Bitcoin Blink : A Peer-to-Peer Global Finance System

Joby Reuben
joby@bitcoinblink.org

Purva Chaudhari
purva@bitcoinblink.org

Ajay Joshua
ajay@bitcoinblink.org

Abstract. Bitcoin's Proof-of-Work consensus is replaced with a propagation competition on blocks sent across a certain set of validators under a time interval stamped with cryptographic proofs to claim fees and solve forks as per proof weight. To achieve adaptable scalability, each block's size is determined in consensus among elected nodes to reduce transaction waiting time. Gossip systems are replaced with a privacy-centered direct messaging system by constructing encrypted paths to deliver unconfirmed transactions and confirmed blocks. Apart from bringing speed, we resolved the need for a single transaction fee token by bringing forth a novel non-custodial per-token staking system that allows users to pay transaction fees in any token. The "bitcoin" as a currency will ensure network security with staking and yielding fees. Since Bitcoin script adapts a Turing-incomplete language, fees are imposed for renting UTXOs which makes transactions cheaper and optimizes the chain's ledger size. We propose solutions for regulation revolving around taxation within the self-custody ecosystem by offloading responsibilities to wallet providers.

1 Introduction

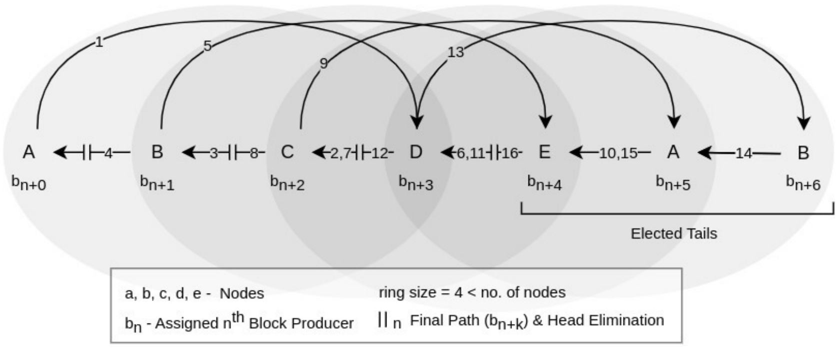
The Bitcoin Network [1] and other altcoin blockchains with newer consensus and programmable money are unable to compete with centralized payment providers in speed and volume due to their sheer inability to scale with centralization issues. Many consensus models rely on external validation concepts such as finding a nonce and proving stake instead of rules tied to propagation itself. Imposing heavy reliance on users to acquire native chain tokens diminishes the adoption of blockchain, thus keeping them far from the wonders of this technology. Decentralized networks can effectively adapt to users' needs by 1. Increasing block size 2. Decreasing block time 3. Increasing production requirement. Retail staking with non-custodial solutions encourages users to stake their bitcoin to become a world reserve currency for ensuring the security of every financial instrument.

Instead of storing UTXOs for an indefinite period, which compromises storage, renting UTXOs and replacing them with a fingerprint after they have expired without altering the block's Merkle-root provides cheaper fees. Bitcoin's unlocking script offers developers to create custom scripts with regulatory options involving various types of taxes within its UTXOs, performing identity verification off-chain, with signatures instructing nodes to validate regulated payments under self-custody of tokens. Decentralized altcoins can be bridged one way to facilitate payments inside a scalable decentralized infrastructure. A floating rate stablecoin [2] without pegging to fiat can be issued with bitcoin as collateral for staking and yielding fees in the future. Basic banking solutions can be developed in Bitcoin script, whereas common computable programs can be deployed to a Layer 2 State Machine [3], where nodes update the state by providing a Proof-of-Receipt paid in any token in the Bitcoin network.

2 Bitcoin Blink

2.1 Validation

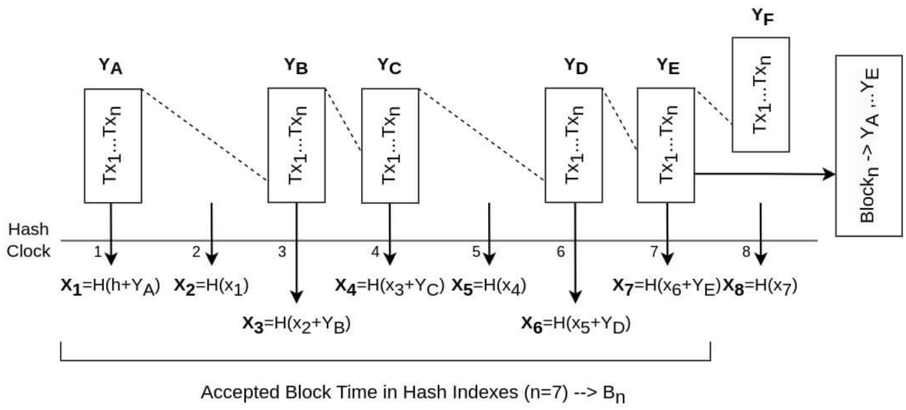
Block size denotes the amount of data that can be propagated across every node on the Bitcoin network. Hence, its success rate is directly dependent on the bandwidth allocated by each node for confirmed block transmission. Block size is not limited and is fixed in every new block, which assures that every validator of the block can send and receive the data size. Variable block size helps to scale the network by increasing transactions per block when nodes upgrade and announce their bandwidth. A ZK-SNARK-based proof takes a node's bandwidth as a public input signal, which is compared to the threshold range of the desired bandwidth of the network. To prevent tampering, the node will commit a salt, which is a large random number that will be added to the bandwidth to generate a hash which will be verified in the proof. Thus, the proof will attest its bandwidth output to the UTXO's script and get validated.



