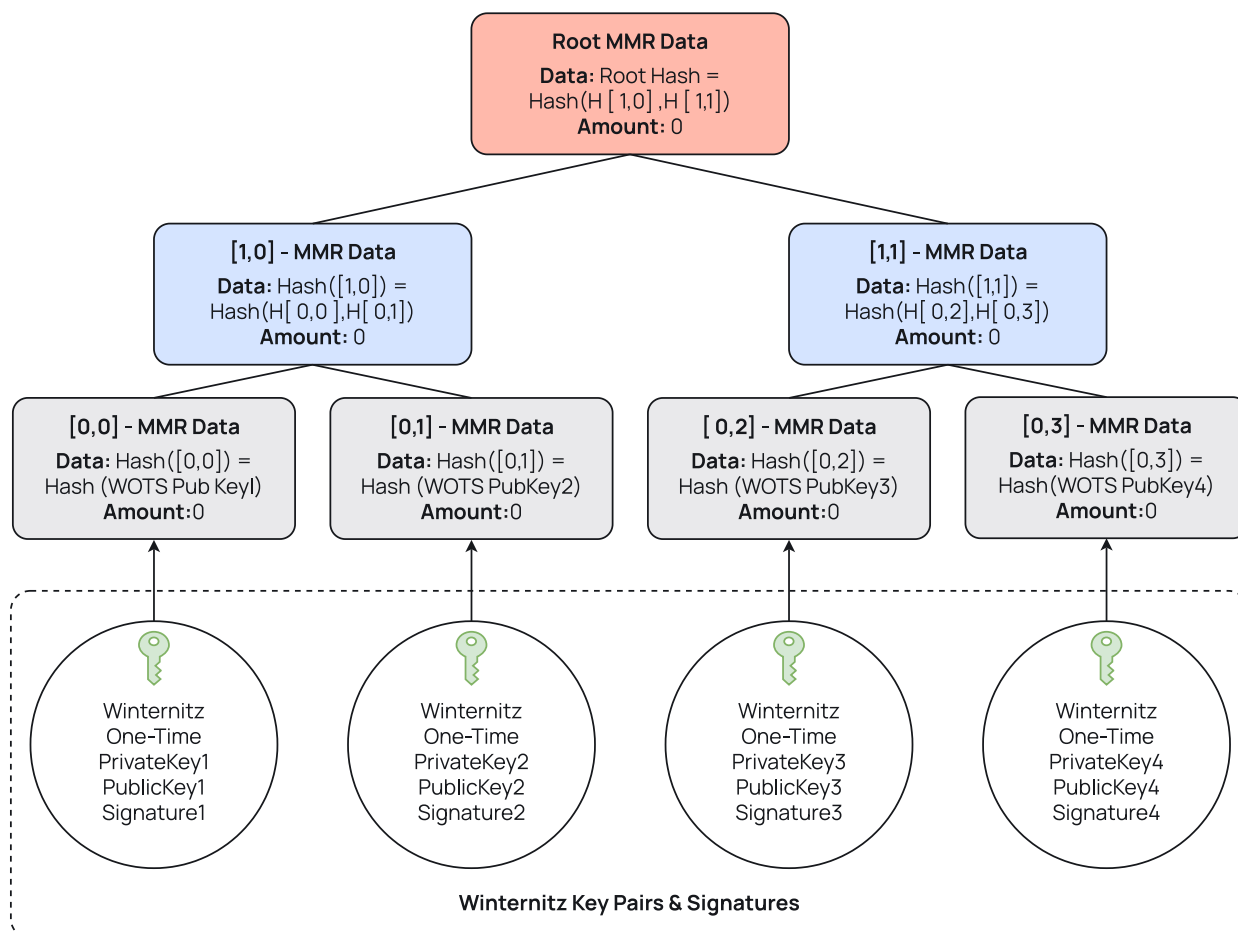
- **a private seed** - this is generated by concatenating a number from 0-63 with the private seed of this TreeKeyNode. i.e. $\text{Hash}(i, \text{PrivateSeed})$ for $i = 0-63$.
- **a hash function with specified digest size** - Minima uses the SHA3 hash function with a 256 bit digest.
- **a chosen Winternitz parameter** - Minima uses a Winternitz parameter of 8.

To find out more about Winternitz security, see <https://eprint.iacr.org/2011/191.pdf>

Therefore each leaf node corresponds to a Winternitz Key Pair and Signature:

WOT Signature Scheme	Description
WOTS Private Key	Single use Winternitz private key
WOTS Public Key	Single use Winternitz public key
WOTS Signature	The one-time signature (of a given message/transaction) generated with the private key

Example TreeKeyNode with just 4 leaf nodes:



Once all 64 keys are generated, the root hash can be calculated. Clearly a public key which can only be used securely 64 times would not be sufficient. To get more uses from a single root public key,

there needs to be more keys (leaf nodes). However, the more Winternitz keys that exist, the longer it takes to generate them and the longer it takes to generate the root hash i.e. the root public key.

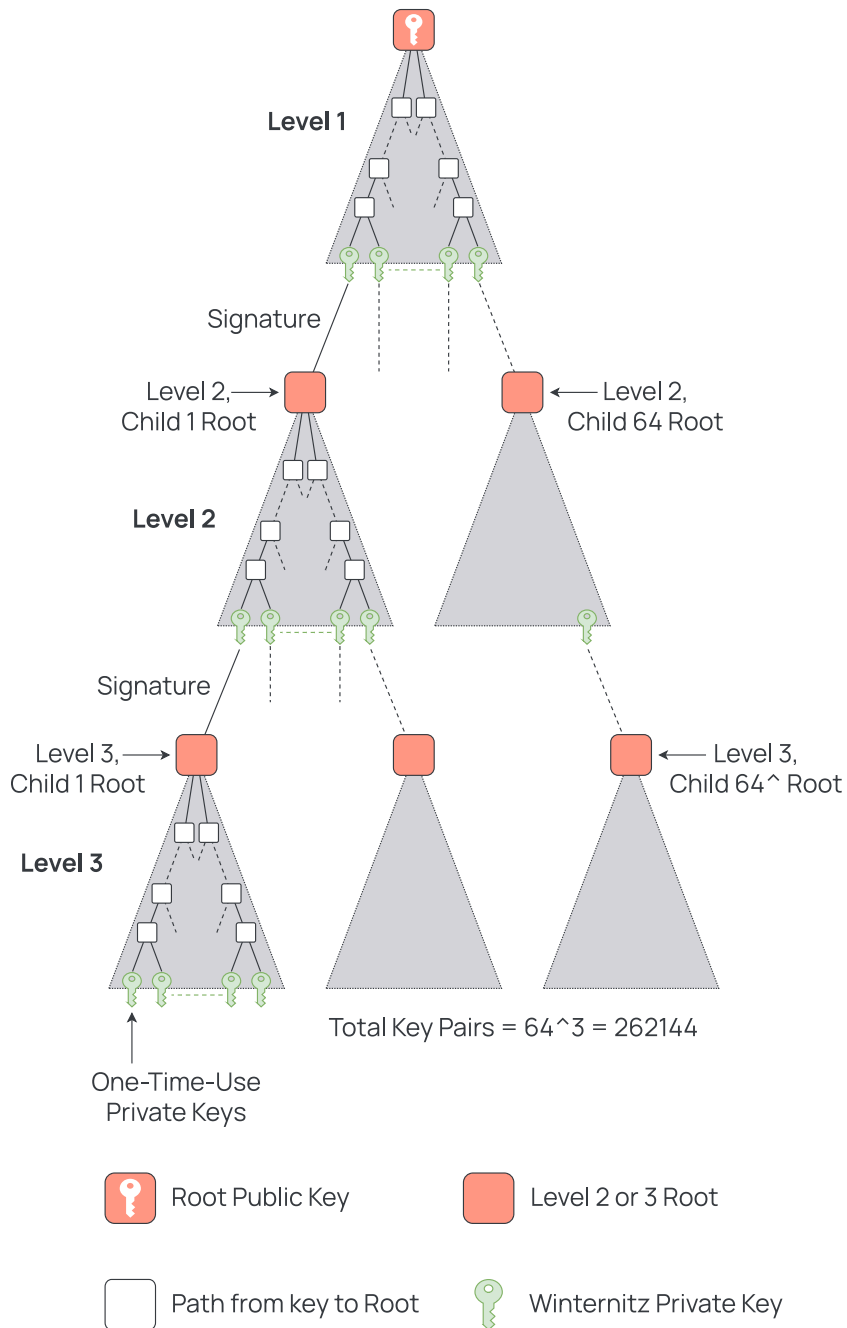
In order to efficiently scale the number of uses possible for a root public key, instead of simply generating a single large Tree of Keys with hundreds of thousands of leaf nodes, Minima constructs a **Tree of Trees** with multiple levels and a single top root.

