The social platform that realizes blockchain authentication even in environments without the internet

1 Overall structure and overview (1-a)

This structure uses a platform called Narrowcast as an advanced means of information transmission, utilizing conventional television signals to deliver block information for blockchain authentication as public data. It aims to enable public authentication in environments without internet access.

- **Issuance of certificates via the Cardano blockchain and retrieval of generated block information**
- **Information distribution and reception via television signals using Narrowcast (IPDC)**
- **Reconstruction of block information and verification of certificates**

In this demonstration, we will implement reliability verification using broadcast waves in environments where the internet is unavailable. Normally, certificates created with electronic signatures in an internet environment can be verified by utilizing information from a third party, known as a "Certificate Authority (CA)."

However, when a disaster occurs, it becomes difficult to query information from the "Certificate Authority" due to factors such as the collapse of base stations. In this demonstration, we will utilize IPDC technology to superimpose foundational information onto broadcast waves, enabling the verification of data reliability in disaster-affected areas.

Furthermore, unlike regular electronic signatures, the use of blockchain ensures that the proof of existence for verified data will persist into the future. This can serve as a basis for maintaining economic activities even in situations where access to various databases is cut off during disasters.

1.1 Overall Configuration

The system developed in this demonstration will consist of a ”Certificate Generation Subsystem” and a ”Certificate Verification Subsystem.”

In the Certificate Generation Subsystem, transactions that make up the components of the certificate will be created and registered on the blockchain, and the block header that approves these transactions will be broadcasted via IPDC.

In the Certificate Verification Subsystem, the reliability of the transactions that make up the components of the certificate will be verified based on the block header received through IPDC.

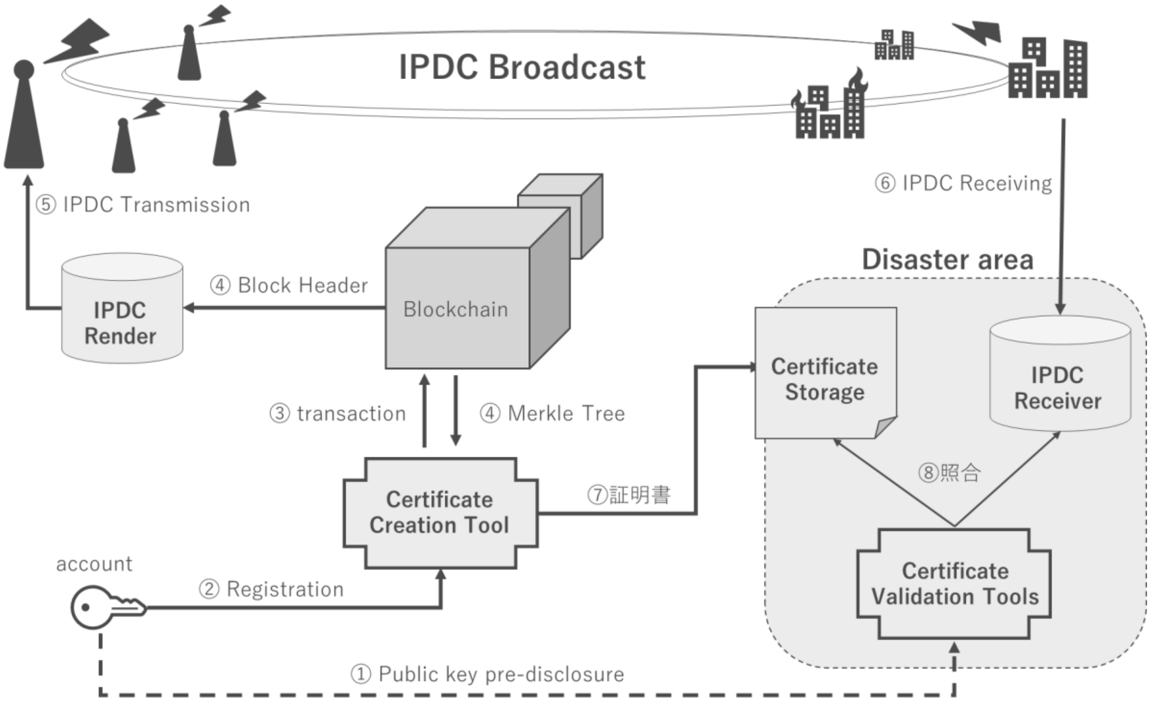


FIGURE: OVERALL CONFIGURATION

In this configuration, disaster areas are assumed as environments without internet access. During large-scale disasters, communication infrastructure used in daily life (such as 4G/5G mobile, free Wi-Fi, and home Wi-Fi) may become unusable due to various reasons. The recovery time after disaster victims lose access to mobile communication can range from several days to weeks (as seen in the case of the Noto Peninsula earthquake in Japan on January 1, 2024). During this period, smartphones such as Android and iPhone may become unusable. Currently, most mobile communication relies on terrestrial stations, wired connections like fiber optic cables, or terrestrial wireless systems, making it vulnerable to large-scale disasters. In this situation, if public authentication using the Cardano blockchain becomes possible, it could help prevent crimes.

In disaster areas, people who have evacuated because their homes have been damaged take refuge in evacuation centers. When a disaster lasts for a long time, fatigue from communal living can lead to distrust, increased crime, and the spread of infectious diseases. In such chaotic situations, blockchain demonstrates its power in terms of reliability. Trust eases the mind of evacuees, prevents crime, and gives evacuees peace of mind and safety. This is important as it also concerns human lives.

Overall flow

As shown in the diagram, at the current stage, the ”certificate” issued through signature registration on the Cardano blockchain is crucial, and authentication processing can be carried out as usual if the internet environment is available. However, our assumption is that even when the internet environment is lost after a large-scale disaster, it should still be possible to conduct ”public authentication” and, consequently, ”protect the lives of disaster victims.” In other words, even if a large-scale disaster occurs, the Cardano blockchain can be utilized to enable authentication under more severe conditions, thus achieving social contributions.

・Assumption 1: Public Key is Announced in Advance (1)
It is assumed that the public key has been announced in advance.

・Register the Certificate by Signing with the Private Key (2)(3)(4)
Information signed with the private key is registered on the Cardano blockchain. The block header and Merkle tree are retrieved from the generated transaction.

・Issue the Certificate (7) (2)
The certificate is issued and handed over to the registrant.

・Broadcast & Receive via IPDC Waves (5)(6) (3)
Block header information is continuously broadcasted via waves and received using a dedicated device.

・Verification at Disaster Base (8)
The certificate is verified and checked based on (1), (2), and (3).

Basic explanation 1: Difference between cryptography and digital signatures

In the blockchain, we mainly utilize the mechanism of electronic signatures. The private key is strictly managed by the sender and is used for generating the signature. On the other hand, the public key is accessible to anyone and is used for verifying the signature. This mechanism does not encrypt the data itself.