The network in consensus can restrict low bandwidth nodes from participating in the election to produce blocks by keeping the election requirement variable. Bandwidth requirements are increased for nodes to get elected for the next block production can assure the capacity to hold more transactions to scale further. Elections will be conducted based on nodes' bandwidth and each node's honesty weight. Node identities are masked by their public keys encouraging privacy. For each block, the elected node will produce, and a set of nodes will validate and attach to the longest chain [1]. This set of nodes is finite and unique for every block and is called a "ring validator". After each block's confirmation, the head of the ring is eliminated and a tail is elected which can be validated by nodes during propagation. The ring head after receiving the confirmed block will acquire bandwidth to gossip its block over to other full or SPV nodes to secure its rewards. The tail election's random seed is taken from concatenating Merkle-Chain-root and Hash Proof [4] of the recent head's confirmed block. Thus, the election is conducted for every block where a tail node is assigned.

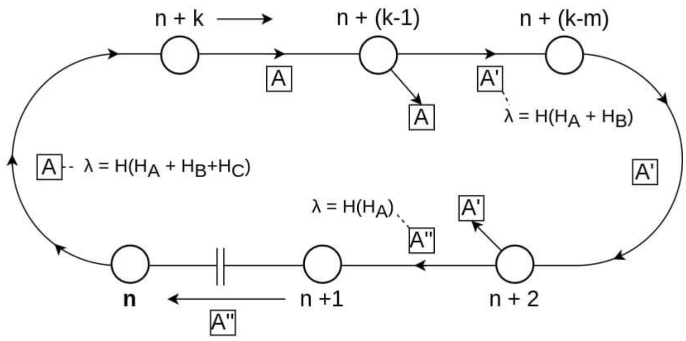
2.2 Propagation

A block is constructed as snips and collectively validated. Snips are divisible block chunks produced by the producer and directly messaged to the block's ring validators with routing instructions to propagate and get confirmed. Each snip references the previous snip's hash similar to the chain of blocks for proper identification of each block's snips. For a block, a competition to deliver all snips under x time interval in a finite number of hashes is required to win rewards and avoid slashing of fees. When a block fails to win, it will do an intra-block fork and not mint its last of snips which will contain the fees and rewards. Hash proofs are attached for every snip during propagation among its assigned ring validators to declare the state of each block's competition and resolve forks. Failed block fees and rewards are slashed by sending them to a burn address by the next blocks which also results in the addition of negative weights that indirectly slashes the node's bandwidth costing capital. Null-Blocks are self-minted by its ring validators with proofs.



To synchronize time, each node's single-threaded hash rate is proved cryptographically onchain and taken in multiples of a reference hash rate. An individual hash-rate proof is provided along with bandwidth proof for every x block height (where x is the ring size) before it expires. This trustlessly synchronizes all nodes as a single hardware producing continuous hashes concated with all snips. The ZK IHR proofs are algorithmically similar to the bandwidth proofs, where the hash rate provided by the node is compared to the threshold range of the desired hash rate and salt is used to prevent tampering attacks. This time-based competition can be termed "Proof of Speed".

2.3 Forks

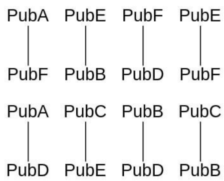


Confirmed blocks as snips are streamed in a backward ring manner (block$_n$ to block$_{n+k}$...block$_{n+1}$). For each block to prove block time by providing Hash proofs, a set of validators is initialized by the ring's size denoting the unique number of blocks and their producers. Intra-block forks arise when ring validators reject snips of a block which can be resolved by the very next block providing instant finality to users. Intra-block forks can be minimized by keeping the ring size variable and always lesser than the total number of nodes, a viable alternative to sharding. Block size and time are decided on consensus by the block's ring validators bringing a healthy propagation. Bandwidth upgrades are needed for the nodes which initiate the intra-block fork and are punished by the network by imposing negative weights.

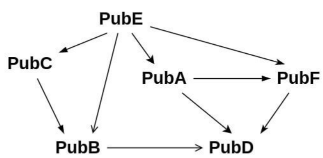
Each new block will include the previous block's Hash proof which resolves the intra-block forks by verifying the accepted snips and updating the chain using Merkle Chain Roots. During the offline activity of ring validators, inter-block forks arise which can be resolved by attesting Hash Proofs of offline activity and thereby adding negative weights to it. Hence, each block will have its ring validators who are assigned by onchain accounted bandwidth and honesty weights, where the heaviest Hash-proofed version of the block$_n$ is added to the longest chain.

