7 Verification after IPDC Reception（3-b）

7.1 Research on Signature Verification

7.1.1 Research Objective

We will confirm the process to verify the integrity of transaction data on the Cardano blockchain and strengthen its reliability. By utilizing Merkle Tree and the Blake2b hash function to generate the Merkle Path for a specific transaction and saving the results as a JSON file, blockchain data can be verified and audited in an efficient and highly reliable manner.

<Generation and Verification of Merkle Path>
To prove that a specific transaction is correctly included in the blockchain, it is necessary to generate the Merkle Path and verify the transaction. By saving the result of the Merkle Path generation as a JSON file, the generated Merkle Path can be used for future audits and data verification, thus strengthening the reliability of the blockchain.

<Ensuring Data Persistence and Reusability with JSON Files>
By exporting the generated Merkle Path as a JSON file, data persistence and reusability are ensured. This provides a concrete method to later verify transaction integrity or allow third-party audits. Saving it in a file format makes it easier to verify data across different systems and environments, supporting long-term data management.

7.1.2 Libraries used and environment

<Libraries used>

axios

: Used to make HTTP requests and retrieve data from the Blockfrost API.

blakejs

: Using the Blake2b hash function, we perform transaction hash calculation and construct a Merkle Tree.

fs

: Using the Node.js standard library for file system operations, the generated Merkle Path is saved as a JSON file.

<environment>

This is JavaScript code that runs in the Node.js environment.
An API key is required to use the Blockfrost API.
The fs library is used to save the Merkle Path to a file.

7.1.3 Basic Research Content

Getting Transactions

```
async function fetchBlockTransactions(height, apiKey) {
    try {
        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 [];
    }
}
```

fetchBlockTransactions Function
: Retrieves transaction IDs using the Blockfrost API based on the specified block height (height). Authentication is performed using the API key (apiKey).

Error Handling
: If the API request fails, an error message is displayed and an empty array is returned.

Transaction ID Hash Calculation

```
function getTransactionHashes(transactions) {
    return transactions.map(tx => blake.blake2bHex(tx, null, 32));
}
```

getTransactionHashes Function

: Hashes the retrieved transaction IDs using the Blake2b hash function, generating 32-byte hash values

Hash Concatenation and Merkle Tree Construction

```
function hashConcat(left, right) {
    return blake.blake2bHex(left + right, null, 32);
}

function buildMerkleTree(hashes) {
    let tree = [hashes];
    while (tree[tree.length - 1].length > 1) {
        let currentLevel = tree[tree.length - 1];
        let nextLevel = [];

        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;
}
```

hashConcat Function

: Concatenates the left and right hashes, then re-hashes them using the Blake2b hash function. This is used when constructing the upper levels of the Merkle Tree.

buildMerkleTree Function

: Constructs a Merkle Tree from transaction hashes. At each level, from the leaves to the root, it concatenates the hashes, ultimately generating a single root hash (Merkle Root).

Searching of Merkle Path

```
function findPath(tree, targetHash) {
    let path = [];
    let currentHash = targetHash;

    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 (left === currentHash || right === currentHash) {
                path.push({
                    position: left === currentHash ? 'left' : 'right',
                    hash: left === currentHash ? right : left
                });
                currentHash = combinedHash;
                break;
            }
        }
    }
    return path;
}
```

findPath Function

: Generates the Merkle Path corresponding to a specific transaction hash. It tracks the hash concatenation at each level and creates a path to the Merkle Root.

Generating Merkle Path Data

```
(async () => {
    const apiKey = process.env.PROJECT_ID; // BlockfrostのAPIキー
    const height = 2528206;
    const specificTx = 'aa92a3df4bbfacf22108b66f2d6a245002ae8501b51240fb1df8bbab9a759e29';

    const transactions = await fetchBlockTransactions(height, apiKey);
    if (transactions.length === 0) {
        console.error('No transactions found');
        return;
    }

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

    const transactionHashes = getTransactionHashes(transactions);
    console.log('Transaction Hashes:', transactionHashes);

    const tree = buildMerkleTree(transactionHashes);

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

```
        console.log(`Level ${index}:`, level);
    });

    const root = tree[tree.length - 1][0];
    console.log('Merkle Root:', root);

    const targetHash = blake.blake2bHex(specificTx, null, 32);
    console.log('Specific Transaction Hash:', targetHash);

    const path = findPath(tree, targetHash);
    console.log('Merkle Path:', JSON.stringify({ merklePath: path }, null, 2));

