REFERENCE ALGORITHMS

Purva Choudhari, Ajay Joshua*

February 27, 2023 (Genesis Version)

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 from Node to 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)
    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
    /* Add the node weight to the temporary storage for the
       current node */
    node_weights[block.node_id] = node_weight
    continue
```

Algorithm 7: Snip Construction

Algorithm 8: Hash Proofing

Algorithm 9: Hash Reward

*ajay@projectblink.org

Algorithm 10: 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 11: 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 12: 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 13: 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 14: Exchange Rate Calculation

```
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 15: 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
exchange_rate ← initial_exchange_rate
tds ← TDS
Function spend (sig, amount, current_exchange_rate, tax_percent,
  receiver_pubkey,preimage):
    if checkSig(sig, pubKey) and Tx.checkPreimage(preimage) then
        scriptCode ← SigHash.scriptCode(preimage)
        codeend ← position where the opcode ends
        codepart ← scriptCode[:codeend]
        gains ← (amount * current_exchange_rate) - (amount *
          exchange_rate)
        if gains > 0 then
            amount ← amount -
              (gains*(tax_percent/100))*(current_exchange_rate)
            if amount ≤ (amount − tds) 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
```