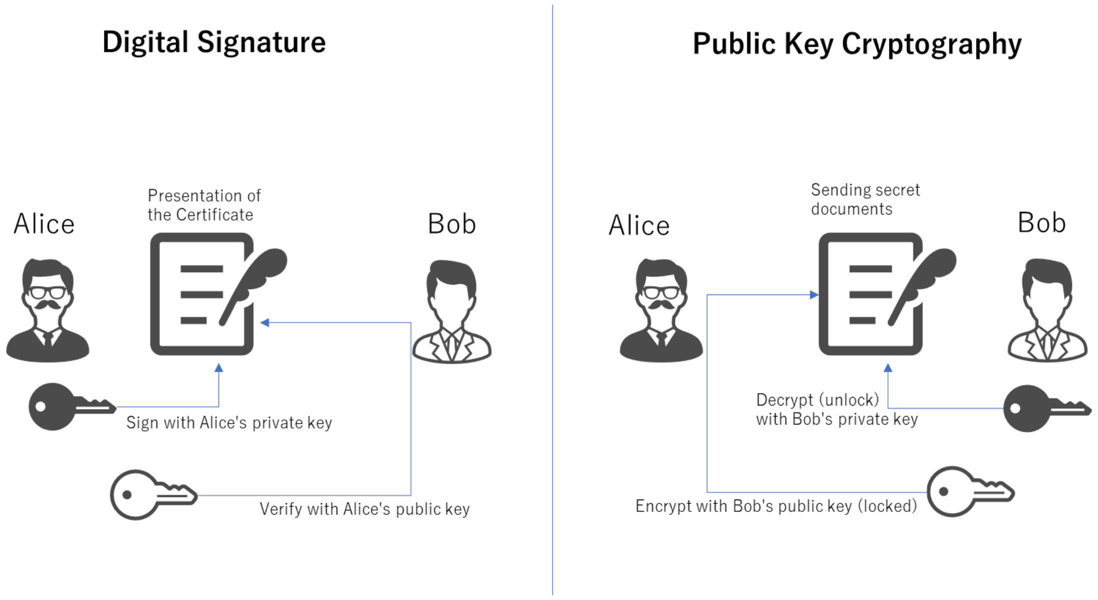
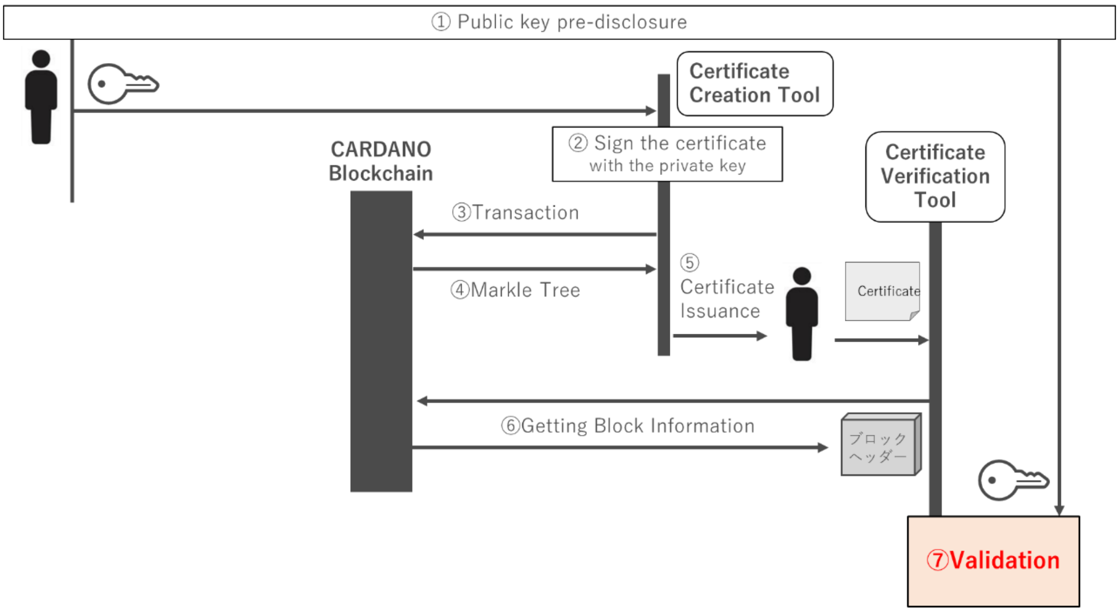

Digital Signature
Alice
Presentation of the Certificate
Bob
Sign with Alice's private key
Verify with Alice's public key
Public Key Cryptography
Alice
Sending secret documents
Bob
Decrypt (unlock) with Bob's private key
Encrypt with Bob's public key (locked)

FIGURE: ENCRYPTION AND SIGNATURES

1.2 Overall processing flow

Normal processing (Internet connection available)

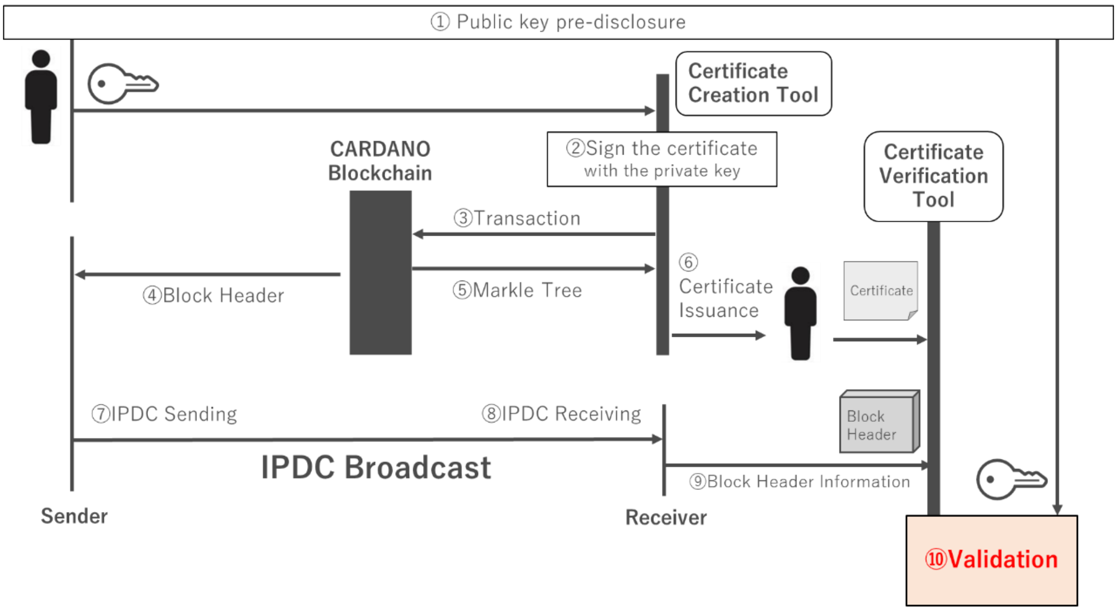
Using a certificate creation tool with a unique account, a certificate is issued via the Cardano blockchain. Even in disaster areas, if the internet environment remains available, the block information can be retrieved from the Cardano blockchain for verification.


① Public key pre-disclosure
Certificate Creation Tool
CARDANO Blockchain
② Sign the certificate with the private key
Certificate Verification Tool
③Transaction
④Markle Tree
⑤ Certificate Issuance
Certificate
⑥Getting Block Information
ブロック ヘッダー
⑦Validation

① Public Key Pre-Announcement
② Register Certificate by Signing with the Private Key
③ Transaction (CARDANO)
④ Merkle Tree Structure
⑤ Issuance by the Certifier
⑥ Block Information Retrieval (CARDANO)
⑦ Validation

Processing with IPDC Broadcast

Using a certificate creation tool with a unique account, a certificate is issued via the Cardano blockchain. In disaster areas where there is no internet access, it is not possible to access the Cardano blockchain. Therefore, after receiving the IPDC broadcast, the block information is retrieved for verification.



① Pre-announcement of Public Key
② Sign the Certificate with the Private Key and Register
③ Transaction (CARDANO)
④ Retrieve Block Header
⑤ Construct Merkle Tree
⑥ Issuance by the Certifier
⑦ IPDC Sender
⑧ IPDC Receiver
⑨ Retrieve Block Header Information
⑩ Validation

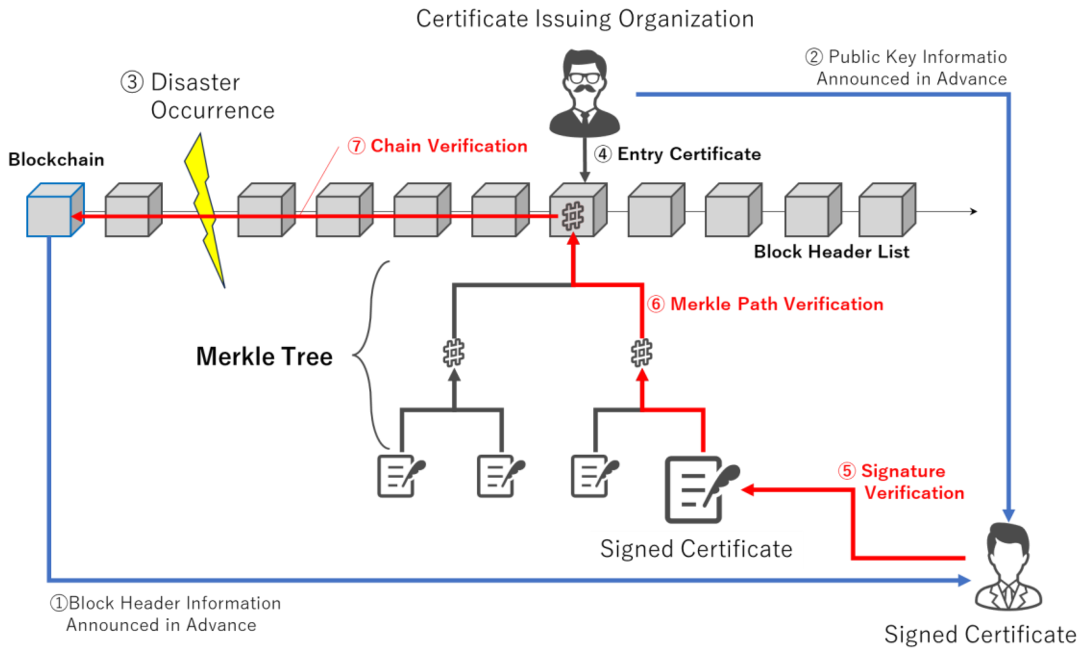
1.3 Certificate Issuance Subsystem

After the transaction is signed by the account of the organization issuing the certificate, it is recorded on the blockchain. When recorded on the blockchain, the Merkle tree information can be calculated from the recorded block height and other transactions recorded in the same block.

