3 Transaction before sending as IPCD（2-a)

As part of the research on transaction and metadata creation on the Cardano platform, we conducted investigations related to accounts and transactions.

3.1 Research into Account Creation

3.1.1 Research Objective

In the use of the Cardano blockchain, key generation is the most fundamental and important part. The private and public key pair is used for signing and authenticating transactions, forming the core of security. We will review methods for generating keys in a safe and efficient manner in Cardano's key generation process.

<Secure Entropy Generation>

Entropy is the foundation of key generation, and its quality directly affects the security of the generated keys. We will demonstrate how to generate secure entropy using the crypto library in Node.js and review methods to ensure the reliability of the key generation process.

<Account and address management>

BIP32 (Hierarchical Deterministic Wallets) is a standard that enables hierarchical and deterministic key generation. The derivation of account keys and address keys based on the BIP32 standard plays a crucial role in Cardano wallet management. We will clarify the methods for deriving account keys and address keys, and review the specific process for generating addresses from these keys.

3.1.2 Libraries used and environment

<Libraries used>

@emurgo/cardano-serialization-lib-nodejs

: It is used for Cardano key generation and address generation.

crypto

: It is used for generating random entropy with Node.js's standard cryptographic library.

<environment>

We will use Node.js. Use Node.js. Please install the necessary libraries in advance. For example, install them with the following command.

```
npm install @emurgo/cardano-serialization-lib-nodejs crypto
```

3.1.3 Basic Research Content

Generate private and public keys based on the BIP32 format for Cardano, as well as base addresses for the testnet, to understand the basic process of key generation and address generation on the Cardano blockchain.

Entropy Generation

Generate 32 bytes of random entropy. This entropy will be used later as a seed for key generation.

```
const crypto = require('crypto');
const entropy = crypto.randomBytes(32);
```

Private Key Generation

Generate a BIP32 root private key using the generated entropy.

```
const rootKey = CardanoWasm.Bip32PrivateKey.from_bip39_entropy(entropy, '');
```

Derivation of the Account Key

Derive the account key from the root private key. The values used here are based on BIP44.

```
const accountKey = rootKey.derive(1852 | 0x80000000) // Purpose
                          .derive(1815 | 0x80000000) // Coin type (ADA)
                          .derive(0 | 0x80000000);   // Account
```

1852 | 0x80000000	：	Cardano の目的（Purpose）
1815 | 0x80000000	：	ADA のコインタイプ
0 | 0x80000000	：	アカウント番号

Derivation of the Account Key and Address Generation

Derive the address key from the account key and obtain the corresponding public key.
Generate the base address for the testnet. Here, the public key hash is used as the stake credential.

```
const privateKey = accountKey.derive(0).derive(0).to_raw_key();
const publicKey = privateKey.to_public();
const baseAddress = CardanoWasm.BaseAddress.new(
    0, // Testnet = 0, Mainnet = 1
    CardanoWasm.StakeCredential.from_keyhash(publicKey.hash()),
    CardanoWasm.StakeCredential.from_keyhash(publicKey.hash())
).to_address().to_bech32();
```

3.1.4 Research Results

In this process, the key generation based on Cardano's BIP32 format is demonstrated. Starting from entropy, the root key, account key, and address key are sequentially derived, eventually generating the public key and base address. Since this is targeting the testnet, 0 is specified during address generation.
The generated private key, public key, and address are displayed in both Hex and Bech32 formats, making it easy for developers to verify.

3.2 Research on Transaction Generation

3.2.1 Research Objective

We will review transaction generation and signing on the Cardano blockchain. To build advanced applications within the Cardano ecosystem, we will deepen our understanding of key management, the UTXO model, the use of Blockfrost API, protocol parameters, and security.

3.2.2 Libraries used and environment

<Libraries used>

@emurgo/cardano-serialization-lib-nodejs

：Used for Cardano key generation, transaction generation, and signing.

axios

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

Dotenv

: Used to read environment variables.