A **Tree of Trees** consists of (a default) 3 levels where the root of all level 2 and 3 trees are signed by a Winternitz key pair in the level above (as shown below). The level 3 trees contain the Winternitz keys which are used to sign transactions.

Each individual Tree in Level 2 is connected to a leaf node key on Level 1 and will have its own **private seed**, generated by hashing the Level 1 key number (0-63) with its private key. Likewise, each Level 3 tree is connected to a key from a Level 2 tree.

The private keys from level 1 and 2 are used to sign the root hash of the level 2 and 3 trees respectively, creating a Tree of Key Trees, connected through signatures.

Diagram showing a full Tree of Trees (with default 3 levels & 64 keys/tree)



With each individual MMR tree containing 64 keys as leaf nodes; adding a second level of MMR trees provides 64^2 WOTS public-private key pairs.

The MMR tree grows logarithmically, with n levels providing a maximum of 64^n one-time-use key pairs for the user to sign transactions with.

Minima uses a default 3 levels, providing a total of $64^3 = 262144$ one-time-use keys.

The advantage of a 'tree of trees' over a single large tree is that:

- The user's **root public key** is calculated efficiently - only requiring the top (level 1) tree to be generated;
- Each level of trees is connected via digital signatures which can be independently verified
- Each level 3 tree can be added on demand as the user requires more private keys for signing transactions.

Single Tree (TreeKeyNode) properties

TreeKeyNode Property	Description	Type
Size	The number of leaf nodes in this MMR Tree. Default is 64.	Integer
Tree	The MMR Tree structure of this TreeKeyNode	MMR
Children	An array of the child Trees belonging to this Tree (default 64 for each level 1 and 2 tree, 0 for level 3 trees)	TreeKeyNode array
Keys	An array of the Winternitz Keys added as leaf nodes to this Tree (default 64).	Winternitz Keys array
ChildSeed	The hash of the Private Seed that was used to generate this Tree i.e. Hash(PrivateSeed). This child seed will be used as the base to generate the private seeds for each child Tree (for level 1 & 2 trees only)	MiniData
PublicKey	The root hash of the tree. If this tree is Level 1, this will be the user's root public key.	MiniData
ParentChildSig	The signature generated when the parent tree signed the root of this Tree (levels 2 and 3 only)	Signature Proof

Tree of Trees (TreeKey) properties

TreeKey Property	Description	Type
Root	The top tree of this tree of trees, generated from the user's Base Private Seed.	TreeKeyNode
Levels	Number of levels of trees in this tree of trees (default 3)	Integer
KeysPerLevel	The number of keys per single tree (default 64)	Integer
Uses	The number of times the root public key has been used	Integer
Max Uses	The maximum number of uses = $(\text{Keys/level})^{\text{number of levels}}$. Default is 64^3 .	Integer
PrivateSeed	The PrivateSeed used to generate all the trees in this TreeKey.	MiniData
Public Key	The root hash from the root tree.	MiniData

Each TreeKey requires a private seed from which all the leaf node private keys are generated. This is covered in more detail next.

Constructing a Tree of Trees

On creation of a new Minima node, a 32 byte **Base Private Seed** is created using a Pseudo Random Number Generator (PRNG). From this base private seed, further private seeds are generated by combining it with a **modifier** (another 32 byte random number).

Each of these **modified private seeds** is used to create a Tree of Trees, the root of which becomes one of the user's multiple-use root public keys.

Executing the **keys** command shows us the **root keys** that have been created:

```
keys
{
```



```
"command": "keys",
"status": true,
"response": [{
  "size": 64,
  "depth": 3,
  "uses": 0,
  "maxuses": 262144,
"modifier": "0x02984CB232D0C003F6681980689F45BA255522131882E1530D393518401A6CF8",
"publickey": "0x9F9FBFD83D999D952BE4A6538252043987F3937F3BBC361F00D5AE708EF1A105",
"privatekey": "0x28AF0DD826C1D49A74F6533920AFBCE5D2044AA822591B389E4A4518C483E672"
},
```

Public Keys, Scripts and Addresses in Minima

All 0x addresses in Minima, which can be shared publicly to receive funds, are **Pay-to-Script-Hash (P2SH)**. This means that all funds are sent to the hash of a script. A **Script** is a series of instructions which are executed when a transaction is validated and added to a block. A transaction is only valid if the script returns a value of TRUE.

Every user has a set of default addresses, and hence scripts. For each of their root public keys, there is an associated default script - **RETURN(SIGNEDBY(RootPublicKey))**, the hash of which is an address of the user which can be used to receive funds.

Whenever a user receives funds to a specific address, a new coin is created containing that script, which must return a value of TRUE at the time of being spent in a transaction. This script will only return TRUE if the rightful owner of the coin has signed the transaction with one of the private keys for the given root public key - the one in the script - else it will return FALSE.

Example

When Alice sends funds to Bob's address, she is actually locking those funds into a new coin with a script which says RETURN(SIGNEDBY(Bob'sPublicKey)). This coin can only be spent when a transaction, containing this coin as an input, is signed with a single-use private key of Bob'sPublicKey.

Assuming Bob is the only one holding his private keys, Bob is the only person who can spend this coin.


Signature Generation

When choosing to sign a transaction with the root public key, the user must not only provide the **WOT signature** of the transaction, but also the **Parent-Child signatures** that connect the multiple levels of the tree, and the proof path from leaf node to root for each level. This provides the full path from the transaction signature to the root public key, which can be then validated by any other user in the network.

A full signature, required for the transaction **Witness** therefore includes a list of **MMR Proof paths** and **signatures** from the bottom of the tree of trees, to the root public key at the top.

For example, a full signature proof in a transaction Witness would consist of:

1. MMR Proof path from the root public key to a Level 1 Winternitz key leaf node with the **ParentChildSignature connecting Level 1 to 2** (as a result of using a level 1 key to sign a Level 2 root)
2. MMR Proof path from the root of the first Level 2 tree to a Level 2 Winternitz key leaf node with the **ParentChildSignature from Level 2 to 3** (as a result of using a level 2 key to sign a Level 3 root)
3. MMR Proof path from the root of a Level 3 tree to a Level 3 Winternitz key leaf node with the **Signature of the transaction** (as a result of using a level 3 key to sign the transaction)

 **NOTE**

Only the final signature is the signature generated from signing the transaction, the preceding signatures are the Parent-Child signatures that connect the multiple tree levels.

To learn more about MMR proofs, see the section on [MMR Database](#).

