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
```