These transaction details, Merkle tree information, and block height information are collectively used as a certificate, which is assigned to personnel or equipment that may need identity verification or contract status confirmation during disasters.

Additionally, to verify the existence of the transaction information, it is assumed that block headers generated by the blockchain at regular intervals will be continuously broadcast via IPDC.

Basic explanation: Certificate Issuance Steps



① First, the verifier obtains in advance the block header information, which is well-known and difficult to tamper with on the network, as "information serving as the basis for reliability verification."

② At the same time, the verifier also obtains the public key of the "certificate issuing organization," which will be needed for verification after a disaster occurs.

③ Block headers from the blockchain are broadcasted via IPDC to receivers throughout the broadcast area every time they are generated. Even after a disaster, the robustness of the broadcast facilities, including resistance to earthquakes and wind pressure, ensures that block headers can continue to be distributed via IPDC each time they are generated.

④ After the disaster occurs, the certificate issuing organization creates the necessary certificates for disaster response and recovery support at any given time, recording them as transactions

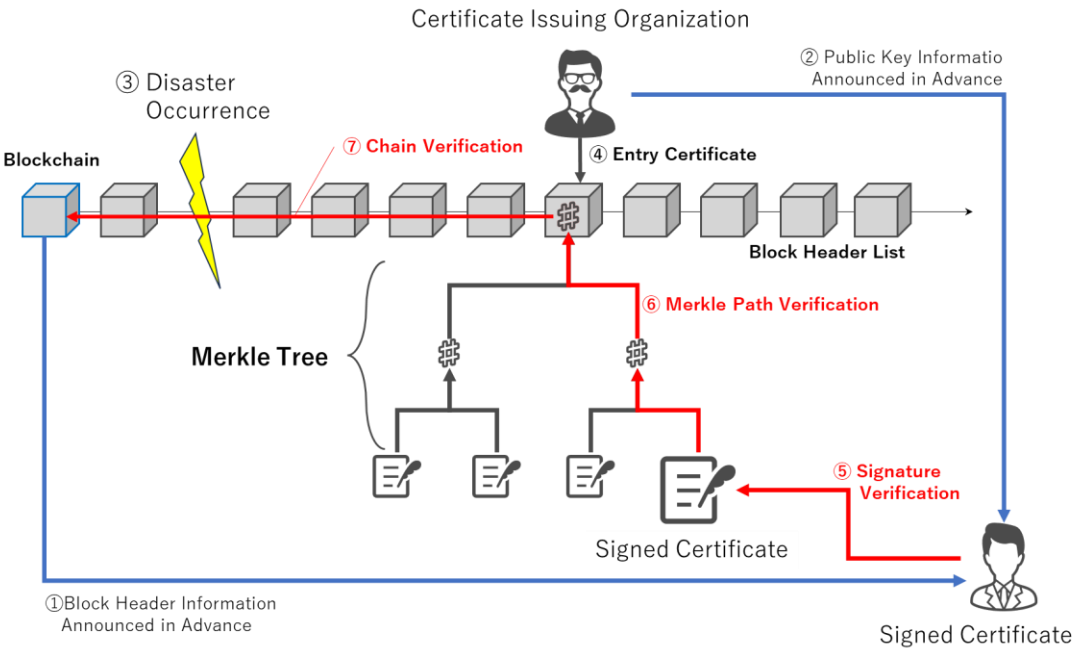
1.4 Certificate Validation Subsystem

The reliability is verified by matching the block header information received via IPDC with the certificate brought into the disaster area. First, the IPDC receiver constantly receives block headers from the broadcast and regularly monitors the integrity of the blockchain. If a rollback occurs, the receiver updates the saved block header list by receiving the broadcast again. The verifier extracts the block header information from the IPDC receiver and first verifies that it traces back to a known block height.

The verifier extracts the block header information from the IPDC receiver and verifies that it traces back to a known block height. Then, they receive the certificate from the presenter and verify the signature on the certificate content. Furthermore, the verifier calculates the transaction root information using the transaction information recorded in the certificate’s critical parts and the Merkle tree information attached to the certificate.

If this transaction root information matches the transaction root value within the block header at the specified block height, it proves that the transaction is recorded on the blockchain.

Basic Description: Certificate Verification Steps



⑤ If a staff member providing support on-site needs to prove their identity, they present the "verifiable certificate" to the local verifier. The local verifier first checks whether the presented certificate has been tampered with by verifying it using the public key of the certificate-issuing organization.

⑥ Next, after confirming the integrity of the certificate, the verifier calculates the transaction root using the Merkle path and verifies whether it matches the transaction root of a specific block header received from the IPDC receiver. If these values match, it means that the block header could not have been generated without the transaction being present, confirming that the transaction meets the various constraints that occur when the block is generated.

⑦ If the transaction root matches the block header of block "n," the information contained within that block header proves the existence of the "previous block hash" for block n-1. By sequentially verifying this backwards and confirming that the block header obtained in step

(1) exists, the chain information sent via broadcast is verified to be an integrity-maintained chain, starting from the ”block header information that serves as the basis of reliability.”

2 The Cardano blockchain and its relationship with the airwaves（1-b）

In the previous year (2023), verification with broadcast signals was conducted in the Symbol blockchain environment. The Symbol blockchain uses an account-based model, which differs from the UTXO model used by the Cardano blockchain. Therefore, this project has significant implications for next-generation cross-chain technology by conducting prototype verification using the Cardano blockchain.

2.1 Differences between Carudano (UTXO) and Symbol (account-based)

2.1.1 UTXO（Unspent Transaction Output）Model

The UTXO model is a system that is particularly useful for tracking and managing transactions, and it stands for ”Unspent Transaction Output.” Specifically, it refers to the balance of tokens or coins that have not yet been spent after a transaction has occurred on the blockchain.

・Input and Output of Transactions:

Each transaction has one or more ”inputs” and ”outputs.” The input refers to the UTXO generated by a previous transaction, and the output is the newly created UTXO.

・Consumption and Generation of Transactions:

UTXOs are consumed as inputs in transactions and are newly generated as outputs. For example, when Person A sends 1 ADA to Person B, the UTXO used for that transfer is consumed, and a new UTXO is generated in Person B’s wallet.

・Transparency and Ease of Tracking:
Each transaction can be clearly traced back to the UTXO it is based on, making the flow of transactions highly transparent.

・Scalability:
In the UTXO model, there is no need to track account balances; it simply manages unspent transaction outputs, making the verification of specific transactions straightforward.

・Security:
Since UTXOs can only be used once, the model prevents double-spending (the issue of using the same asset more than once). This enhances the consistency and security of the blockchain.

・Parallel Processing Capability:
The UTXO model allows for easier parallel processing of transactions, which contributes to improved scalability and performance of the system.

2.1.2 ExUTXO（Extended Unspent Transaction Output）Model

Cardano adopts an improved version of the Bitcoin UTXO model called the **ExUTXO (Extended Unspent Transaction Output)** model. This results in the following features:

・Integration with Smart Contracts:
In the EUTXO model, transaction logic can be integrated with smart contracts, and it is combined with Cardano's smart contract functionality (Plutus). This allows for more complex logic and conditional transactions.

・Deterministic Transactions:
In EUTXO, it is predetermined which UTXO each transaction will consume, making the transaction results predictable and easier to maintain the overall integrity of the blockchain.