The serialised data for each MMR Signature Proof consists of:

Signature Proof Attribute	Description	Type
---------------------------	-------------	------

Signature Proof Attribute	Description	Type
Public Key	The Winternitz public key of the leaf node	64 byte hash (MiniData)
Proof	A list of Proof Chunks (nodes in the MMR tree) which provide the path from leaf node to the root of the tree.	MMR Proof
Winternitz Signature	A Winternitz Signature from signing either a child MMR key tree or some data e.g. a transaction	64 byte hash (MiniData)

In a transaction Witness, each Signature Proof also shows a root key which is the root hash of either the Level 1, 2 or 3 tree, where the Level 1 root hash is a multiple use root public key of the user.

Example of a full signature for a transaction:

```
"signatures": [{
  "signatures": [

{"publickey": "0xE574DE48114CE0C8B73B40BBA9069EE354C227EC0965123B458D2CB24EFD6A83",
"rootkey": "0xF9C0872B59932D11434CF3CCB23EDA1F7F189AC4438AD1D00AF94D7C28B6275B",
  "proof": {
    "blocktime": "0",
    "proof": [{
      "left": false,
      "data": {

"data": "0x51761AF1E1BD225EAED96916AC1317B9F47315B5155156B681B5DAA4B65EB699",
        "value": "0"}},
      {"left": false,
        "data": {

"data": "0x6F612A1F62206489CF6F00C3B6424C2D54E02B72C11763E21EC0E0C3D85161B8",
```

```

        "value":"0"}},
    {"left":false,
     "data":{
"data":"0x2E999787BFA586571880580CD7B99C748F18900D1CD207C9AC9538A30207285B",
     "value":"0"}},
    {"left":false,
     "data":{
"data":"0xD36DBD6C4E23A75B55AFB813EF067CA25A661C8AB82F5288BD0D0B2DCF0CA140",
     "value":"0"}},
    {"left":false,
     "data":{
"data":"0x4DDC92942DBCCBBC1C026ADC92715F8A021F16E6FE35810AAD30D6E698980E3D",
     "value":"0"}},
    {"left":false,
     "data":{
"data":"0x840F1A656596F02F1B787605E131AA1E3CF93DE618FFFC0EB03DDF301E861741",
     "value":"0"}}],
    "prooflength":6,
    "signature":"0x...",

{"publickey":"0xC14C2C8B35E55A2DF25EA0BA8A528BEEF8BD3FE688885B176BEBB8E8D95FAE67",
"rootkey":"0x707BE4E4F280CC96F5972F66FCDFCFC78356ACD548EF74513E24257FFED8DE6C",
 "proof":{
    "blocktime":"0",
    "proof":[{
        "left":false,
        "data":{
"data":"0x4F13DDDA0847150B2427C8E477908C790C5E001378816041CB550185303B9319",
        "value":"0"}},
        {"left":false,
         "data":{
"data":"0xE3B4F5B1ADB6F71974C13AADDE8C26FAC61F6C0DAD76BA9C316740DA1B5480B2",
         "value":"0"}},
        {"left":false,
         "data":{
"data":"0xE0B0051ED7B743EE466BF96D554B629B21A1588E09A18E6D2D02DBC77F26D473",
         "value":"0"}},
        {"left":false,
         "data":{
"data":"0xD42055AFA35C85B98C92F1734956857816EB67BB4C16AEB12CFF3EDD0BC2488A",
         "value":"0"}},
        {"left":false,

```

```

      "data":{
"data":"0xF1F986238938A8F82FD393E70F9226959FF115D3A4DC2AC7E13E40A58565B0E7",
  "value":"0"}},
    {"left":false,
      "data":{
"data":"0x1029171D8035E7461C5367987A6096A832F56C7558E6E859088718FC321F8DFA",
"value":"0"}
    }],
    "prooflength":6},
    "signature":"0x...",

{"publickey":"0xA54B6673D6A890444A90EFBE64FBD8576D59E144BB7166DA83109C9C32CF93B2",
"rootkey":"0x15DF8CEA59E66D31762DB7F8D3A972CFF55F0E8DA25CA9C4222AAF93BBD7A31E",
  "proof":{
    "blocktime":"0",
    "proof":[{
      "left":false,
      "data":{
"data":"0x6E2A8A0201D45E5B21003FA39FC32CF78755028F670687145DF365788AB83BEF",
"value":"0"}},
      {"left":false,
        "data":{
"data":"0xB9975A60B1187FFDF8C5F82DE910D47186DBB72A3CB478CB6BE168AD5FCD6AFD",
"value":"0"}},
        {"left":false,
          "data":{
"data":"0x07197775FA938D76A4252E2DD010E4C6145DCBE4EFD362E68DBDE79745563853",
"value":"0"}},
          {"left":false,
            "data":{
"data":"0x2C923B177B0417AF1E9A89858980D544CC065004F361B1BE395EC60DF674781F",
"value":"0"}},
            {"left":false,
              "data":{
"data":"0x235B875C4E2A48248DE06079613724D982A96069AE86ADD26F772AC192DDC5A2",
"value":"0"}},
              {"left":false,
                "data":{
"data":"0x9B90AD45CAEC23926163E65CC838207CDE1D48B990602E8F4AF060116BBEDA36",
"value":"0"}},
                "prooflength":6
            }],
    },

```

```
"signature":"0x...."  }  
]  
}],
```

Scripting

Minima has its own, Turing Complete, scripting language for creating Smart Contracts.

Minima, like Bitcoin, uses the UTxO model so writing smart contracts on Minima is quite different to writing them on an Account based model like Ethereum.

A Minima script (contract) returns TRUE or FALSE. The default is return FALSE, so all scripts must explicitly RETURN TRUE for the transaction to be valid.

A script can run for 512 instructions. An instruction is 1 operation or function.

The process to create a basic Smart Contract is as follows:

1. Write a script that will return TRUE when the funds should be spendable
2. Create the script, determining the **address** of the script. The address is the hash of the script.
3. Send funds to the script address and set the state variables, this will lock the funds in a coin.
4. Add the coin as an input to a transaction. A transaction in Minima is a set of input coins, a set of output coins and a state variable list from 0-255. Each 'coin' has an *amount*, *address (script hash)*, *tokenid* and *coinid*.
5. The transaction will only be valid when the script in the input coin returns TRUE, at which point the amount in the coin can be spent - in full.

A 'contract' is the script that locks the funds in a coin and is interchangeable with the word script.

A transaction can be signed by 1 or more public keys - and Signatures can even be added as state variables if you want oracle style contracts. Minima script is case sensitive.

The addition of the state variables in the MMR Proof DB, allow for complex scripts with knowledge of their past to be created. A simple state mechanic for transactional history rather than a global state for ALL transactions.

Each user tracks the coins to an address they possess and all coins that have a public key or address they possess in the STATE or PREVSTATE.

Minima transactions are scriptable Logic Gates, with analogue inputs and outputs, a simple yet powerful control language, and a previous history state mechanic.

Contracts are inherently compatible with Layer 2.

The scripting language supports SHA2-256 to allow cross-chain hash lock contracts with legacy chains.

Types of Contracts possible:

- Basic Signed
- Time Lock
- Multi-sig
- Complex multi-sig
- M of N multi-sig
- Hashed Time Lock (including cross-chain)
- Exchange
- FlashCash
- MAST

Example multi-sig contract:

```
scripts action:newscript trackall:true script:"RETURN  
SIGNEDBY(0x1539C2B974C1589C6AB3C734AA41D8E7D999759EFE057B047B200E836BA5268A) AND  
SIGNEDBY(0xAD25E1E40605A68AFE357ECF83E51FE27EC10013851AE95889A00C695D5B9402)"
```

Token Scripts

Each token has a separate script that must also return TRUE when attempting to spend a UTxO. For instance this could be 'make sure 1% is sent to this address, for a charity coin,

```
RETURN VERIFYOUT(@INPUT CHARITY_ADDRESS @AMOUNT*0.01 @TOKENID)
```

or a counter mechanism that checks a counter has been incremented:

```
RETURN STATE(99) EQ INC(PREVSTATE(99))
```


Both the address script and the Token script must return TRUE.