2.4 Messaging

Delivery of unconfirmed transactions to nodes plays an important role in finality. Shared mempools defile the network with duplicated data, resulting in a poor choice of transactions by accepting higher fees to include in a block. A direct-messaging system should be deployed with messaging instructions specific to each party as opposed to gossip protocols. Paths are attached with unconfirmed transactions directly from the constructed network graph, which is available to all nodes with public keys as pseudonymous identities protecting privacy. Two peering parties mutually sign a 2–2 random message for every x block height (where x is the ring size) and are gossiped across the network to identify the connection as online. All the signed random messages prove that each pubkey signature can display a network topology map from a node's point of reference.



2-2 Signed Message



Origin (E) -> Destination (D)

Paths are encrypted end-to-end with routing [5] instructions so that the origin cannot be traced. Nodes route transactions to the destinations where producers can attach the transaction to their allocated block. Since the block producers and their stake information is available in the public ledger, wallet providers can construct transactions along with path information that routes to the nearest blocks for instant finality. Transactions are always atomic, providing a solution to queuing issues. Responsibilities are provided to all participants, where nodes only receive the transactions that they need to include, and wallet providers should construct shorter paths to provide the best user experience.

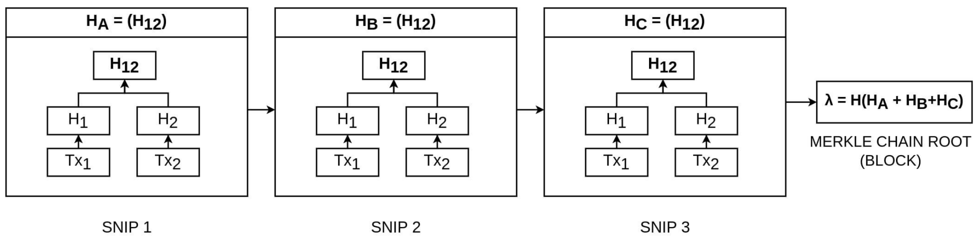
2.5 Staking

The chain's native token - bitcoin can be staked for nodes' public keys with specified token IDs, where the collateral can be used only once for a block. This results in a stake per token per block bringing the throughput (tps), per token basis. The proofs are based on Zk-Snark-based Semaphore protocol [6] which will allow users to prove their membership in a group and send signals without revealing the original identity. Each token per block's collateral requirement is given in bitcoin denominations. This collateral requirement provides security to restrict fraudulent tokens by requiring a bitcoin exchange rate at initial for the token's transaction to be included in the block. The bitcoin exchange rate inherent to the system independently offers market rates in a trustless manner without requiring a stable fiat-based token or off-chain oracles.

For specific nodes, bitcoins can be staked to collateralize or lock in their allocated block. Staked bitcoins can be withdrawn at any time, with no vesting period, except when the block is being created. This brings in a retail and non-custodial solution as opposed to security deposit-type PoS chains. Delegators won't lose their stakes as slashing is done directly on the fees during intra-block forks. For each transfer, wallet providers can offer bitcoin users rewards from their accepted tokens to avail non-custodial liquidity for staking that benefits both parties. In this way, for a specific token's transaction to be included in a block, the first of transactions should prove the collateral specific to the token.

2.6 Renting

Instead of a single Merkle-root for a block, each snip's Merkle-root is taken and linearly hashed to find the Merkle-Chain-root, a pseudo-random value. Snips contain hashed transactions which can be pruned if the UTXOs are spent, burnt, or expired without altering the Merkle-Chain-root. Each UTXO expiry block height is embedded in its script, and can be scanned by nodes, and pruned to optimize their data storage. Wallet providers can store each of their users' transaction history and can be audited onchain using Merkle-Chain-roots. Renting rates can be independently voted by producer nodes per byte per block in bitcoins. Users cannot directly pay for rent, but rather each new UTXO created is charged a transfer fee in the range of 0.05% - 0.005% decided based on the total volume of all transactions settled on the previous set of blocks (ring size).



Transfer fee charges more fees for higher value utxos and less for lesser value utxos bringing ease to transact for retailers. Each UTXO will set an expiry date for itself. Users can be incentivized by charging zero fees to combine UTXOs to a single balance holding UTXO with an increased expiry value saving validators' disk space.

2.7 Rewards

Rewards are given for each snip hash concated with transactions validated by Hash proofs. Since newly mined bitcoins for a period of time are definite and each block's time is capped in reference hash-rate, new bitcoins can be supplied exactly proportional to the current Bitcoin issuance rate with halving. The halving of Bitcoin issuance is done every $R \cdot (1.26 \cdot 10^8)$ hashes, where R is the reference single thread hash rate. Each hash proof represents work done by nodes on validating and propagating transactions. When an intra-block fork arises due to rejected snips, the rest of the block time and its allocated rewards are slashed. Meanwhile, when a block is fully minted before the block time finishes, the rest of its allocated rewards are distributed to the current halving period. This incentivizes nodes to attest and receive snips at the earliest to get an increased block reward in the upcoming block production. Tax outputs and rewards are attached as zero input transactions within the snip it contains. Fee outputs are created as a separate snip which denotes the end of a block.

