9 Prototype Verification for Certificate Validation （2）

9.1 Block Information Verification

9.1.1 Signature Verification

Verify the Merkle Tree Using the Cardano Blockchain

Prototype : Source Code （verify_gen_merkle.js）

```
const axios = require('axios');
const blake = require('blakejs');
const fs = require('fs');

require('dotenv').config();

// Fetch transaction information for the specified block from the
Blockfrost API
async function fetchBlockTransactions(height, apiKey) {
    try {
        // Retrieve the block's transactions
        const response = await axios.get(`https://cardano-
preprod.blockfrost.io/api/v0/blocks/${height}/txs`, {
            headers: { 'project_id': apiKey }
        });
        return response.data;
    } catch (error) {
        console.error('Error fetching block transactions:', error);
        return [];
    }
}
```

```
// Generate hash values from the array of transactions
function getTransactionHashes(transactions) {
    return transactions.map(tx => blake.blake2bHex(tx, null, 32));
}

// Concatenate two hashes and generate a new hash
function hashConcat(left, right) {
    return blake.blake2bHex(left + right, null, 32);
}

// Build a Merkle tree
function buildMerkleTree(hashes) {
    let tree = [hashes];

    // Construct each level of the Merkle tree
    while (tree[tree.length - 1].length > 1) {
        let currentLevel = tree[tree.length - 1];
        let nextLevel = [];

        // Concatenate hashes in pairs to create the next level
        for (let i = 0; i < currentLevel.length; i += 2) {
            const left = currentLevel[i];
            const right = i + 1 < currentLevel.length ? currentLevel[i +
1] : left;
            nextLevel.push(hashConcat(left, right));
        }

        tree.push(nextLevel);
    }

    return tree;
}
```

```
// Find the Merkle path for the specified transaction hash
function findPath(tree, targetHash) {
    let path = [];
    let currentHash = targetHash;

    // At each level, construct the path while recording the hash adjacent
to the target hash
    for (let level = 0; level < tree.length - 1; level++) {
        const currentLevel = tree[level];
        for (let i = 0; i < currentLevel.length; i += 2) {
            const left = currentLevel[i];
            const right = i + 1 < currentLevel.length ? currentLevel[i +
1] : left;
            const combinedHash = hashConcat(left, right);

            // If the target hash is on the left or right, add the adjacent
hash to the path
            if (left === currentHash || right === currentHash) {
                path.push({
                    position: left === currentHash ? 'left' : 'right',
                    hash: left === currentHash ? right : left
                });
                currentHash = combinedHash;
                break;
            }
        }
    }

    return path;
}

(async () => {
    const apiKey = process.env.PROJECT_ID;
    const height = 2528206;
```

```
    const specificTx =
'aa92a3df4bbfacf22108b66f2d6a245002ae8501b51240fb1df8bbab9a759e29';

    // Fetch transactions from the specified block
    const transactions = await fetchBlockTransactions(height, apiKey);
    if (transactions.length === 0) {
        console.error('No transactions found');
        return;
    }

    console.log('Transaction IDs:', transactions);

    // Calculate the transaction hashes
    const transactionHashes = getTransactionHashes(transactions);
    console.log('Transaction Hashes:', transactionHashes);

    // Build the Merkle tree
    const tree = buildMerkleTree(transactionHashes);

    console.log('Merkle Tree:');
    tree.forEach((level, index) => {
        console.log(`Level ${index}:`, level);
    });

    // Get the Merkle root (the topmost hash of the tree)
    const root = tree[tree.length - 1][0];
    console.log('Merkle Root:', root);

    // Calculate the hash of a specific transaction
    const targetHash = blake.blake2bHex(specificTx, null, 32);
    console.log('Specific Transaction Hash:', targetHash);

    // Retrieve the Merkle path
    const path = findPath(tree, targetHash);
```

```
    console.log('Merkle Path:', JSON.stringify({ merklePath: path }, null,
2));

    // Export as a JSON file
    const jsonOutput = JSON.stringify({ merklePath: path }, null, 2);
    fs.writeFileSync('merkle_path.json', jsonOutput);
    console.log('Merkle path saved to merkle_path.json');
})();
```

Execution result

```
[ec2-user@ip-172-31-19-190 research]$ node verify_gen_merkle.js
Transaction IDs: [
  '79d415fe3ce94447edec54de6496239ad08327280c091bcb26a33e17172e10ec',
  'aa92a3df4bbfacf22108b66f2d6a245002ae8501b51240fb1df8bbab9a759e29'
]
Transaction Hashes: [
  'db6cad6ca79e25d75fafd14912936f3d53b9571d499439eca8c69938efa45a1d',
  '5bdb3a75f12ffb47d37750d91173917caf2db251b7fe1d5c71cd5f1eccf406a8'
]
Merkle Tree:
Level 0: [
  'db6cad6ca79e25d75fafd14912936f3d53b9571d499439eca8c69938efa45a1d',
  '5bdb3a75f12ffb47d37750d91173917caf2db251b7fe1d5c71cd5f1eccf406a8'
]
Level 1: [ '701270a92ca73abaa563a7279720f9684f1d28c36fa333a2a6b0c030c44bd233' ]
Merkle Root: 701270a92ca73abaa563a7279720f9684f1d28c36fa333a2a6b0c030c44bd233
Specific Transaction Hash: 5bdb3a75f12ffb47d37750d91173917caf2db251b7fe1d5c71cd5f1eccf406a8
Merkle Path: {
  "merklePath": [
    {
      "position": "right",
      "hash": "db6cad6ca79e25d75fafd14912936f3d53b9571d499439eca8c69938efa45a1d"
    }
  ]
}
Merkle path saved to merkle_path.json
```