A token by default has RETURN TRUE as it's script. This token structure is added to any transaction wishing to use that token so every user can know how many, what scripts, name etc of the Token is correct and valid.

Grammar

```
ADDRESS      ::= ADDRESS ( BLOCK )
BLOCK        ::= STATEMENT_1 STATEMENT_2 ... STATEMENT_n
STATEMENT    ::= LET VARIABLE = EXPRESSION |
                LET ( EXPRESSION_1 EXPRESSION_2 ... EXPRESSION_n ) = EXPRESSION |
                IF EXPRESSION THEN BLOCK [ELSEIF EXPRESSION THEN BLOCK]* [ELSE
BLOCK] ENDIF |
                WHILE EXPRESSION DO BLOCK ENDWHILE |
                EXEC EXPRESSION |
                MAST EXPRESSION |
                ASSERT EXPRESSION |
                RETURN EXPRESSION
EXPRESSION   ::= RELATION
RELATION      ::= LOGIC AND LOGIC | LOGIC OR LOGIC |
                LOGIC XOR LOGIC | LOGIC NAND LOGIC |
                LOGIC NOR LOGIC | LOGIC NXOR LOGIC | LOGIC
LOGIC         ::= OPERATION EQ OPERATION | OPERATION NEQ OPERATION |
                OPERATION GT OPERATION | OPERATION GTE OPERATION |
                OPERATION LT OPERATION | OPERATION LTE OPERATION | OPERATION
OPERATION     ::= ADDSUB & ADDSUB | ADDSUB | ADDSUB | ADDSUB ^ ADDSUB | ADDSUB
ADDSUB        ::= MULDIV + MULDIV | MULDIV - MULDIV | MULDIV % MULDIV |
                MULDIV << MULDIV | MULDIV >> MULDIV | MULDIV
MULDIV        ::= PRIME * PRIME | PRIME / PRIME | PRIME
PRIME         ::= NOT PRIME | NEG PRIME | NOT BASEUNIT | NEG BASEUNIT | BASEUNIT
BASEUNIT      ::= VARIABLE | VALUE | -NUMBER | GLOBAL | FUNCTION | ( EXPRESSION )
VARIABLE      ::= [a-z]+
VALUE         ::= NUMBER | HEX | STRING | BOOLEAN
NUMBER        ::= ^[0-9]+(\\\\. [0-9]+)?
HEX           ::= 0x[0-9a-fA-F]+
STRING        ::= [UTF8_String]
BOOLEAN       ::= TRUE | FALSE
FALSE         ::= 0
TRUE          ::= NOT FALSE
```

```

GLOBAL      ::= @BLOCK | @INBLOCK | @BLOCKDIFF | @INPUT |
               @AMOUNT | @ADDRESS | @TOKENID | @COINID |
               @SCRIPT | @TOTIN | @TOTOUT
FUNCTION    ::= FUNC ( EXPRESSION_1 EXPRESSION_2 .. EXPRESSION_n )
FUNC        ::= CONCAT | LEN | REV | SUBSET | GET | OVERWRITE |
               CLEAN | UTF8 | REPLACE | SUBSTR |
               BOOL | HEX | NUMBER | STRING | ADDRESS |
               ABS | CEIL | FLOOR | MIN | MAX | INC | DEC | SIGDIG | POW |
               BITSET | BITGET | BITCOUNT | PROOF | KECCAK | SHA2 | SHA3 |
               SIGNEDBY | MULTISIG | CHECKSIG |

```

Globals

```

@BLOCK      : Block number this transaction is in
@INBLOCK    : Block number when this output was created
@BLOCKDIFF  : Difference between @BLOCK and INBLOCK
@INPUT      : Input index of a coin used in the transaction. First input coin has
an index of 0.
@COINID     : CoinID of this input
@AMOUNT     : Amount of this input
@ADDRESS    : Address of this input
@TOKENID    : TokenID of this input
@SCRIPT     : Script for this input
@TOTIN      : Total number of inputs for this transaction
@TOTOUT     : Total number of outputs for this transaction

```

Functions

```

CONCAT ( HEX_1 HEX_2 ... HEX_n )
Concatenate the HEX values.

```

```

LEN ( HEX|SCRIPT )
Length of the data

```

```

REV ( HEX )
Reverse the data

```

SUBSET (HEX NUMBER NUMBER)

Return the HEX subset of the data - start - length

OVERWRITE (HEX NUMBER HEX NUMBER NUMBER)

Copy bytes from the first HEX and pos to the second HEX and pos, length the last NUMBER

GET (NUMBER NUMBER .. NUMBER)

Return the array value set with LET (EXPRESSION EXPRESSION .. EXPRESSION)

ADDRESS (STRING)

Return the address of the script

REPLACE (STRING STRING STRING)

Replace in 1st string all occurrence of 2nd string with 3rd

SUBSTR (STRING NUMBER NUMBER)

Get the substring

CLEAN (STRING)

Return a CLEAN version of the script

UTF8 (HEX)

Convert the HEX value of a script value to a string

BOOL (VALUE)

Convert to TRUE or FALSE value

HEX (SCRIPT)

Convert SCRIPT to HEX

NUMBER (HEX)

Convert HEX to NUMBER

STRING (HEX)

Convert a HEX value to SCRIPT

ABS (NUMBER)

Return the absolute value of a number

CEIL (NUMBER)

Return the number rounded up

FLOOR (NUMBER)

Return the number rounded down

MIN (NUMBER NUMBER)

Return the minimum value of the 2 numbers

MAX (NUMBER NUMBER)

Return the maximum value of the 2 numbers

INC (NUMBER)

Increment a number

DEC (NUMBER)

Decrement a number

POW (NUMBER NUMBER)

Returns the power of N of a number. N must be a whole number.

SIGDIG (NUMBER NUMBER)

Set the significant digits of the number

BITSET (HEX NUMBER BOOLEAN)

Set the value of the BIT at that Position to 0 or 1

BITGET (HEX NUMBER)

Get the BOOLEAN value of the bit at the position.

BITCOUNT (HEX)

Count the number of bits set in a HEX value

PROOF (HEX HEX HEX)

Check the data, mmr proof, and root match. Same as mmrproof on Minima.

KECCAK (HEX|STRING)

Returns the KECCAK value of the HEX value.

SHA2 (HEX|STRING)

Returns the SHA2 value of the HEX value.

SHA3 (HEX|STRING)

Returns the SHA3 value of the HEX value.

SIGNEDBY (HEX)

Returns true if the transaction is signed by this public key

MULTISIG (NUMBER HEX1 HEX2 .. HEXn)

Returns true if the transaction is signed by N of the public keys

CHECKSIG (HEX HEX HEX)

Check public key, data and signature

GETOUTADDR (NUMBER)

Return the HEX address of the specified output

GETOUTAMT (NUMBER)

Return the amount of the specified output

GETOUTTOK (NUMBER)

Return the token id of the specified output

VERIFYOUT (NUMBER HEX NUMBER HEX)

Verify the specified output has the specified address, amount and tokenId

GETINADDR (NUMBER)

Return the HEX address of the specified input

GETINAMT (NUMBER)

Return the amount of the specified input

GETINTOK (NUMBER)

Return the token id of the specified input

VERIFYIN (NUMBER HEX NUMBER HEX)

Verify the specified input has the specified address, amount and tokenId

STATE (NUMBER)

Return the state value for the given number

PREVSTATE (NUMBER)

Return the state value stored in the MMR data in the initial transaction this input was created. Allows for a state to be maintained from 1 spend to the next

SAMESTATE (NUMBER NUMBER)