During staking, producers announce their accepted tokens for which they will directly withdraw the commission. For other tokens, delegators can stake with a condition that their stake in bitcoins will be traded for the collected fees. During the commission withdrawal of non-accepted tokens, the producer will deposit collected fees to delegators and inflate the stake 1:1 ratio to withdraw the collateral. Users can pay transaction fees in any token, delegators incur the risk, and producers get paid in tokens of their choice to validate transactions.

2.8 Treasury

For the active development and sustainability of the project, a Decentralized Bitcoin Organization is set up to fund developers and its community. A minimal commission is imposed on producer fees and deposited to a treasury script. Memberships are non-fungible 1. Temporary (Core-Developers), 2. Permanent (Investors, Community), 3. Contracts (Employees, Operations, Grants, etc given in x amount per y term for z period). Except for Permanent members, votes are taken to decide membership and treasury decisions. For decisions involving protocol upgrades, the majority of nodes can decide. Temporary members can be kicked out for failure in active contribution whereas Permanent members cannot be kicked out due to their external contribution (funds) towards building the project.

Permanent and Temporary members cannot be added after the first mainnet but can appoint heirs for their registered membership. Contracts are paid out initially and the rest is provided to other memberships as dividends according to their weight. Participants are only rewarded for their active contribution, not indefinitely like holding a fungible token or speculative participation. Temporary members' weights get halved every 4 years to decentralize decisions among community participants and completely allocate treasury for contract memberships. A decentralized open-sourced organization structure is maintained with decisions involving votes bringing forth sustainable growth for the future of bitcoin blink.

2.9 Taxes

Regulation via centralized exchanges & custodians risks funds and doesn't encourage a self-custodial ecosystem. Wallet providers bear the responsibility to acquire the payee and payer's regional information to register taxable details onto their newly created UTXO's unlocking script in compliance with regulatory rules. For every transaction, a client witness signature is added. Based on the client provided - user signed spending conditions the Bitcoin network will execute unlocking scripts and deduct taxes. Bitcoin scripts can work efficiently and securely, as opposed to turing-complete smart contracts. Tax models such as capital gains, TDS, and sales tax which depict all the available tax slabs in current finance can be brought and issued by wallet providers. While spending the UTXO, taxes are transferred to regulators' wallet addresses added in the transaction's snip itself. A trustless & decentralized tax-deduction environment is provided to regulators as an option to adopt Bitcoin for global finance.

3 Future

The future of the Bitcoin network includes building finance-specific L1 applications such as bridges, lending & borrowing, insurance, token minting, and bank-mirrored wallets. Since these applications are developed inside an unlocking script, it requires preimage construction off-chain and settles onchain - inheriting the security of Bitcoin that centralized applications don't offer. A floating rate stablecoin will be developed and issued as an alternative to currently available stable assets fully backed by bitcoin for stable payments. Privacy can be improved by obscuring amounts similar to Monero [7], with tax validation assisting regulators and masking financial information of a specific country. Emphasis on Zk delivery and validation of snips can reduce influence attacks in the future.

An offline digital cash-payment system will be developed to provide an alternative to paper cash with keyless-signing methods. To provide a general-programmable environment, a State Machine [3] featured with multiple high-level languages using LLVM [8] IR code will be deployed as a layer that can update its global and contract state by providing a gas limit through a receipt-proof paid in Bitcoin Network. The smart contracts will not contain balances and Externally Owned Accounts, but rather purely executed for business-logic to build DApps without a financial scope. To store client-side assets and data files for building fully decentralized applications, a Content Delivery Network module will be developed in Bitcoin Network without duplicating data among nodes to provide faster content delivery to end-user. Moreover, research will be conducted to merge various Bitcoin and altcoin chains into a single decentralized global scalable infrastructure for a clean experience on the whole of finance and computing without the hassles that are prevalent in the current web3 ecosystem.

References

[1] S. Nakamoto, "Bitcoin : A peer-to-peer electronic cash system," *Whitepaper*, p. 9, 2008.

[2] "Dai stablecoin purple paper." `https://web.archive.org/web/20210127042114/https://makerdao.com/purple/`.

[3] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[4] A. Yakovenko, "Solana: A new architecture for a high performance blockchain v0. 8.13," *Whitepaper*, 2018.

[5] J. Poon and T. Dryja, "The bitcoin lightning network: Scalable off-chain instant payments," 2016.

[6] "semaphore-protocol/semaphore: A zero-knowledge protocol for anonymous signalling on ethereum." `https://github.com/semaphore-protocol/semaphore`.