・Low Cost and High Efficiency:
The EUTXO model is designed to reduce unnecessary gas fees and improve transaction processing efficiency. As a result, it provides a cost-effective system for both developers and users.

2.1.3 Account-based Model

The account-based model directly manages the balance of each user’s account (wallet). Ethereum, among others, adopts this model. It has the following three characteristics:

・**Balance Management**:
Balances are recorded for each account, and transactions cause these balances to increase or decrease.

・**Simple Design**:
The balance of each account is recorded as is for every transaction.

・**Compatibility with Smart Contracts**:
It is well-suited for executing smart contracts and modifying their states, making it highly compatible with smart contract operations.

2.1.4 Differences between the UTXO model and the account-based model

The difference between the UTXO (Unspent Transaction Output) model and the Account-Based model primarily lies in the structure of transactions and the way account balances are managed. Below is a detailed explanation of the characteristics of each model and their differences.

Differences in basic structure

<UTXO Model>

Each transaction uses ”Unspent Transaction Outputs (UTXO)” as inputs and generates new outputs. With each transaction, the previously unspent funds (UTXO) are used in the next transaction. Bitcoin and Cardano adopt this model.

<Account-Based Model>

Each account has a balance, and when a transaction occurs, the balance of that account is directly increased or decreased. With each transaction, the new account state is updated, and instead of using UTXO as the input/output of the transaction, the account balance is modified. Ethereum and Symbol adopt this model.

How to process transactions

<UTXO Model>

Each transaction uses multiple UTXOs as ”inputs” and generates multiple ”outputs.” All UTXOs can only be used once, and once used, they are consumed. Payments can be made by combining multiple UTXOs, and if change is needed, a new UTXO is generated. Since each UTXO is associated with a fixed amount of cryptocurrency, transactions are highly transparent, and the risk of double-spending is reduced.

<Account-Based Model>

In a transaction, the balance of the sender's account decreases, and the balance is added to the receiver's account. Since the balance is directly updated on the blockchain, there is no need to track transaction outputs like in UTXO. Transactions are simple and only require consideration of the amount sent and the fees.

Parallel Processing

<UTXO Model>

In the UTXO model, transactions are independent, making parallel processing easier. Since multiple UTXOs can be processed simultaneously, it offers excellent scalability. Each transaction specifies which UTXO it consumes, and this does not affect other transactions.

<Account-Based Model>

In the account-based model, since the balance of each account is continuously updated, parallel processing can be difficult. If multiple transactions attempt to process the same account simultaneously, conflicts may occur.

Gas charges and fees

<UTXO Model>

Each transaction is calculated based on which UTXO it uses. The gas fee is usually determined by the complexity of the transaction and the number of UTXOs used.

<Account-Based Model>

Transaction fees may either be fixed per transaction or vary depending on the transaction’s content (such as data size or computational load). When there are complex operations, such as executing smart contracts, the fees tend to increase.

Transparency and Security

<UTXO Model>

Since all UTXOs are tracked as inputs and outputs of transactions, the flow of transactions is highly transparent. The entire transaction history is clearly visible, preventing double-spending.

<Account-Based Model>

Since the account balance is directly modified, the transaction history appears simple at first glance, but it does not have the same level of transparency as the UTXO model when it comes to tracking balances. In the account-based model, additional measures may be necessary to enhance security.

Compatibility with smart contracts

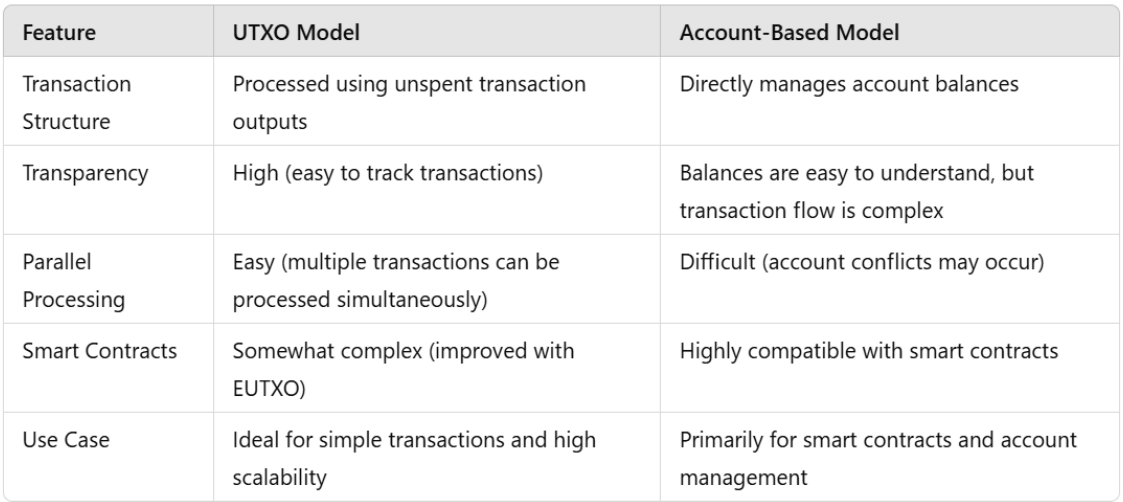
<UTXO Model>

Integration with smart contracts is somewhat complex. Since UTXOs are consumed once used, it becomes difficult to manage the state of smart contracts. However, Cardano adopts the ”EUTXO” model, an improved version of the UTXO model, which allows for integration with smart contracts.

<Account-Based Model>

Integration with smart contracts is somewhat complex. Since UTXOs are consumed once used, it becomes difficult to manage the state of smart contracts. However, Cardano adopts the ”EUTXO” model, an improved version of the UTXO model, which allows for integration with smart contracts.

Comparison Table



Feature	UTXO Model	Account-Based Model
Transaction Structure	Processed using unspent transaction outputs	Directly manages account balances
Transparency	High (easy to track transactions)	Balances are easy to understand, but transaction flow is complex
Parallel Processing	Easy (multiple transactions can be processed simultaneously)	Difficult (account conflicts may occur)
Smart Contracts	Somewhat complex (improved with EUTXO)	Highly compatible with smart contracts
Use Case	Ideal for simple transactions and high scalability	Primarily for smart contracts and account management

2.2 Integration of IPDC and Blockchain Technology and the Significance of This Demonstration

2.2.1 Method for Reliability Verification by Integrating IPDC and Blockchain

IPDC (IP Data Cast) is a technology that enables efficient data transfer in a unidirectional transmission environment via broadcasting, allowing information to be transmitted over a wide area even in situations where internet connectivity is difficult. This technology is an effective means of ensuring that critical information is delivered reliably to recipients during widespread communication failures or localized cloud outages, such as in disaster scenarios.

However, due to the unidirectional nature of the transmission, there is a challenge in how to verify the reliability of the received data. In particular, with the increasing prevalence of data tampering and forgery using artificial intelligence in modern times, if the reliability of the received information is not guaranteed, decisions and economic activities based on that information may involve significant risks.

One important factor in verifying reliability is the presence of a "trust anchor." In traditional Public Key Infrastructure (PKI), third-party certification authorities (CAs) function as trust anchors, ensuring the reliability of information. In environments where bidirectional communication over the network is possible, queries can be made to the certification authority to verify trustworthiness, reducing the likelihood of problems. However, in unidirectional transmission environments such as IPDC, it is difficult for the receiver to scrutinize the contents of the information, and additional methods are required to ensure reliability.

2.2.2 Method for New Reliability Verification Using Blockchain Technology

One promising method to solve the challenges of reliability verification in a unidirectional transmission environment is the use of blockchain technology. Blockchain has a chain structure where each block is linked by a hash, making it highly reliable and difficult to tamper with. Furthermore, by verifying that the received data is recorded in a specific block, blockchain ensures that the data has not been tampered with.

Additionally, since broadcasting is publicly available as a social infrastructure, it is continuously monitored by viewers, making it practically impossible to deliver different information only to specific users. Therefore, using blockchain as a system to prove the existence of data can be a safer and more reliable approach. By leveraging smart contracts, it is also possible to control the recording of only electronically signed data that meets certain conditions, enabling more flexible reliability verification.