Return TRUE if the previous state and current state are the same for the start and end positions

Examples

```
LET thing = 23
LET ( 12 2 ) = 45.345
LET ( 0 0 1 ) = 0xFF
LET ( 3 ( thing + 1 ) ) = [ RETURN TRUE ]

--

RETURN SIGNEDBY ( 0x12345.. )

--

IF SIGNEDBY ( 0x123456.. ) AND SIGNEDBY ( 0x987654.. ) THEN
    RETURN TRUE
ELSE IF @BLKNUM GT 198765 AND SIGNEDBY ( 0x12345.. ) THEN
    RETURN TRUE
ENDIF

--

LET x = STATE ( 23 )
LET shax = SHA3 ( x )
IF shax EQ 0x6785456.. AND SIGNEDBY ( 0x12345.. ) THEN
    RETURN TRUE
ENDIF

--

EXEC [ RETURN TRUE ]

--

MAST 0xA6657D2133E29B0A343871CAE44224BBA6BB87A972A5247A38A45D3D2065F7E4

--

ASSERT STATE ( 0 ) EQ INC ( PREVSTATE ( 0 ) )

"
}
```


Quantum Security

If Minima is truly expected to stand the test of time, it must be Quantum secure from the beginning. Once scaled to hundreds of millions of nodes, each constructing and validating, the consensus critical components of the protocol must be finished, requiring no future changes.

Minima's approach to Quantum security is two-fold:

Hashing

Minima uses the **SHA3-256 hash function**, considered to be post-quantum sufficient by the National Institute of Standards and Technology (NIST).

SHA3-256 is used for TxPoW mining, block and transaction hashes, proof chains, and signing or verifying data. All of the cryptographic security of Minima is provided by hash functions.

Signatures

Minima uses Winternitz One Time Signature (WOTS) with a Winternitz parameter of 8. WOTS is a hash based digital signature scheme which is considered [Quantum resistant](#).

The cost of being Quantum secure is that signatures are at least 10-20x as big as Elliptic Curve Digital Signature Algorithm (ECDSA) used in Bitcoin. A one time use WOTS is 400-800 bytes. Minima signatures are certainly large when compared to normal Bitcoin transactions, but they are not kept forever since almost all data is eventually pruned, so although a bandwidth issue, they are only a temporary storage overhead.

Configuration

Base Types

Minima uses the following custom Base types:

Type	Description	Size (bytes)
MiniByte	Integer, Byte or Boolean	1
MiniData	A Hex or Base 32 hash	4 bytes (length) + Data length in bytes (Max 256MB)
MiniNumber	A decimal number with maximum 20 digits in front of the decimal and 44 digits after	1 byte (scale) + 1 byte (data length) + Data length in bytes
MiniString	UTF-8 String	MiniData representation of a UTF-8 string

General Parameters

Parameter	Type	Default Value	Description
IS_MOBILE	boolean	FALSE	Is this node running on mobile? Mainly for metrics
IS_ACCEPTING_IN_LINKS	boolean	TRUE	Can the node accept incoming connections?

Parameter	Type	Default Value	Description
PRIVATE_NETWORK	boolean	FALSE	Is this a private network? If TRUE, don't connect to any users.
AUTOMINE	boolean	FALSE	Are we automining a TxPoW every block?
GENESIS	boolean	FALSE	Are we creating the genesis block?
CLEAN	boolean	FALSE	Are we wiping previous data?
DATA_FOLDER	string	userhome.minima	Where the database files are stored. This is set at startup.
MINIMA_HOST	string	127.0.0.1	The Host IP
IS_HOST_SET	boolean	FALSE	Is the HOST set from command line?
MINIMA_PORT	int	9001	The main Minima port
RPC_PORT	int	9005	The Minima RPC port
TEST_PARAMS	boolean	FALSE	Test Params or Main Params
P2P_ENABLED	boolean	TRUE	Is the P2P System Enabled?
P2P_ROOTNODE	string		Host and IP of the first P2P node

Parameter	Type	Default Value	Description
CONNECT_LIST	string		Manual list of Minima nodes to connect to
NUMBER_DAYS_SQLTXPOWDB	long	3	How many days to keep the TxPoW in the SQL DB
NUMBER_HOURS_RAMTXPOWDB	long	1	How many hours to keep the TxPOW in the RAM mempool
NUMBER_DAYS_ARCHIVE	long	90	How many days do you archive the TxBlocks to resync users
USER_PULSE_FREQ	long	100*60*10	Number of seconds before sending a pulse message - every 10 minutes

Global Parameters

Parameter	Type	Default Value	Description
MINIMA_VERSION	string		The client version of Minima
MINIMA_BLOCK_SPEED	MiniNumber	0.02	Speed in blocks per second 0.02 = 50 second block time
MINIMA_BLOCKS_SPEED_CALC	MiniNumber	256	The number of blocks back to consider when checking speed and difficulty

Parameter	Type	Default Value	Description
MINIMA_CONFIRM_DEPTH	MiniNumber	3	The number of blocks deep before a block is considered confirmed
MINIMA_CASCADE_FREQUENCY	MiniNumber	100	How often (in blocks) the chain is Cascaded
MINIMA_CASCADE_START_DEPTH	MiniNumber	1024	Depth of heaviest chain before we cascade
MINIMA_CASCADE_LEVEL_NODES	int	128	Number of blocks at each cascade level
MINIMA_CASCADE_LEVELS	int	32	The number of Cascade levels
MINIMA_MMR_PROOF_HISTORY	MiniNumber	256	Max Proof History - how far back to use a proof of coin. If there is a re-organization of more than this the proof will be invalid.
MEDIAN_BLOCK_CALC	int	32	The MEDIAN time block is taken from this many blocks back. When calculating the Difficulty of a block (both from the tip and the previous block). This smooths out the time fluctuations for different blocks and removes incorrect times.

About

What is Maxima?

Every Minima node includes Maxima.

Maxima is to information what Minima is to value.

Where Minima provides freedom of *value* exchange; Maxima provides freedom of *information* exchange.

Maxima is an information transport layer protocol that allows anyone to exchange information, with their chosen contacts, over Minima's peer-to-peer network.

This unlocks the power of composability to create true Web 3 applications combining peer-to-peer exchange of value and information.

Totally decentralized - without censorship - without personal data collection.

Each Minima node runs the Minima blockchain in full to enable all nodes to transact with each other without third-party intermediaries.

Transacting over Minima involves sending coins, custom tokens or NFTs between nodes.

Transactions can be simple payments or programmed to cater for more complex use cases using Minima's smart-contract language, KISS.

When sending transactions over the network, users must perform a small amount of work from their device, called **Transaction Proof of Work (TxPoW)**, which secures the network from attack.

Each Minima node also runs the Maxima protocol to enable nodes to send encrypted data like text, pictures or documents to each other, independently of the Minima blockchain.

Users can communicate with a peer on the network e.g. a friend, family member or another device, by adding them as a **Maxima Contact**. Messages can then be sent to Maxima Contacts via

decentralized apps (known as MiniDapps) such as MaxSolo or Chatter which interact with the Maxima protocol.

Although users can send messages over Maxima free of charge, to prevent spam, they must contribute to the security of the Minima network by performing a small Tx-PoW when sending a message. On average, the TxPoW required to send a message amounts to 1 second of work.

Maxima Contacts

Your Maxima Contacts are your friends, family or other connections that you wish to communicate with, peer-to-peer, without a centralised third-party being able to manage, collect or intercept your messages.

Maintaining connectivity with each contact involves providing a small amount of Tx-PoW, in the same way that it is required for transacting over Minima, to contribute to the security of the blockchain.

As a result, currently, it is reasonable for a node to have around 20 contacts. Any more and it may start to impact the node's performance.