[7] S. Noether, A. Mackenzie, and t. M. Research Lab, "Ring confidential transactions," *Ledger*, vol. 1, p. 1–18, Dec. 2016.

[8] "The llvm compiler infrastructure project." `https://llvm.org/`.

REFERENCE ALGORITHMS

March 7, 2023 (v0.1)

Disclaimer

These pseudocodes provide basic instructions for the implementation of bitcoin blink protocols. It is periodically reviewed and updated by core programmers and co-authors. Revisit for newer versions.

Algorithm 1: Network Graph

```
var networkGraph = {}
/* an empty object to represent the network graph */
var transactions = [ ]
function addPeerToGraph (peer):
    networkGraph[peer.publicKey] = peer
function removePeerFromGraph (peer):
    delete networkGraph[peer.publicKey]
function signAndSendRandomMessage (peer):
    var randomMessage = generateRandomMessage()
    var signedMessage = signMessage(randomMessage)
    sendMessage(peer, signedMessage)
function receiveAndVerifyRandomMessage (peer, message):
    var verified = verifyMessage(peer.publicKey, message)
    if (verified) then
        updateNetworkGraph(peer, message)
function updateNetworkGraph (peer, message):
    networkGraph[peer.publicKey] = { publicKey: peer.publicKey,
    address: peer.address,
    status: 'online',
    lastMessage: message,
    lastUpdated: Date.now() }
```

Algorithm 2: Network Graph Transaction functions

```
function sendTransaction (origin, destination, message):
    /* Construct the transaction with path and encrypted
       instructions */
    var transaction = constructTransaction(origin, destination,
      message)
    /* Add the transaction to the list of unconfirmed transactions
       */
    transactions.push(transaction)
/* Function to process unconfirmed transactions and add them to
   blocks */
function processTransactions ():
    var producers = getProducers()
    for (var i = 0; i < transactions.length; i++) do
        var transaction = transactions[i]
        /* Add the transaction to each producer's block */
        for (var j = 0; j < producers.length; j++) do
            var producer = producers[j]
            var added = addTransactionToBlock(producer,
              transaction)
            if (added) then
                /* Remove the transaction from the list of
                   unconfirmed transactions */
                transactions.splice(i, 1)
                i-
                break
function constructTransaction (origin, destination, message):
    /* Find the shortest path between the origin and destination
       */
    var shortestPath = findShortestPath(origin, destination)
    var encryptedMessage = encryptMessage(message,
      destination.publicKey)
    var transaction = {
    path: shortestPath,
    instructions: encryptedMessage,
    } /* Return the constructed transaction */
    return transaction
/* Function to add a new producer to the network */
function add_producer (public_key, allocated_blocks):
    producer = (public_key, allocated_blocks)
    producers.append(producer)
```

Algorithm 3: Network Graph Topology

```
/* Function to get the network topology from a given reference
   node */
function get_topology (reference_node):
    topology = [ ]
    for node in nodes do
        if node != reference_node then
            path = shortest_path(reference_node, node)
            topology.append((node, path))
    return topology
```

Algorithm 4: Path Finding

```
function findPath (fromNode, toNode):
    /* Find a route from Node to Node */
    paths = getAllPaths(fromNode)
    routes = [ ]
    for path in paths do
        /* Check if the path is connected to the destination node
           */
        if path.toNode == toNode then
            return [path]
            /* Try to find a route from the destination node
               through this channel */
            route = findPath(path.toNode, toNode)
            if route is not None then
                /* Add this path to the route */
                routes.append([path] + route)
        /* Return the route */
        if len(routes) > 0 then
            return (routes)
            else
                return None
```

Algorithm 5: Onion Peeling

```
function onion_path (mint_hash, route):
    /* Get the next hop path in the route */
    next_path = route.pop()
    packet = create_onion_packet(mint_hash, next_path)
    for path in reversed(route) do
        eph_key = generate_ephemeral_key()
        packet = add_path_to_onion_route(path, eph_key, packet)
    send_packet_to_next_hop_path(packet, next_path)
    response = receive_response_from_next_hop_path()
    for path in reversed(route) do
        response = decrypt_response_with_ephemeral_key(response,
          path, eph_key)
    return response

/* Notes:  The onion peeling algorithm is used to protect the
   privacy of the mint route, by encrypting the mint information
   multiple times, with each layer containing information for the
   next hop.  As the payment packet is passed from hop to hop,
   each node removes a layer of encryption to reveal the next hop
   in the route.

   • mint_hash is the unique identifier for the minted transaction
   • route is a list of the nodes in the mint route
   • add_path_to_onion_packet function adds a new layer to the onion
     packet for the current hop
   • ephemeral key will be used to decrypt the response from that hop

                                                                  */
```