<environment>

We will use Node.js. Use Node.js. Please install the necessary libraries in advance. For example, install them with the following command.

```
npm install @emurgo/cardano-serialization-lib-nodejs axios dotenv
```

Set the following environment variables in the .env file:

```
PRIVATE_KEY_HEX=your_private_key_hex
PROJECT_ID=your_blockfrost_project_id
```

3.2.3 Basic Research Content

Importing Libraries

Import the necessary libraries.

```
const CardanoWasm = require('@emurgo/cardano-serialization-lib-nodejs');
const axios = require('axios');
require('dotenv').config();
```

Loading Environment Variables

```
const privateKeyHex = process.env.PRIVATE_KEY_HEX;
const projectId = process.env.PROJECT_ID;
```

Retrieve the private key and Blockfrost project ID from the environment variables.

```
const apiBase = "https://cardano-preprod.blockfrost.io/api/v0";
```

Get UTXO

After getting the UTXOs for the specified address, select the largest UTXO and return it. Construct and sign the transaction, then output the signed transaction in Hex format.

```
async function getMaxUTXO(address) {
    try {
        const response = await axios.get(`${apiBase}/addresses/${address}/utxos`, {
            headers: { 'project_id': projectId }
        });
        const utxos = response.data;

        if (utxos.length === 0) {
            throw new Error("No UTXOs found for this address.");
        }

        // 最大のUTXOを選択
        let maxUTXO = utxos[0];
        utxos.forEach(utxo => {
            if (parseInt(utxo.amount[0].quantity) > parseInt(maxUTXO.amount[0].quantity)) {
                maxUTXO = utxo;
            }
        });

        return {
            txHash: maxUTXO.tx_hash,
            txIndex: maxUTXO.tx_index,
            amount: maxUTXO.amount[0].quantity
        };
    } catch (error) {
        console.error('Error fetching UTXOs:', error);
        throw error;
    }
}
```

Getting Protocol Parameters

Get the latest protocol parameters and use them to configure the transaction builder.

```
async function getProtocolParameters() {
    try {
        const response = await axios.get(`${apiBase}/epochs/latest/parameters`, {
            headers: { 'project_id': projectId }
        });
        const data = response.data;

        return CardanoWasm.TransactionBuilderConfigBuilder.new()
            .fee_algo(
                CardanoWasm.LinearFee.new(
                      CardanoWasm.BigNum.from_str(data.min_fee_a.toString()),
                      CardanoWasm.BigNum.from_str(data.min_fee_b.toString())
                  )
              )
              .pool_deposit(CardanoWasm.BigNum.from_str(data.pool_deposit))
              .key_deposit(CardanoWasm.BigNum.from_str(data.key_deposit))
              .max_value_size(data.max_val_size)
              .max_tx_size(data.max_tx_size)
              .coins_per_utxo_word(CardanoWasm.BigNum.from_str(data.coins_per_utxo_word))
              .build();
      } catch (error) {
          console.error('Error fetching protocol parameters:', error);
          throw error;
      }
  }
```

3.2.4 Research Results

Using this code, a transaction on the Cardano blockchain was generated and signed by following these steps:

1. Load the private key and Blockfrost project ID from the environment variables.
2. Generate the public key and address from the private key.
3. Retrieve the largest UTXO for the specified address.
4. Fetch the latest protocol parameters.
5. Build the transaction, using the largest UTXO as input, and set the amount to be transferred after deducting the fees.
6. Sign the transaction and output the signed transaction in Hex format.

3.3 Research on Transaction Submission

3.3.1 Research Objective

Blockfrost API is a powerful tool for interacting with the Cardano blockchain. We will review how to manage the API key, how to create HTTP requests, and how to handle errors when sending transactions on the Cardano blockchain.

3.3.2 Libraries used and environment

<Libraries used>

axios

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

dotenv

: Used to read environment variables.

<environment>

We will use Node.js. Use Node.js. Please install the necessary libraries in advance. For example, install them with the following command.

```
npm install axios dotenv
```

Set the following environment variables in the .env file:

```
PROJECT_ID=your_blockfrost_project_id
```

3.3.3 Basic Research Content

Importing Libraries

Import the necessary libraries.

```
const axios = require('axios');
require('dotenv').config(); // 環境変数を読み込む
```

Getting environment variables and command line arguments

```
const apiKey = process.env.PROJECT_ID; // BlockfrostのAPIキー
const signedTxHex = process.argv[2]; // コマンドライン引数からsignedTxHexを取得

if (!signedTxHex) {
    console.error('Error: Please provide the signed transaction hex string as a command line
argument.');
    process.exit(1);
}
```

Retrieve the Blockfrost API key from the environment variables. Obtain the signed transaction Hex string from the command-line arguments.If a signed transaction is not provided, display an error message and terminate the process.