Contact Addresses

Similar to a wallet address used to receive funds on Minima, all users have a Contact Address which can be used to receive messages, over Maxima.

Knowing someone's Maxima Contact Address will enable you to send messages to their node. This communication is off-chain and uses peers on the network to ensure the encrypted message reaches the desired node.

Contact addresses change every 20 minutes, therefore contacts must be added shortly after an address is shared.

Once a contact is added, a user's [Maxima Location Service host \(mls\)](#) manages the connection between a user and their contacts to ensure they remain connected to their latest contact address.

Example Contact Address

```
MxG18HGG6FJ038614Y8CW46US6G20810K0070CD00Z83282G60G1JP0Y8WHVT0JJPGJ01YAQPCEU3PWF51V5J  
YAH34AJ1PZ4GWHCW7SEDQ0HQ9R4TU2G7NS8N816V13ERQ532PYWK9Z732RBZ7KFCQCENAMAJP9V7EH3R06493  
TB5VUJRV6QYVK1060800712NCHC@187.220.305.194:9001
```

Your Contact Address can be found and shared from the MaxContacts MiniDapp or by running the `maxima` command from the Minima Terminal and finding the `contact` address.

Privacy

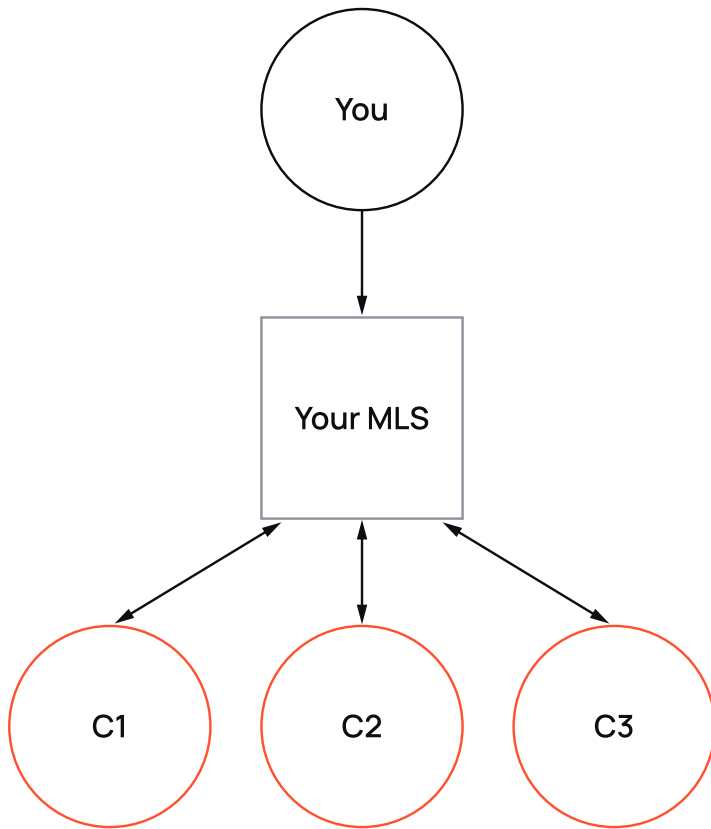
Both parties have control over their Contact list; if User A no longer wants to be connected to User B, User A can remove User B from their Contact list, which also removes them from User B's Contact list.

Furthermore, Contact Addresses change periodically, preventing User B from being able to reach User A at the same Contact address again, and allowing User A to become unreachable on Maxima, to avoid unwanted messages.

Maxima Location Service

The connection between you and your Contacts is maintained by your **Maxima Location Service (MLS)**, a randomly selected node on the Minima network.

Your MLS ensures that, although your Contact Address periodically changes, you will still remain connected to your Contacts, provided that you connect to the network at least once in a 24-hour period.



To learn more about MLS, see the [Maxima Location Service](#) page.

Maxima Messaging

Maxima Hosts

Each user has a set of Maxima hosts which facilitate the relaying of messages to the user. Maxima hosts are randomly selected server nodes a user has previously connected to.

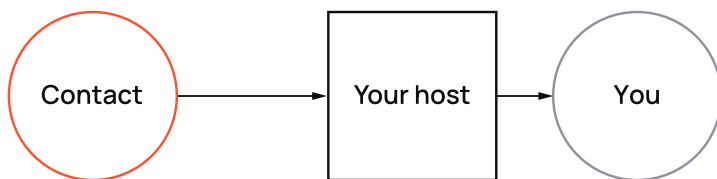
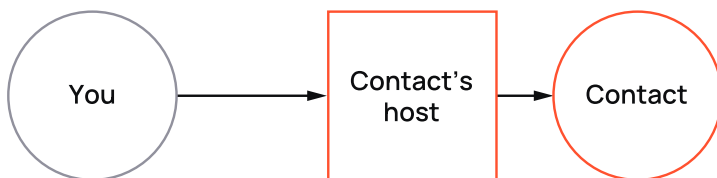
At any given time, only one Maxima host is responsible for forwarding a user's encrypted message.

You can see your Maxima hosts, including the one you are currently connected to, with the `maxima action:hosts` command from the Minima Terminal.

The host you are connected to determines the ip:port shown in your contact address.

Example Contact Address

```
MxG18HGG6FJ038614Y8CW46US6G20810K0070CD00Z83282G60G1JP0Y8WHVT0JJPGJ01YAQPCEU3PWF51V5J  
YAH34AJ1PZ4GWHCW7SEDQ0HQ9R4TU2G7NS8N816V13ERQ532PYWK9Z732RBZ7KFCQCENAMAJP9V7EH3R06493  
TB5VUJRV6QYVK1060800712NCHC@187.220.305.194:9001
```



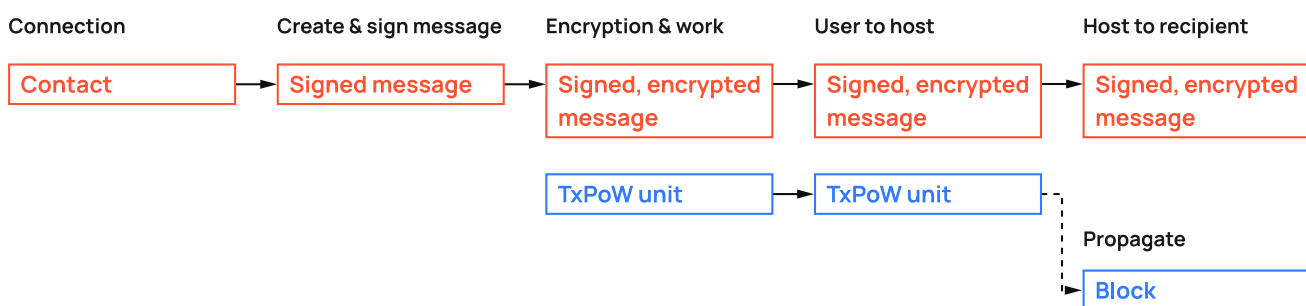
Sending Messages

Sending a message over Maxima has the following steps:

1. **Connection** - The user exchanges contact details with a friend and (optionally) adds them as a Maxima contact.
2. **Create & Sign Message** - The sender creates a message either via a Minima MiniDapp or using Terminal commands. On sending the message, the message is signed with the sender's Maxima private key, generating a signature for the message.
3. **Encryption & Work** - the Maxima data package containing the public key of the sender, the message and the signature is encrypted. Sending a message has no monetary cost, but it must be paid for in 'work'. A TxPoW unit is created, mined and sent to the recipient's Maxima host with the signed, encrypted message.
4. **Send: Sender to Host** - the signed, encrypted message and mined TxPoW unit is sent to the recipient's Maxima host who checks whether the TxPoW is valid and the minimum amount of 'work' has been completed.
5. **Send: Host to Recipient** - If the TxPoW is valid and the minimum amount of 'work' has been completed, the message will be forwarded to the recipient, else it is discarded (off-chain process).
6. **Block propagation** - If the TxPoW unit is a block, it will be propagated to ALL nodes for processing, otherwise the TxPoW unit will be discarded (on-chain).

i NOTE

All Maxima messages are encrypted end-to-end using asymmetric RSA encryption. Messages can only be decrypted and read by the recipient, other nodes involved in the relaying of the message cannot read the data.