Algorithm 6: Node Weights

```
bandwidth = x
block_size_limit = 1000000 /* in bytes */

node_weights = {}
/* Scan the blockchain from the genesis block to the current block */
for each block in blockchain do
    proof_utxo = get_bandwidth_proof_utxo(block)
    proof_data = get_proof_data(proof_utxo)
    node_weight = calculate_weight(proof_data, bandwidth)
    fork_proof = get_fork_proof(block)
    /* Add the node weight to the temporary storage for the current node */
    node_weights[block.node_id] = node_weight
    if fork_proof is not None then
        prover_node = fork_proof.prover_node
        forker_node = fork_proof.forker_node
        node_weights[prover_node] += 0.01 * block_size_limit
        node_weights[forker_node] -= 0.01 * block_size_limit
    else
        node_weights[block.node_id] -= 0.01 * block_size_limit
    /* Block added successfully */
    if block then
        Node_weights[producer_node] += 0.01 * block_size_limit
    continue
```

Algorithm 7: Adding new block

```
chain = [ ]
ring_size = 1
block_size_limit_per_sec = 0
set_weights = [ ]
confirm_snips = false
function add_new_block():
    new_block = get_new_block()
    last_block = get_last_block(chain)
    new_hash_proof = last_block.hash_proof
    new_block.hash_proof = new_hash_proof
    if new_hash_proof.node_weight >
      last_block.hash_proof.node_weight then
        /* Find the snips to remove by linearly hashing one by one snip */
        new_snips = last_block.snips
        for snip in last_block.snips do
            if linear_hash(snip) == new_hash_proof.MCR_output
              then
                break
            new_snips.remove(snip)
        new_block.snips = new_snips
    if block_time(new_block) or block_size_capped(new_block) or
      end_snip(new_block) then
        chain.append(new_block)
```

Algorithm 8: Set new ring Validators

```
function set_ring_size(new_block):
    if is_confirmed(new_block) then
        if is_forked(new_block) then
            ring_size -= 1
            end_election()
        else
            ring_size += 1
            tail_join_req = 2
            set_ring_size(ring_size)
        return ring_size
function set_ring_validators():
    set_weights = sorted(nodes, key=lambda node: node["weight"],
      reverse=True)
    set_weights = [n for n in set_weights if n not in
      prev_ring_validators]
    prev_ring_validator_weights = [n.weight for n in
      prev_ring_validators if n.weight ≥ 0]
    mean_weight = mean(prev_ring_validator_weights)
    maxima_rent_rates = 1.1*maxima(prev_ring_validator_rent_rates)
    tail_join = mean_weight
    k = calculate_MD160hash(new_block)
    while len(set_weights) < 2 do
        set_weights = [n for n in set_weights if n.bandwidth >
          tail_join and n.rent_rate < maxima_rent_rate]
    /* Current hex should be lesser than k */
    Valid_keys = [ ]
    for i in range(len(set_weights)) do
        if set_weights[i].pubkey.hex < k then
            valid_keys.append(set_weights[i].pubkey)
    Rand1, rand2 = get2_random_numbers_in_range(0,
      len(valid_keys)-1)
    pubkeys.append(valid_keys[rand1])
    pubkeys.append(valid_keys[rand1])
    /* If none, take immediate greater 2 values */
    if pubkeys is None then
        Valid_keys = sorted(nodes, key=lambda node: set_weights,
          reverse=false)
        pubkeys.append(valid_keys[0])
        pubkeys.append(valid_keys[1])
    ring_validators = set_weights
    ring_validators.append(keys for key in pubkeys)
```

Algorithm 9: Confirm Snips

```
function confirm_snips():
    block_size_limit = min([node.bandwidth for node in
      ring_validators])
    block_times = [block.time for block in previous_blocks]
    block_time_median = median(block_times)
    per_block_size = block_size_limit * block_time_median
    per_block_time_count = int(per_block_size / ihr)
    /* Set a cap that not more than the individual block time the producer should produce */
    max_individual_block_time_count = cap_value
    if per_block_time_count > max_individual_block_time_coun then
        per_block_time_count = max_individual_block_time_count - 1
        confirm_snips = true
    else
        confirm_snips = false
```

Algorithm 10: Merkle Chain

```
class MerkleChain
pre: the snip is added to the data
post: the data is added to the chain
add_node(snip)
d ← snip
if head = null then
    head,tail ← add_data(d)
else
    tail ← add_data(d)
---
class add_data(d)
pre: the value is added to the vector
post: the vector is generated to a Merkle tree and added to the chain
New Vector data
data ← d
if size(data) == max_block_size then
    generate_root(data)
---
generate_root()
pre: the vector data is added as the leaves
post: merkel tree and its root is generated
New Vector temp_data
temp_data ← data
while temp_data > 1 do
    for i = 0 i < size(temp_data) i+2 do
        Left ← temp_data[i]
        Right ← (i+1 == size(temp_data)) ? temp_data[i] :
          temp_data[i+1]
        combined = Left + Right
        new_temp_data ← hash(combined)
    temp_data ← new_temp_data
node_root ← temp_data[0]
---
main()
initialized: chain is an object of class MerkleChain and string data
while true do
    Output "enter data (q to quit)" Get data
    if data = q then
        break
        else
            addnode(data)
```