However, there are several challenges when using blockchain technology in a unidirectional transmission environment. Typically, blockchain requires bidirectional communication to maintain data consistency between nodes and needs to constantly verify the synchronization of the entire network. Therefore, when using IPDC to transmit information to terminals outside the network, the challenge lies in how to verify the reliability of the received information. In other words, while IPDC ensures the reliability of the transmission path through broadcasting, the key issue is whether the reliability of the received information can also be guaranteed. If this can be achieved, a platform that ensures trustworthiness can be established.

2.2.3 Conditions necessary for realization

To implement this reliability verification model, several important conditions must be met.

・The trust anchor and the information of the transaction issuer must remain unchanged.

In unidirectional transmission, there is no way to detect changes in the source information. Therefore, to ensure the reliability of the received information, it is essential that critical information, such as the trust anchor and the public key of the transaction issuer, has not been tampered with. Additionally, to prevent data tampering, strict security measures, such as regular rotation of private keys, are necessary.

・All information required for verification must be provided to the receiver.

In a unidirectional communication environment, all data required for verification must be provided to the receiver. For example, while transaction data and trust anchor information are connected through Merkle trees and block headers, all of this information must be provided. To meet these conditions, it is essential to select and implement appropriate blockchain technology.

2.2.4 Previous Cases Using the Symbol Blockchain and Their Outcomes

In our previous research, we confirmed that these issues can be resolved by utilizing the Symbol blockchain. First, in blockchain systems, the genesis block or widely-known block hashes can be used as trust anchors, and the risk of these being tampered with is low. Additionally, the security of the transaction issuer’s key information can be maintained by regularly rotating private keys. Furthermore, with Symbol's flexible multisig (multi-signature) functionality, the configuration of signatories can be easily modified, enabling secure operation.

Symbol also implements a decentralized API (REST API), allowing verification of blockchain information while checking the synchronization status of multiple nodes without the need to maintain your own node. This mechanism is particularly effective for broadcasters who need to provide information during disasters, and it is also advantageous for building a multi-chain broadcast system.

2.2.5 Challenges in the Cardano Blockchain and Considerations for Disaster Operations

Although multi-signature is available on the Cardano blockchain, changing the key configuration is difficult, and it seems unlikely that operations can be performed where the public key remains fixed while rotating the private key. Additionally, the public APIs of Cardano (e.g., blockfrost) do not provide sufficient information to verify blocks and transactions, which may limit the reliability verification in disaster situations.

For this reason, a new approach called "Trusted Web" has been proposed in Japan. Unlike Web3, which verifies all information, the Trusted Web adopts a system where trusted companies ensure part of the reliability, aiming for more flexible and efficient data circulation. Based on this concept, it is possible to build a highly reliable information provision and verification environment by leveraging the public auditability of broadcasting.

About blockfrost (reference)

An API platform that functions as infrastructure for accessing data on the Cardano blockchain. Blockfrost enables developers to easily access Cardano blockchain data and functions, supporting the rapid development of applications and services. The following are its features:

·Easy API Access

It provides a mechanism to easily access Cardano's blockchain data through a REST API. This allows programs to access block information, transactions, addresses, assets (such as tokens), and stake pool information.

·Cardano Blockchain Integration

It connects to the Cardano blockchain nodes, allowing users to utilize Cardano's data without managing the nodes themselves. This eliminates the need for developers to build and maintain their own infrastructure, allowing them to focus on application development.

·Transaction submission

Using Blockfrost, it is possible to send transactions directly to the Cardano blockchain. This simplifies the development of services such as wallet apps and payment apps that require interactions on the blockchain.

·Multiple Network Support

It supports both the Cardano mainnet and testnet, allowing developers to smoothly test and deploy in both production and development environments.

·IPFS Support

In addition to Cardano, it also supports the decentralized file system IPFS (InterPlanetary File System). This allows developers to build DApps (decentralized applications) and services using decentralized file storage.

·Scalability and Stability

It provides scalable infrastructure, enabling reliable access even for large-scale applications. Developers can efficiently handle a large number of requests, with a focus on the reliability and response speed of the API.

About the Trusted Web (reference)

The Trusted Web is a new internet concept and technical framework promoted by the Japanese government, certain companies, and research institutions. This initiative is designed to enhance the reliability and transparency of the digital society, ensuring the authenticity of information, preventing tampering, and protecting privacy. Unlike traditional centralized web systems, it aims to create a decentralized system for sharing trustworthy information, with the goal of building an internet that everyone can use with confidence.

・Ensuring reliability

The Trusted Web aims to ensure the reliability of information exchanged in the digital space. It addresses issues such as false information, fake news, and data tampering. By verifying the source and authenticity of each piece of data, it provides a system that ensures the proper circulation of trustworthy information.

・Utilization of Decentralized Technology

It is based on technologies such as blockchain and Distributed Ledger Technology (DLT). These technologies prevent data tampering and enable transparent and secure information sharing among multiple parties. By eliminating reliance on a central authority, trust is autonomously maintained.

・Data Privacy Protection

Protecting user privacy is also an important element of the Trusted Web. It ensures the exchange of trustworthy data while preventing unnecessary leakage of personal data and its misuse by third parties.

・Transparent and tamper-proof

In the Trusted Web, information transparency and the ability to track change histories are possible. By using decentralized databases like blockchain, it records who updated the information, when, and how, making data tampering impossible.

・Increased trust in online transactions

By utilizing this technology, it provides an environment where all parties involved in online transactions, contracts, and other exchanges can trust the information. As a result, it allows for safer transactions in B2B and B2C commercial activities.

2.3 Utilization of the Cardano Blockchain on IPDC and Future Prospects

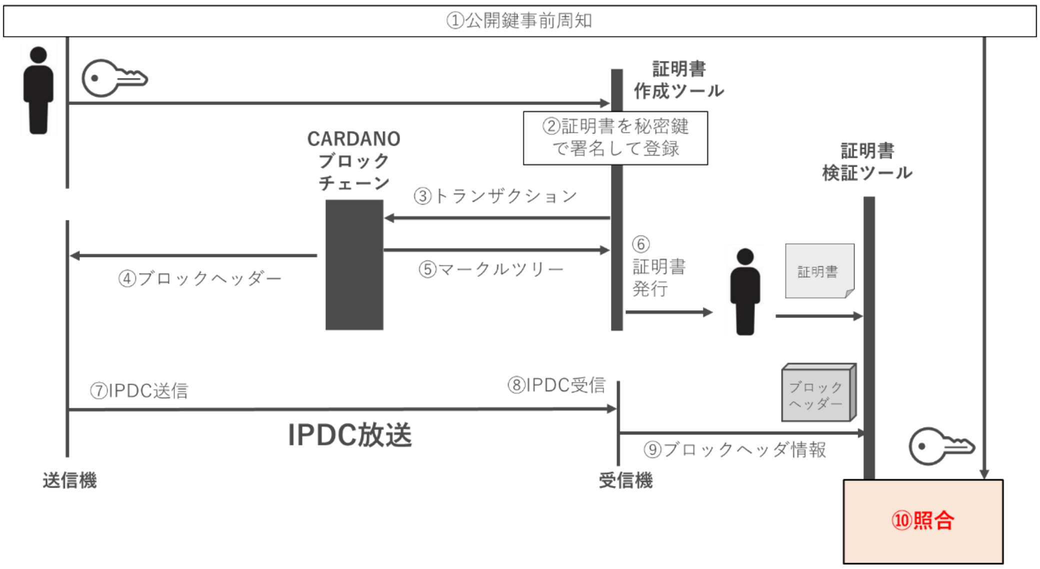
To implement a reliability verification system that integrates IPDC technology with the Cardano blockchain, the following elements are essential:

2.3.1 Ensuring the reliability of the certificate issuer through broadcasting.

The broadcaster guarantees reliability by authenticating the document issuer. Additionally, a system should be established that excludes transactions that do not meet authentication requirements from broadcasting, thereby preventing the transmission of fraudulent information.

2.3.2 Broadcasting supplements information necessary for verification

Provide the necessary information for verification (such as block headers and transaction verification data) to the receiver via broadcasting. This allows the receiver to trust the broadcast content and perform reliability verification smoothly. It is essential that the transmission and reception of block information via IPDC broadcast, as illustrated in steps ⑦ and ⑧, maintain the highest level of reliability.



With such a system, it is believed that a new mechanism can be established that ensures the reliability of received information while enabling efficient data distribution, even in environments or situations where the internet is unavailable.