    // JSONファイルとしてエクスポート
    const jsonOutput = JSON.stringify({ merklePath: path }, null, 2);
    fs.writeFileSync('merkle_path.json', jsonOutput);
    console.log('Merkle path saved to merkle_path.json');
})();
```

Transaction Retrieval

: Call the fetchBlockTransactions function to retrieve the transactions for the specified block.

Hashing Transaction IDs
: Hash the retrieved transaction IDs using the getTransactionHashes function.

Merkle Tree Construction
: Use the buildMerkleTree function to construct a Merkle Tree based on the hashed transaction IDs.

Merkle Path Search
: Use the findPath function to generate the Merkle Path for a specific transaction.

Export to JSON File
: Save the generated Merkle Path in JSON format to a file.

7.1.4 Research Results

With this code, transactions were retrieved from the specified block, the transaction IDs were hashed using the Blake2b hash function, and a Merkle Tree was constructed. Then, the Merkle Path for the specified transaction ID was searched, and the result was exported as a JSON file.

Merkle Path Generation
: Accurately generates the Merkle Path corresponding to a specific transaction.

File Saving
: Saves the generated Merkle Path as a JSON file, allowing for later verification.

7.2 Research on Merkle Tree

7.2.1 Research Objective

The purpose of this research is to understand and implement practical methods to verify the integrity of specific transaction data on the Cardano blockchain and ensure its reliability.
By using the Blake2b hash function and Merkle Path, we will confirm that blockchain data is correctly stored and has not been tampered with, thereby enhancing the security of the blockchain system.

<**Understanding the Importance and Application of Merkle Path**>
Merkle Path plays a crucial role in verifying blockchain data. In this research, we will explore methods for using Merkle Path to verify whether a specific transaction correctly reaches the Merkle Root in a Merkle Tree. By doing so, we aim to gain a deep understanding of the structure and application of Merkle Path and apply this knowledge to the data verification process in actual blockchain operations.

<**Application and Effectiveness of the Blake2b Hash Function**>
We will calculate the transaction hash using the Blake2b hash function and verify whether it matches the expected Merkle Root based on the correct Merkle Path.

7.2.2 Libraries used and environment

<Libraries used>

fs

: Used to read a JSON file containing the Merkle Path using the Node.js standard library for file system operations.

blakejs

: Using the Blake2b hash function to calculate the transaction hash and verify the Merkle Path.

<environment>

This is JavaScript code that runs in the Node.js environment.
The Merkle Path to be verified needs to be saved as the merkle_path.json file.

7.2.3 Basic Research Content

Hash Concatenation

```
function hashConcat(left, right) {
    return blake.blake2bHex(left + right, null, 32);
}
```

hashConcat Function

: Concatenates two hash values (left and right), then hashes the result using the Blake2b hash function. This is used at each node of the Merkle Path to combine the left and right hashes and calculate the hash for the next level.

Merkle Path Verification

```
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;
        }
    }

    return currentHash === expectedRoot;
}
```

verifyMerklePath Function

: Verifies whether the specified Merkle Path leads the hash of the given transaction (leafHash) to the expected Merkle Root (expectedRoot).
Based on the position of each node, the function concatenates the hashes and calculates the upper-level hash. If the final calculated hash matches expectedRoot, it confirms that the transaction is correctly included in the Merkle Tree.

Verification Process (Matching Process)

```
(async () => {
    const specificTx = 'aa92a3df4bbfacf22108b66f2d6a245002ae8501b51240fb1df8bbab9a759e29';
    const leafHash = blake.blake2bHex(specificTx, null, 32);

    const merklePath = JSON.parse(fs.readFileSync('merkle_path.json', 'utf8')).merklePath;
    const expectedRoot = '701270a92ca73abaa563a7279720f9684f1d28c36fa333a2a6b0c030c44bd233';
    const isValid = verifyMerklePath(leafHash, merklePath, expectedRoot);
    console.log('Is the leaf hash valid in the Merkle tree?', isValid);
})();
```

Hash Calculation for a Specific Transaction

: The hash value of the specified transaction ID (specificTx) is calculated using the Blake2b hash function, and it is stored as leafHash.

Reading the Merkle Path

: The Merkle Path is read from the merkle_path.json file. This file contains a previously generated Merkle Path.

Verification of the Merkle Path

: Using the verifyMerklePath function, it verifies whether the calculated leafHash reaches the correct Merkle Root based on the loaded Merkle Path. If the result is true, it confirms that the transaction is correctly included in the blockchain.

7.2.4 Research Results

This code hashes the specified transaction ID and verifies whether the hash correctly reaches the expected Merkle Root through the stored Merkle Path:

Verification of the Merkle Path
: It checks whether the hash of the specific transaction exists at the correct position within the Merkle Tree and ultimately matches the specified Merkle Root.

Output of Verification Results
: The verification result is output to the console. If the result is true, it confirms that the transaction is correctly included in the blockchain.