Algorithm 11: Hash Proofs : helper functions

```
function reject_snips():
    new_block_hash = produce_block(prev_block_hash,
      current_block_snips, current_block_time)
    send_block_to_network(new_block_hash)
    /* Reset variables for new block */
    current_block_snips = []
    current_block_size = 0
    current_block_time = 0
    prev_block_hash = new_block_hash
    snips_received = false
function accept_snips():
    /* single threaded hash concatenate */
    routing_instruction = get_routing_instruction()
    snip_data = receive_snip_data()
    preimage = generate_preimage(snip_data, prev_snip_hash)
    signature = sign_preimage(preimage)
    hashed_data = hash(concatenate(preimage, signature))
    send_snip_to_next_node(routing_instruction, hashed_data)
    current_block_snips.append(hashed_data)
    current_block_size += get_snip_size(hashed_data)
    current_block_time = get_current_block_time()
    prev_snip_hash = hashed_data
    mcr = produce_mcr(snips)
    block_header.add(mcr)
    snips_received = true
```

Algorithm 12: Hash Proofs

```
prev_snip_hash = null
prev_block_hash = genesis_block_hash
current_block_size = 0
current_block_snips = [ ]
current_block_time = 0
block_size_limit_per_sec = initial_block_size_limit_per_sec
snips_received = confirm_snips() /* snips_algo-3 */
while true do
    if snips_received then
        if current_block_size ≥ block_size_limit_per_sec ∗
          current_block_time or current_block_time ≥
          individual_block_time_cap then
            reject_snips()
        /* Move on to next snip */
        current_snip = next_snip()
        else
            accept_snips()
    else
        accept_snips()
```

Algorithm 13: Hash Reward

```
initial_reward = 50 ∗ 10∗∗8 /* example 50 BTC */
halving_period = 210_000 /* example blocks */
/* Set the starting block height and the total number of remaining
   blocks */
block_height = 0
remaining_blocks = halving_period
percent_hash_rate = 0
all_nodes_IHR = 100000 /* example total IHR of all nodes */
while true do
    /* Calculate the total number of remaining coins and remaining
       hashes */
    remaining_coins = initial_reward ∗ remaining_blocks
    remaining_hashes = remaining_blocks ∗ 1.26 ∗ 10∗∗8
    percent_hash_rate = get_node_IHR()/all_nodes_IHR
    /* Calculate the reward per block and the reward per hash */
    reward_per_block = remaining_coins / remaining_blocks
    if fork_slash then
        reward_per_hash = (remaining_coins / remaining_hashes) ∗
          (percent_hash_rate)
    else
        remaining_coins = remaining_coins + (remaining_coins /
          remaining_hashes) ∗ (percent_hash_rate)
    /* Check if it's time to halve the rewards */
    if remaining_blocks ≤ 0 then
        break
    /* Halve the remaining blocks and update the block height */
    remaining_blocks /* = 2 */
    block_height += halving_period
```

Algorithm 14: Transfer Fee

```
transfer_fee = 0.0005 /* Transfer fee should be in range 0.0005 to
   0.00005 */
function check_range(transaction_fee):
    if (transaction_fee > 0.0005) then
        return 0.0005
    if (transaction_fee < 0.00005) then
        return 0.00005
/* called for every nth block , where n is the ring size */
function transaction_fee():
    /* Find the total volume of all blocks of all tokens with
       their exchange rate */
    ring_volume1 = get_volume_of_tokens(block → previous)
    ring_volume2 = get_volume_of_tokens(block → previous →
      previous)
    if standard_deviation(ring_volume1, ring_volume2) ≥ 0.75 then
        if ring_volume1 > ring_volume2 then
            transfer_fee += 0.000005
            transfer_fee = check_range(transaction_fee)
        if ring_volume1 < ring_volume2 then
            transfer_fee -= 0.000005
            transfer_fee = check_range(transaction_fee)
    else
        continue
```

Algorithm 15: ZK IHR Circuit

```
/* Public signals */
signal input: node_ihr
signal input: ihr_hash
/* Private signals */
signal input: salt
signal input: required_ihr
/* Output signal */
signal output: if_pass
/* Range proof check */
signal buffer
signal range_check
if node_ihr > required_ihr - buffer and node_ihr < required_ihr +
  buffer then
    range_check = true
/* Verify hash */
signal hash
signal hash_check
/* RIPEMD160 to calculate the hash */
hash = RIPEMD160 (salt, required_ihr)
if hash == ihr_hash then
    hash_check = true
if range_check and hash_check then
    if_pass = true
else
    if_pass = false
/* Bandwidth circuit ≡ IHR circuit */
```