Orange: A Maxima process

Blue: A Minima process

Maxima Location Service

Your Maxima Location Service (MLS) is a randomly selected server-based node that ensures your Contacts always have your most recent Contact Address. Your contacts periodically request your current Maxima contact address from your MLS, allowing them to always reach you.

When your Contact Address changes, your MLS host is updated and your online Contacts are informed of your new MLS and current Contact Address, ensuring you always remain connected.

For enhanced privacy, your node connects to a new MLS host every 12 hours. To ensure a smooth changeover, for 12 hours after moving to a new MLS host, your node continues to share your latest Contact Address with your previous MLS as well, in case any of your Contacts have been offline for longer than 12 hours.

This ensures that if you remove a contact, they will not be able to use the MLS they hold for you for much longer, to request your contact address.

The outcome is that, provided you and your Contact are online at least once in a 24-hour period, you will be able to communicate with each other over Maxima.

Static MLS

You may wish to become your own MLS or use another unchanging MLS to ensure a more stable with your contacts.

You can set up your own MLS by running another Minima node on a server that is open on the main Minima port (default port 9001).

By running your own MLS, your online Contacts will always remain connected to you via your server node, instead of using a randomly selected server node that could go offline and changes every 12 hours.

To set your own static MLS host, from your server-based Minima node, find its `P2P identity` using the `maxima` command via the Minima Terminal.

Server node - example output:

```

maxima
{
  "command": "maxima",
  "status": true,
  "pending": false,
  "response": {
    "logs": true,
    "name": "mlsnode",

    "publickey": "0x30819F300D06092A864886F70D010101050003818D00308189028181009570D2AB5CB4
    "staticmls": false,
    "mls": "MxG18HGG6FJ038614Y8CW46US6G20810K0070CD00Z83282G60G1JG4Q2698U35U5QFQZUYM1Q
    "localidentity": "MxG18HGG6FJ038614Y8CW46US6G20810K0070CD00Z83282G60G16YEDA34PYMJS
    "p2pidentity": "MxG18HGG6FJ038614Y8CW46US6G20810K0070CD00Z83282G60G16YEDA34PYMJS7M
    "contact": "MxG18HGG6FJ038614Y8CW46US6G20810K0070CD00Z83282G60G19DCGVJUKP4Y93Z8NBZ
    "poll": 0
  }
}

```

On the node that you wish to use to communicate with your Contacts, set your static MLS using:

```
maxextra action:staticmls host:INSERTP2PIDENTITY
```

Phone/Desktop/Server node requiring a static mls - example output:

```

maxextra action:staticmls
host:MxG18HGG6FJ038614Y8CW46US6G20810K0070CD00Z83282G60G16YEDA34PYMJS7MC82PYEHB2Q5G2S
{
  "command": "maxextra",
  "params": {
    "action": "staticmls",

    "host": "MxG18HGG6FJ038614Y8CW46US6G20810K0070CD00Z83282G60G16YEDA34PYMJS7MC82PYEHB2Q5
  },
  "status": true,
  "pending": false,
  "response": {
    "staticmls": true,

```

```
"mls": "MxG18HGG6FJ038614Y8CW46US6G20810K0070CD00Z83282G60G16YEDA34PYMJS7MC82PYEHB2Q5G"
}
```

NOTE

This will not prevent your Contact Address from changing, it only ensures that your Contacts can always rely on your static MLS to provide your latest Contact Address.

To subsequently stop using a static mls and return to using a random host, use

```
maxextra action:staticmls host:clear
```

Permanent Addresses

For certain use cases, where it's necessary to be **publicly contactable by someone who is not a Contact**, it's possible to create a **permanent address** for your node by adding your Maxima Public Key to your static MLS node.

Once configured, sharing a permanent address publicly enables any user to always be able to find the current Contact Address of your node, so they will always be able to contact you.

To enable a permanent address for your node, you must set your static MLS node to accept requests for *anyone* (not just contacts) to get your current Contact Address by setting the Maxima Public Key of your node as "permanent" on your static MLS node.

Setting up a Permanent Address:

On the node you wish to use for communication which you have already configured to use a static MLS, find your Maxima Public Key using the `maxima` command from your Terminal MiniDapp.

Example output:

```
maxima
{
```

```

"command":"maxima",
"status":true,
"pending":false,
"response":{
  "logs":false,
  "name":"yourname",

"publickey":"0x30819F300D06092A864886F70D010101050003818D0030818902818100958DD8EA74BD
  "staticmls":true,
  "mls":"MxG18HGG6FJ038614Y8CW46US6G20810K0070CD00Z83282G60G16YEDA34PYMJS7MC82PYEHB
  "localidentity":"MxG18HGG6FJ038614Y8CW46US6G20810K0070CD00Z83282G60G13YGKSTHJVZ77
  "p2pidentity":"MxG18HGG6FJ038614Y8CW46US6G20810K0070CD00Z83282G60G13YGKSTHJVZ77NA
  "contact":"MxG18HGG6FJ038614Y8CW46US6G20810K0070CD00Z83282G60G1FSDCR470HWPNA11N8D
  "poll":0
}
}

```

Copy your Public Key, then **on the static MLS node** enter the command:

```
maxextra action:addpermanent publickey:INSERTPUBKEY
```

Example:

```

maxextra action:addpermanent
publickey:0x30819F300D06092A864886F70D010101050003818D0030818902818100958DD8EA74BD3CB
{
  "command":"maxextra",
  "params":{
    "action":"addpermanent",

"publickey":"0x30819F300D06092A864886F70D010101050003818D0030818902818100958DD8EA74BD
  },
  "status":true,
  "pending":false,
  "response":"Added Permanent Maxima ID :
0x30819F300D06092A864886F70D010101050003818D0030818902818100958DD8EA74BD3CBEE4D1D8F03
}

```

Now the node you are using for communication will be have a Permanent Address in the format

MAX#yourPubKey#staticMLSAddress

Example Permanent Address:

```
MAX#0x30819F300D06092A864886F70D010101050003818D0030818902818100958DD8EA74BD3CBEE4D1D
```

This Permanent Address can now be shared with others, for example on a public site.

Anyone wishing to contact you over Maxima can get your current Contact Address using:

```
maxextra action:getaddress maxaddress:MAX#yourPubKey#staticMLSAddress
```

Example

```
maxextra action:getaddress  
maxaddress:MAX#0x30819F300D06092A864886F70D010101050003818D0030818902818100958DD8EA74
```

The **address** from the returned output is the current contact address of your node that they can use send you a message using:

```
maxima action:send application:none message:0xff contact:INSERTADDRESS
```

PREVENTING UNAUTHORIZED CONTACTS

Having a Permanent Address means anyone else on the network can add you as a Contact without your consent. See [Disabling Contacts](#) to learn how to disable contacts.

LIMITING CONTACTS

Maintaining connectivity with each contact involves providing Tx-PoW to help secure the blockchain, therefore the number of Contacts added to a node cannot be unlimited. **The more contacts your node has, the higher the computational overhead to maintain them.**

Disabling contacts

To prevent anyone from adding you as a Contact when using a Permanent Address, **from the node you are using for communication**, enter the following in your Terminal MiniDapp:

```
maxextra action:allowallcontacts enable:false
```

This will allow anyone to find your current Contact address using your Permanent address and will allow them to send messages to your node, but will reject any requests to add you as a Contact.