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

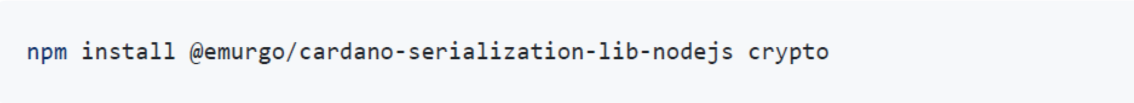
：It is used for Cardano key generation and address generation.

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.



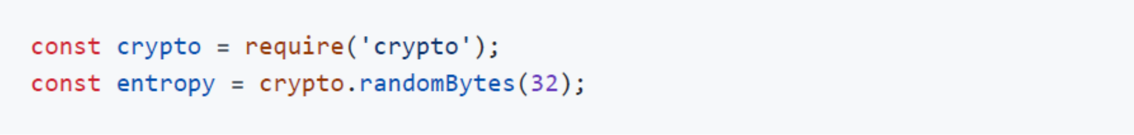
```
npm install @emurgo/cardano-serialization-lib-nodejs crypto
```

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

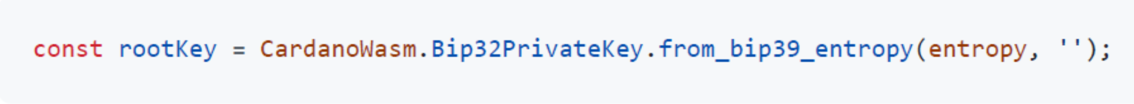
Generate 32 bytes of random entropy. This entropy will be used later as a seed for key generation.



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

Private Key Generation

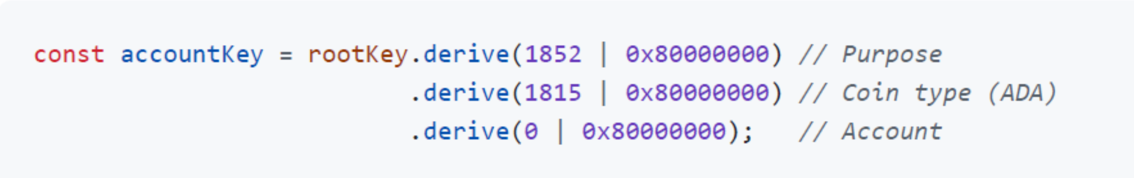
Generate a BIP32 root private key using the generated entropy.



```
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.



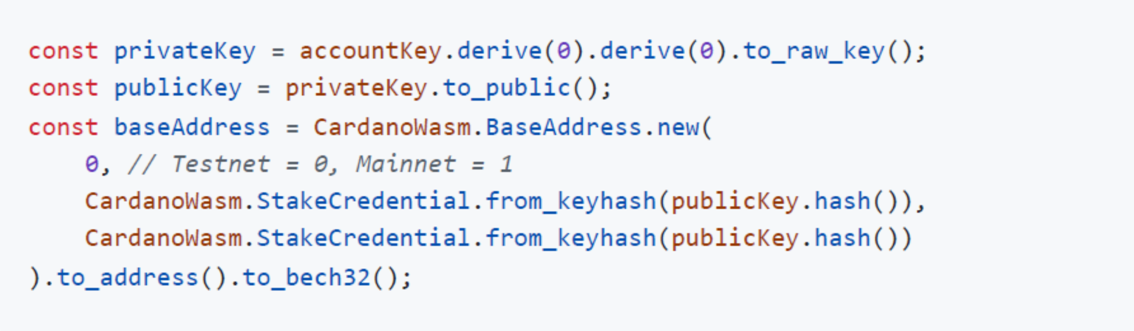
```
const accountKey = rootKey.derive(1852 | 0x80000000) // Purpose
                          .derive(1815 | 0x80000000) // Coin type (ADA)
                          .derive(0 | 0x80000000);   // Account
```

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

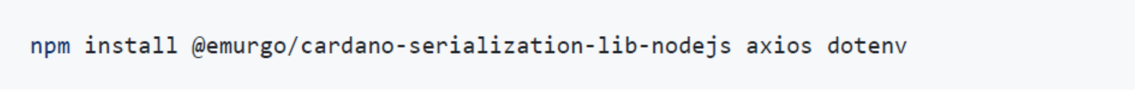
: Used to make HTTP requests and retrieve data from the Blockfrost API.

Dotenv

: Used to read environment variables.

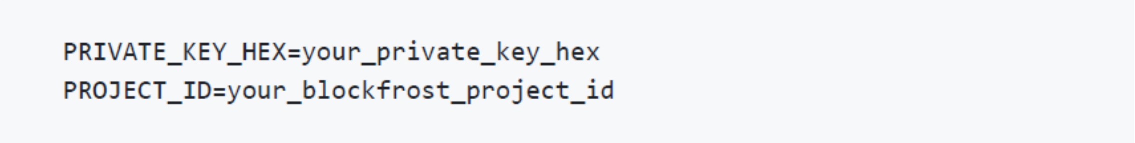
<environment>

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:

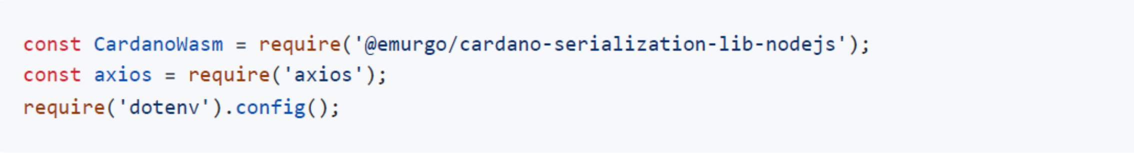