Algorithm 16: Open Order Script Deploy

```
declare token_a as integer
declare seller as PubKey
declare token_b as integer
declare mature_time as integer
set mature_time as expiry_time
function order (sig, b, buyer, current_exchange_rate_value,
  preimage):
    if mature_time > SigHash.nLocktime(preimage) then
        if checkSig(sig, buyer) then
            if Tx.checkPreimage(preimage) then
                if b == this.token_b then
                    scriptCode = SigHash.scriptCode(preimage)
                    codeend = 104
                    codepart = scriptCode[:104]
                    outputScript_send = codepart + buyer +
                      num2bin(this.token_a, 8) +
                      num2bin(current_exchange_rate_value, 8) +
                      num2bin(tds, 8)
                    output_send =
                      Utils.writeVarint(outputScript_send)
                    outputScript_receive = codepart + this.seller +
                      num2bin(this.token_b, 8) +
                      num2bin(current_exchange_rate_value, 8) +
                      num2bin(tds, 8)
                    output_receive =
                      Utils.writeVarint(outputScript_send)
                    hashoutput =
                      hash256(output_send+output_receive)
                    if hashoutput ==
                      SigHash.hashOutputs(preimage) then
                        /* order is open & placed */
```

Algorithm 17: Open Order Claim

```
function claim (sig, value, pubKey, current_exchange_rate_value,
  preimage):
    if mature_time < SigHash.nLocktime(preimage) then
        if pubKey == this.seller then
            if checkSig(sig, pubKey) then
                if Tx.checkPreimage(preimage) then
                    if value == this.token_a then
                        scriptCode =
                          SigHash.scriptCode(preimage)
                        codeend = 104
                        codepart = scriptCode[:104]
                        outputScript_claim = codepart + pubKey
                          + num2bin(this.token_a,8) +
                          num2bin(current_exchange_rate_value,8) +
                          num2bin(tds, 8)
                        output_claim =
                          Utils.writeVarint(outputScript_claim)
                        hashoutput = hash256(output_claim)
                        if hashoutput ==
                          SigHash.hashOutputs(preimage) then
                            /* claim is successful */
```

Algorithm 18: Bitcoin Exchange & Demand Rate

```
function update_token_price_list (open_order_list: List[List[str]]) ←
  Dict[str, Dict[str, float]]:
    token_price_dict = {}
    for each order in open_order_list do
        token_pair = order[0]
        token_id = token_pair.split ('/')[0]
        bitcoin_rate = float (order[1])
        token_rate = calculate_mid_market_price (float(order[2]),
          float(order[3]))
        percentage_movement = calculate_percentage_movement
          (float(order[4]), token_rate)
        if token_pair not in token_price_dict then
            token_price_dict[token_pair] = {'exchange_rate':
              token_rate, 'percentage_movement':
              percentage_movement}
        else
            token_price_dict[token_pair]['exchange_rate'] = token_rate
            token_price_dict[token_pair]['percentage_movement'] =
              percentage_movement
    return token_price_dict
function cal_bdr (token_price_dict):
    token_pairs = [pair for pair in token_price_dict if pair[0] !=
      "00000000"]
    total_volume = 0
    for each pair_info in token_price_dict.values() do
        total_volume = total_volume + pair_info['volume']
    for each pair in token_pairs do
        pair_info = token_price_dict[pair]
        weight = pair_info['volume'] / total_volume
        pair_info['weight'] = weight
    for each pair_info in token_price_dict.values() do
        pair_info['inv_pct_mov'] = -pair_info['pct_mov']
    bdr_pct_mov = 0
    for each pair_info in token_price_dict.values() do
        bdr_pct_mov = bdr_pct_mov + (pair_info['inv_pct_mov'] ∗
          pair_info['weight'])
    bdr = 1 + (bdr_pct_mov / 100)
    return bdr
```

Algorithm 19: Tax Script

```
Key: signature, amount, current_exchange_rate,
     preimage_of_signature, tax_percent
Output: updated stateful contract for the sender & new stateful
        contract for the receiver
DataLen = 1
utxo_amount ← initial_amount
pubKey ← pubkey of the sender
initial_exchange_rate ← initial exchange rate of the token
region_code ← region code of the person
tds ← TDS
Function spend (sig, amount, current_exchangerate, tax_percent,
  receiver_pubkey, preimage):
    if checkSig(sig, pubKey) and Tx.checkPreimage(preimage)
      and check_regiontax(region_code,tax_percent) then
        scriptCode ← SigHash.scriptCode(preimage)
        codeend ← position where the opcode ends
        codepart ← scriptCode[:codeend]
        percentage_movement ←
          get_percentage_movement(initial_exchangerate,
          current_exchangerate)
        if percentage_movement > 0 then
            gains ← (percentage_movement ∗ (tax_percent ∗ 10^-2) ∗
              utxo_amount) /(percentage_movement + 1)
            spendable_amount ← utxo_amount - gains -tds
        else
            spendable_amount ← utxo_amount - tds
        if amount ≤ spendable_amount and sender == pubKey
          and amount ≥ 0 then
            utxo_amount ← utxo_amount - amount
    updated_script ← codepart + utxo_amount+sender +
      current_exchange_rate + tds
    new_script ← codepart+utxo_amount + receiver_pubkey +
      current_exchange_rate + tds
    hash ← sha256(updated_script+new_script)
    if hash == SigHash.hashOutputs(preimage) then
        true
```