Transaction Submission Function

```
async function sendTransaction() {
    try {
        const response = await axios.post('https://cardano-preprod.blockfrost.io/api/v0/tx/submit',
Buffer.from(signedTxHex, 'hex'), {
            headers: {
                'Content-Type': 'application/cbor',
                'project_id': apiKey,
            },
        });

        console.log('Transaction sent successfully:', response.data);
    } catch (error) {
        console.error('Error sending transaction:', error.response ? error.response.data :
error.message);
    }
}
```

In the ‘sendTransaction‘ function, the signed transaction is sent to the Blockfrost API.
If successful, a message indicating that the transaction was sent successfully is displayed in the console.
If an error occurs, the error message is displayed in the console.

3.3.4 Research Results

Using this code, the signed transaction was sent to the Cardano blockchain by following these steps:

1. Load the Blockfrost API key from the environment variables.
2. Obtain the signed transaction Hex string from the command-line arguments.
3. Send the signed transaction to the Blockfrost API.
4. Display the success or failure of the transaction submission in the console.

3.4 Research on Transaction hash

3.4.1 Research Objective

By analyzing the transaction ID retrieval process, we gain an understanding of how the Cardano blockchain ensures data integrity.

<Serialization and Deserialization>
We will review important conversion processes in blockchain technology, such as serialization, deserialization, and hash computation. In particular, learning the structure of transactions and how to retrieve transaction IDs is very useful when verifying or auditing Cardano transactions.

<Research on Cardano Development in Browser Environments>
We will explore methods for using the Buffer polyfill and demonstrate the steps to load cardano-serialization-lib in a browser, providing web developers with techniques for developing Cardano-compatible applications.

3.3.2 Libraries used and environment

<Libraries used>

@emurgo/cardano-serialization-lib-asmjs

：Used to deserialize Cardano transactions and calculate the transaction ID.

Buffer

：Used to enable the use of the Buffer functionality from Node.js in a browser environment.

<environment>

JavaScript code that works in a browser environment.

The cardano-serialization-lib uses the asm.js version to be imported directly in the browser.
By using the Buffer polyfill, it allows the use of Node.js-like Buffer objects in the browser as well.

3.4.3 Basic Research Content

Importing Libraries

Import the necessary libraries.

```
const CardanoWasm = await import('https://unpkg.com/@emurgo/cardano-serialization-lib-
asmjs@11.5.0/cardano_serialization_lib.js');
```

Load the cardano-serialization-lib-asmjs library into the browser environment. This library is specialized for Cardano transaction processing.

```
(async () => {
    const bufferScript = document.createElement('script');
    bufferScript.src = 'https://bundle.run/buffer@6.0.3';
    document.head.appendChild(bufferScript);
    await new Promise(resolve => bufferScript.onload = resolve);
    window.Buffer = window.buffer.Buffer;
})();
```

Load the Buffer polyfill into the browser to make the Buffer object available. This is necessary when handling serialized transactions.

Processing Serialized Transactions

```
const serializedTxHex =
'84a40081825820e75ad3545eae1b6fcda81320224f679548dd4fd3206ad1ccda3bdec0485d7f4800018182583900295b238bbbda
```

Define the serialized transaction as a Hex format string. This string represents the complete transaction data.

```
const txBytes = Buffer.from(serializedTxHex, 'hex');
```

Convert the Hex string to a byte array. This allows the transaction data to be handled in binary format.

Deserializing a transaction and getting the transaction ID

```
const transaction = CardanoWasm.Transaction.from_bytes(txBytes);
```

Deserialize the transaction from the byte array. This allows the transaction content to be read and manipulated.

```
const txBody = transaction.body();
```

Hash the transaction body to retrieve the transaction ID. This ensures the uniqueness of the transaction.

```
console.log("Transaction ID:", Buffer.from(txHash.to_bytes()).toString('hex'));
```

The got transaction ID is output to the console in hex format.

3.4.4 Research Results

Using this code, the transaction ID was retrieved from a serialized transaction by following these steps:

1. Receive the serialized transaction in Hex format and convert it into a byte array.
2. Use cardano-serialization-lib to deserialize the transaction.
3. Calculate the transaction ID (hash) from the transaction body.
4. Display the calculated transaction ID in Hex format.