```
PRIVATE_KEY_HEX=your_private_key_hex
PROJECT_ID=your_blockfrost_project_id
```

3.2.3 Basic Research Content

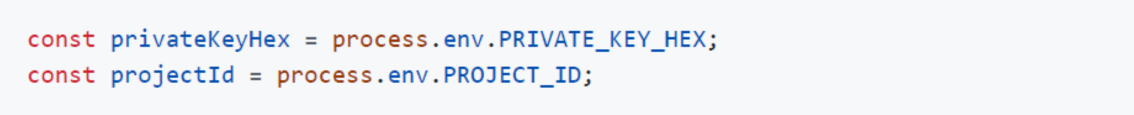
Importing Libraries

Import the necessary libraries.



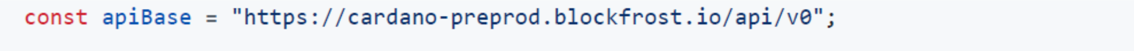
```
const CardanoWasm = require('@emurgo/cardano-serialization-lib-nodejs');
const axios = require('axios');
require('dotenv').config();
```

Loading Environment Variables



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

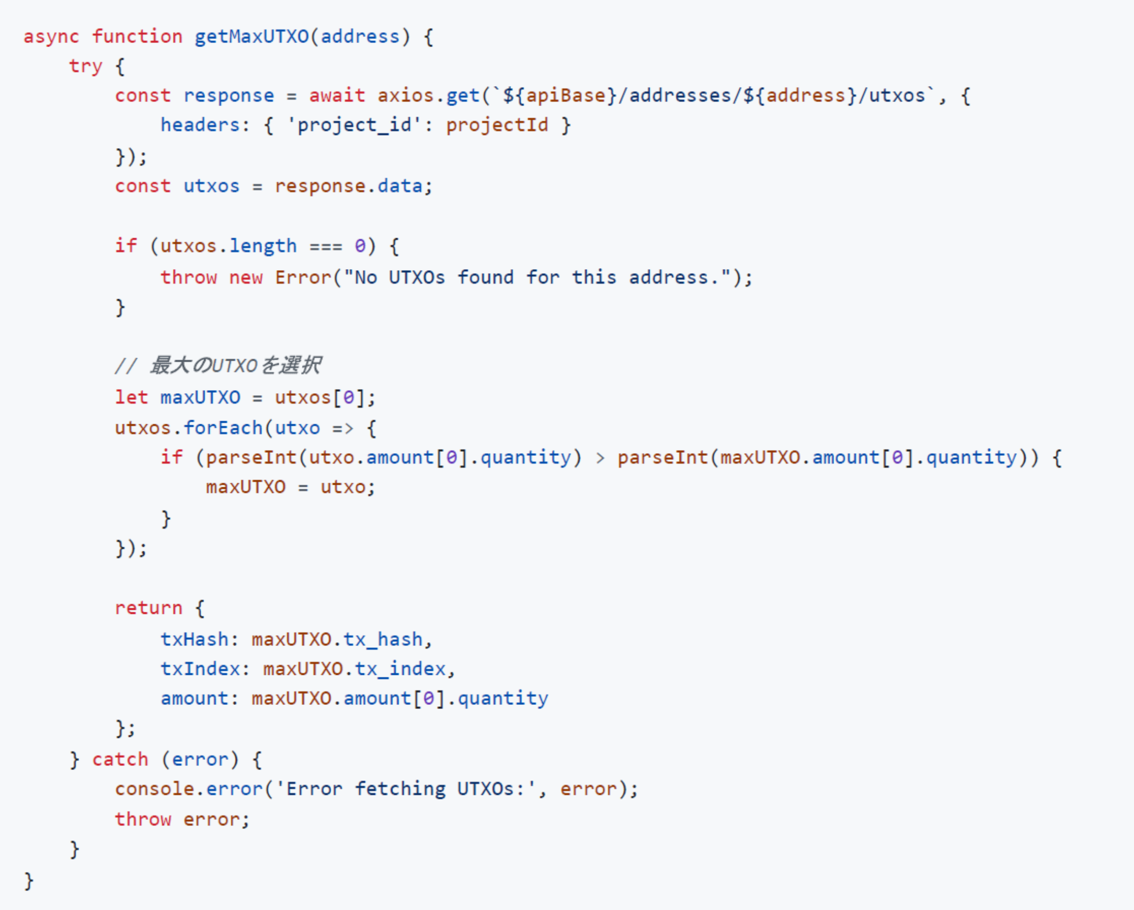
Retrieve the private key and Blockfrost project ID from the environment variables.



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

Get UTXO

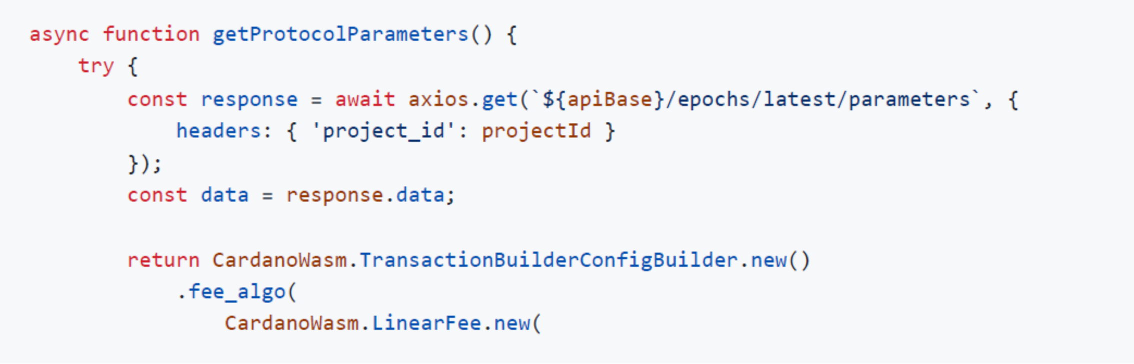
After getting the UTXOs for the specified address, select the largest UTXO and return it. Construct and sign the transaction, then output the signed transaction in Hex format.



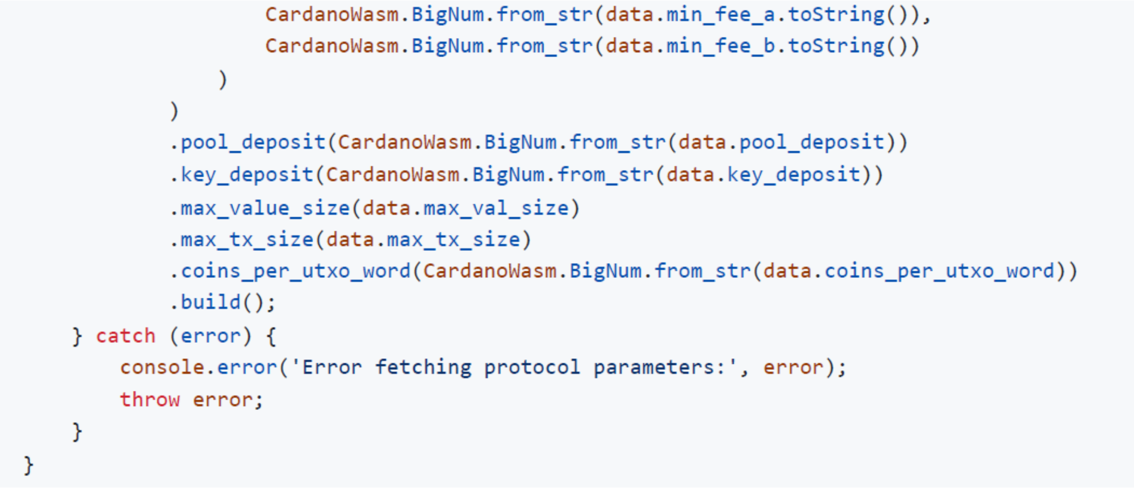
```
async function getMaxUTXO(address) {
    try {
        const response = await axios.get(`${apiBase}/addresses/${address}/utxos`, {
            headers: { 'project_id': projectId }
        });
        const utxos = response.data;

        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

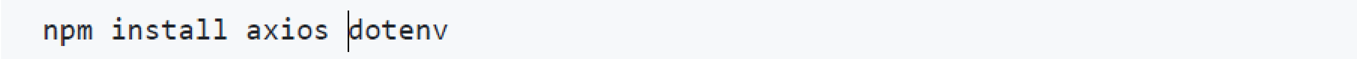
: Used to make HTTP requests and retrieve data from the Blockfrost API.

dotenv

: Used to read environment variables.

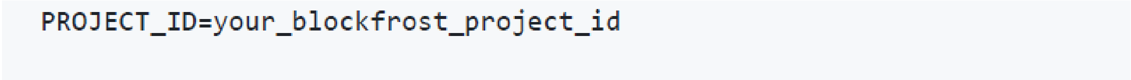
<environment>

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:

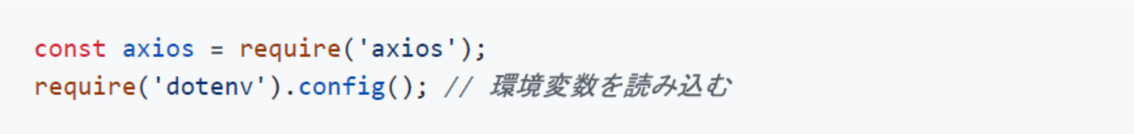


```
PROJECT_ID=your_blockfrost_project_id
```

3.3.3 Basic Research Content

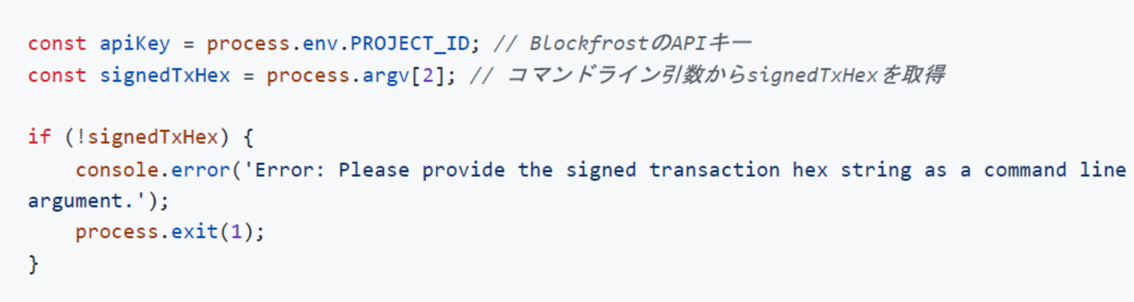
Importing Libraries

Import the necessary libraries.