Example of a Generated JSON File

```
[ec2-user@ip-172-31-19-190 node-apps]$ cat merkle_path.json
{
  "merklePath": [
    {
      "position": "right",
      "hash": "db6cad6ca79e25d75fafd14912936f3d53b9571d499439eca8c69938efa45a1d"
    }
  ]
```

9.1.2 Merkle Tree Verification

Using the Cardano Blockchain, Verify the Validity of Certificates with the Generated Information and Block Information Received via IPDC

Prototype : Source Code（verify_merkle.js）

```
const fs = require('fs');
const blake = require('blakejs');

// Concatenate two hashes and return the BLAKE2b hash of the result
function hashConcat(left, right) {
    return blake.blake2bHex(left + right, null, 32);
}

// Verify the Merkle path from a leaf node to the expected root
function verifyMerklePath(leafHash, path, expectedRoot) {
    let currentHash = leafHash;

    for (let node of path) {
        if (node.position === 'left') {
            currentHash = hashConcat(currentHash, node.hash);
        } else if (node.position === 'right') {
            currentHash = hashConcat(node.hash, currentHash);
        } else {
            console.error(`Unknown position ${node.position}`);
            return false;
        }
    }

    // Check if the computed hash matches the expected Merkle root
    return currentHash === expectedRoot;
}
```

```
(async () => {
    const specificTx =
'aa92a3df4bbfacf22108b66f2d6a245002ae8501b51240fb1df8bbab9a759e29';

    // Compute the leaf hash from the specific transaction using BLAKE2b
    const leafHash = blake.blake2bHex(specificTx, null, 32);
    console.log("leaf hash:",leafHash)

    // Load the path from `merkle_path.json`
    const merklePath = JSON.parse(fs.readFileSync('merkle_path.json',
'utf8')).merklePath;
    console.log("input merkle path: ",merklePath)

    // Set the expected Merkle root (using the previously calculated root)
    const expectedRoot =
'701270a92ca73abaa563a7279720f9684f1d28c36fa333a2a6b0c030c44bd233';
    console.log("merkle root hash: ",expectedRoot)

    // Verify the path to the Merkle root
    const isValid = verifyMerklePath(leafHash, merklePath, expectedRoot);
    console.log('Is the leaf hash valid in the Merkle tree?', isValid);
})();
```

Execution result

```
[ec2-user@ip-172-31-19-190 node-apps]$ node verify_merkle.js
leaf hash: 5bdb3a75f12ffb47d37750d91173917caf2db251b7fe1d5c71cd5f1eccf406a8
input merkle path:  [
  {
    position: 'right',
    hash: 'db6cad6ca79e25d75fafd14912936f3d53b9571d499439eca8c69938efa45a1d'
  }
]
merkle root hash:  701270a92ca73abaa563a7279720f9684f1d28c36fa333a2a6b0c030c44bd233
Is the leaf hash valid in the Merkle tree? true
```

leaf hash:
5bdb3a75f12ffb47d37750d91173917caf2db251b7fe1d5c71cd5f1eccf406a8

merkle root hash:
701270a92ca73abaa563a7279720f9684f1d28c36fa333a2a6b0c030c44bd233

Is the leaf hash valid in the Merkle tree? true

9.1.3 Verification Policy

In this program, the calculated leaf hash (leafHash) based on a specific transaction is compared with the expected Merkle Root (expectedRoot) using the Merkle Tree path.

9.1.3.1. Calculation of the Leaf Hash

In the first part of the program, the hash of a specific transaction is calculated using BLAKE2b. This hash corresponds to a leaf node in the Merkle Tree.

```
javascript
コードをコピーする
const leafHash = blake.blake2bHex(specificTx, null, 32);
```

9.1.3.2. Reading the Merkle Path

Next, the Merkle Path is read from the merkle_path.json file. This path contains the node information from the leaf node to the root node. Each node has the position information (left or right) and the hash of that node.

```
javascript
コードをコピーする
const merklePath = JSON.parse(fs.readFileSync('merkle_path.json', 'utf8')).merklePath;
```

9.1.3.3. Setting the Expected Merkle Root

In the program, the previously calculated Merkle Root is set. This root is the topmost hash of the Merkle Tree and serves as the benchmark for verifying the validity.

```
javascript
コードをコピーする
const expectedRoot = '701270a92ca73abaa563a7279720f9684f1d28c36fa333a2a6b0c030c44bd233';
```

9.1.3.4. Verification of the Merkle Path

Finally, the verifyMerklePath function is called, and the leaf hash, Merkle path, and the expected Merkle root are passed in. In this function, starting from the leaf hash, the Merkle path is traversed, and the hashes of each node are concatenated.

```
javascript
コードをコピーする
const isValid = verifyMerklePath(leafHash, merklePath, expectedRoot);
```

At each step, the generated hash is concatenated with the hash at the specified position (left or right). Finally, it checks whether the computed hash matches the expected Merkle Root.

9.1.3.5. 検証結果

If they match, isValid becomes true, indicating, "This information (leaf hash) matches this information (expected Merkle Root), so the certificate is valid."
If they do not match, isValid becomes false, meaning that the certificate is not valid.
In this way, the program traces the hash from the leaf node to the expected root, and based on the final result, confirms the validity of the certificate.

以上