Whitelisting contacts

Specific nodes can be whitelisted by allowing the Maxima Public Key of the node you wish to be contacts with.

Once allowed, this node will be authorized to add you as a contact and, vice versa, you may add this node as a contact provided they have not disabled contacts.

```
maxextra action:addallowed publickey:INSERTPUBLICKEY
```

Additional Help

For additional help, please use the help command from the Terminal:

```
help command:maxextra
```

Maxima FAQ

How does Maxima strengthen Minima's security?

Each time a message is sent over Maxima, the sender's node must perform a small amount of work (Tx-PoW) which contributes to the overall hash rate of the Minima network, increasing the value of the network as a whole.

The **more messages** sent over Maxima, the more Tx-PoW securing the network;

The **more secure** the network, the more value can be stored on the network;

The **more value** can be stored on the network, **the more valuable the network**.

Is Maxima Free?

There is no monetary cost to send messages over Maxima.

Messages are paid for in work – Transaction Proof-of-Work (Tx-PoW). In other words, your node performs a small amount of work in the background for every message you send, which contributes hash power to secure the Minima network.

You don't need any Minima coins to use Maxima.

What are Maxima Contacts?

Your Maxima Contacts can be your friends, family and connections that you wish to communicate with over Maxima.

By adding the people you know as a Contact in your Minima node, you create a connection with their node across Minima's peer-to-peer network. This means you will be able to chat and transact with them using the MaxSolo MiniDapp and, in future, any other MiniDapps which use Maxima.

What is my Maxima Profile and Contact Address?

Your Maxima Profile is the name that your Maxima Contacts will see once you are connected to each other. Your Contact Address identifies your Minima address and location in the peer-to-peer

network. By sharing your Contact Address with people you know and getting them to add you as a Contact, you are allowing them to communicate with you using the MaxSolo MiniDapp.

What is MaxSolo?

MaxSolo is a messaging MiniDapp that uses Minima and Maxima to enable you to exchange messages, Minima coins, custom tokens and NFTs to your Maxima contacts.

The latest version of MaxSolo can be downloaded from our [MiniDapps site](#) and installed on your Minima node.

Are my Maxima messages private?

Messages sent over Maxima are signed by the sender and encrypted end-to-end, so only the recipient of your message is able to decrypt and read it.

Can I disconnect from a contact?

For privacy, your Maxima contact address changes periodically. When you choose to delete a contact, you also remove yourself from their Contact list, and they will no longer be able to locate you in the peer-to-peer network.

How can I use Maxima for messaging?

Every Minima node includes Maxima. To use Maxima, you must first set your Maxima Profile name and create your Maxima Contacts, then you will be able to communicate with your contacts using any MiniDapp that uses Maxima, for example, MaxSolo.

For setup instructions, please follow the instructions in the [Using Maxima](#) section.

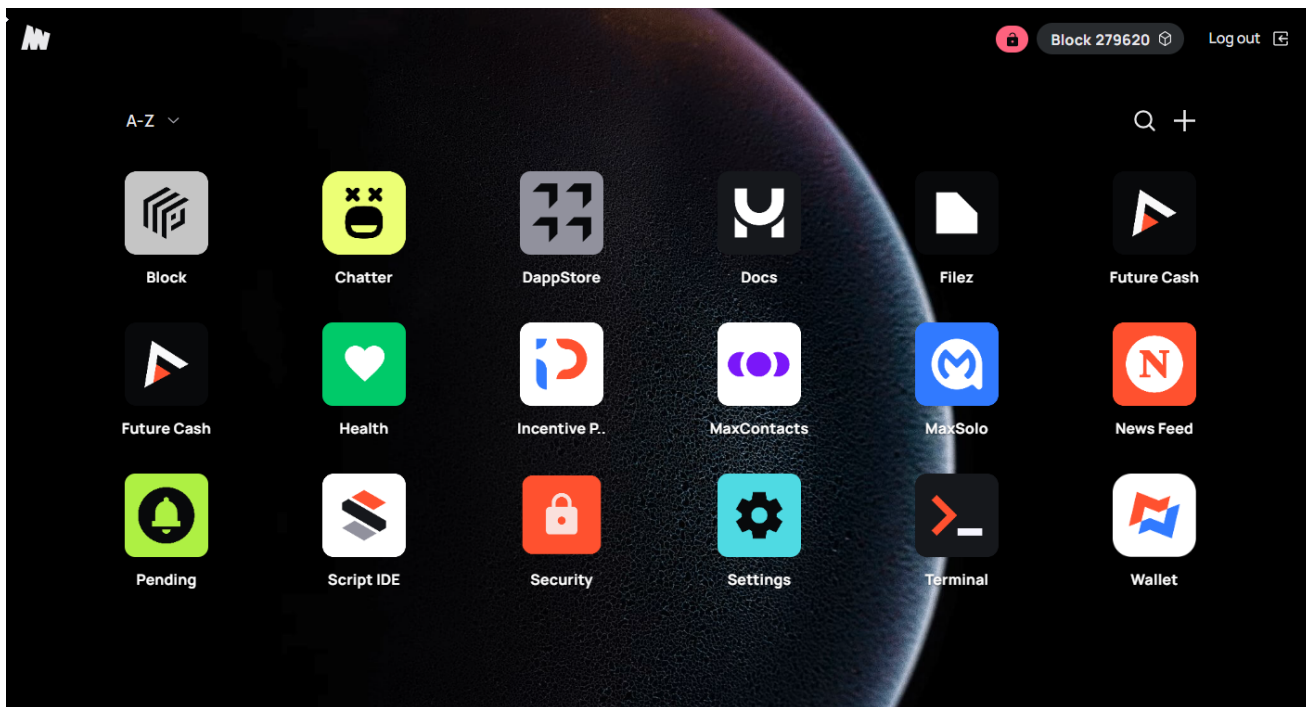
For more information on MiniDapps, please see the [MiniDapps](#) section.

About

Overview

We are excited to announce the highly anticipated release of our new MiniDapp System!

Before you get stuck in, here is an introduction to the MiniDapp System and explanation of what MiniDapps are.



What is the MiniDapp System?

The MiniDapp system is an open application ecosystem.

Similar to the Operating System on your phone that enables you to use Android apps, the MiniDapp System enables you to use MiniDapps on your Minima node.

What are MiniDapps?

MiniDapps are Minima's **Decentralized Applications** running over Minima's blockchain network. They are web applications that utilise the functionality enabled by the Minima blockchain and peer-to-peer network. This includes our Information Layer - Maxima and, in future, our Layer 2 - Omnia. They are your gateway to a true Web 3 economy, open and free.

A MiniDapp can be created for almost anything - some examples include decentralized finance, games, voting, marketplaces, messaging, social media - the world is your oyster.

How do MiniDapps differ from other decentralized applications (dApps)?

Other 'decentralized' applications are decentralized in name only, often relying on centralized services to run effectively. MiniDapps are not. MiniDapps are installed on and run directly from your node, and because every Minima user runs a complete node, there is no need to rely on centralized websites or companies to use MiniDapps. They are permissionless and borderless.

What MiniDapps are currently available?

You will find the latest MiniDapps, developed by the Minima Team, available to download from the [MiniDapps website](#).

In future, we expect the majority of MiniDapps to be built by the open-source community—that could be you!

How do I use MiniDapps?

For help using MiniDapps, please see [Using MiniDapps](#).

Who can build a MiniDapp?

Anyone can learn to build a MiniDapp.

The front-end for MiniDapps can be written using the widely known JavaScript, HTML and CSS. Minima's KISS scripting language is Turing-Complete, allowing for powerful smart contract driven applications. Building a MiniDapp is accessible to anyone willing to learn!

Visit the [Build](#) section to learn more about building MiniDapps on Minima.