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

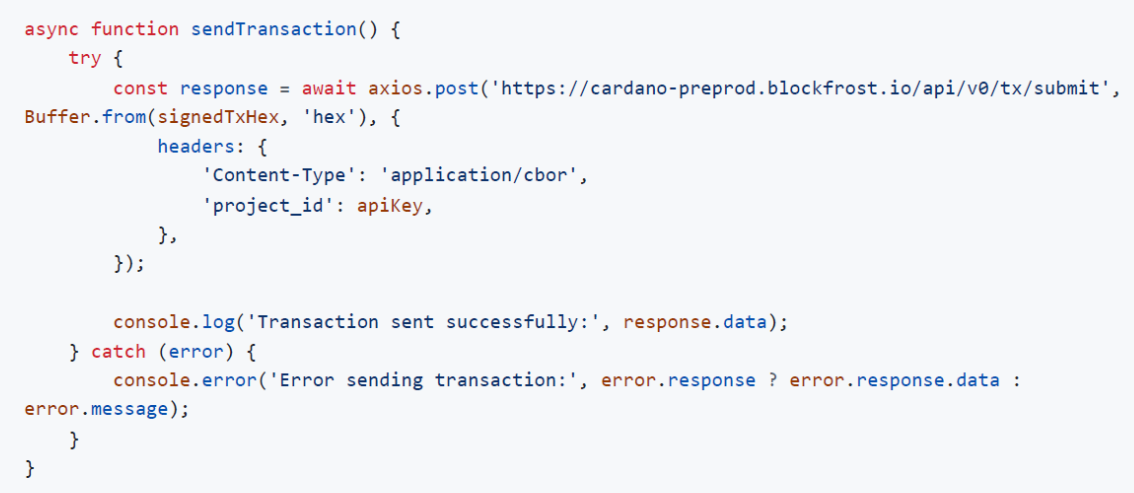
Getting environment variables and command line arguments



```
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＞

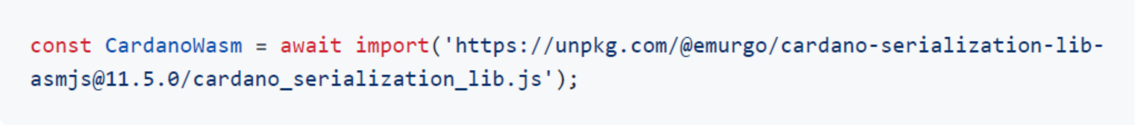
JavaScript code that works in a browser environment.

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

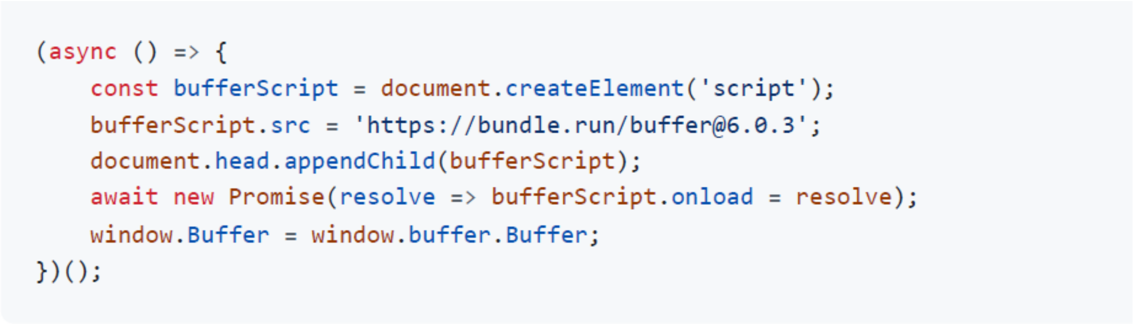
Importing Libraries

Import the necessary libraries.



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

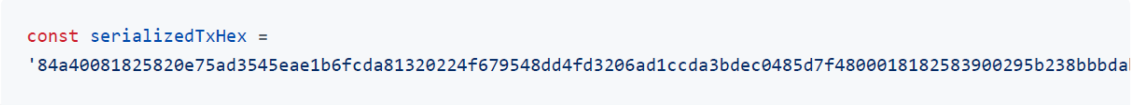
Load the cardano-serialization-lib-asmjs library into the browser environment. This library is specialized for Cardano transaction processing.



```
(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;
})();
```

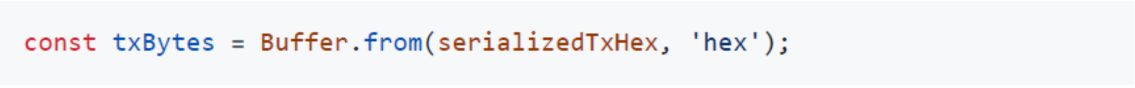
Load the Buffer polyfill into the browser to make the Buffer object available. This is necessary when handling serialized transactions.

Processing Serialized Transactions



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

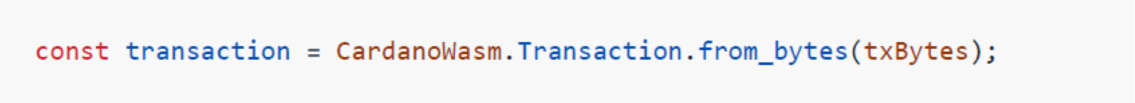
Define the serialized transaction as a Hex format string. This string represents the complete transaction data.



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

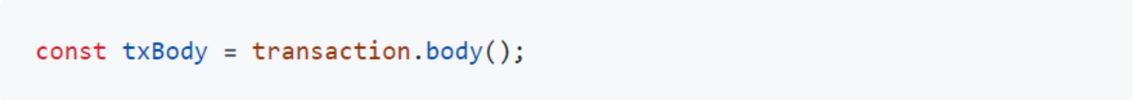
Convert the Hex string to a byte array. This allows the transaction data to be handled in binary format.

Deserializing a transaction and getting the transaction ID



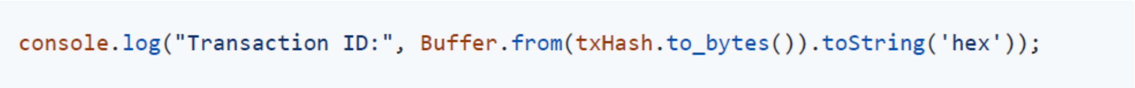
```
const transaction = CardanoWasm.Transaction.from_bytes(txBytes);
```

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.

4 Block information when sending IPDC（2-b）

As part of the research on block headers and Merkle paths on the Cardano platform, we conducted investigations into block validation, rollback, and transaction root analysis.

4.1 Research on Block Confirmation

4.1.1 Research Objective

Monitoring the Integrity of the Cardano Blockchain in Real-Time and Investigating Inconsistency Detection Processes
We investigate methods for ensuring the security and reliability of the blockchain, particularly through the practice of retrieving block information and detecting inconsistencies using the Blockfrost API.

<**Practical Real-Time Monitoring of Block Information**>
Since blockchain operations change in real-time, periodic monitoring is crucial. In this research, we explore a method for real-time monitoring, where the latest block information is retrieved every 15 seconds, and the continuity of blocks and the integrity of block hashes are checked. This technology provides a foundation for always keeping track of the blockchain’s operational status and responding promptly.

<**Validation of Inconsistency Detection Mechanisms**>
To maintain the integrity of the blockchain, it is essential to quickly detect inconsistencies when they occur. This research validates mechanisms for detecting inconsistencies when block hashes do not match or when gaps occur in block heights. By doing so, we verify practical methods for ensuring the security and reliability of the blockchain.

4.1.2 Libraries used and environment

<Libraries used>

fetch API

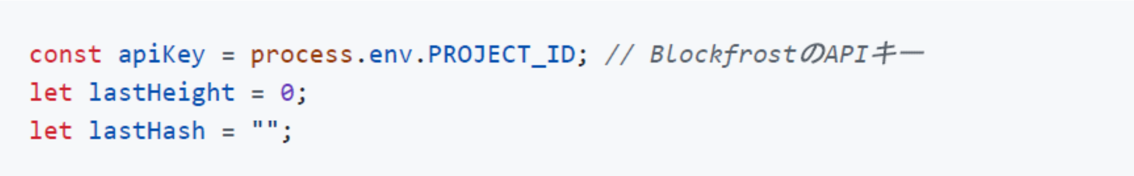
: It is used to make HTTP requests and retrieve data from external APIs using the native JavaScript API.

<environment>

It works in both browser and Node.js environments.
To use the Blockfrost API, you need to obtain an API key in advance.

4.1.3 Basic Research Content

Initial Setup



```
const apiKey = process.env.PROJECT_ID; // BlockfrostのAPIキー
let lastHeight = 0;
let lastHash = "";
```

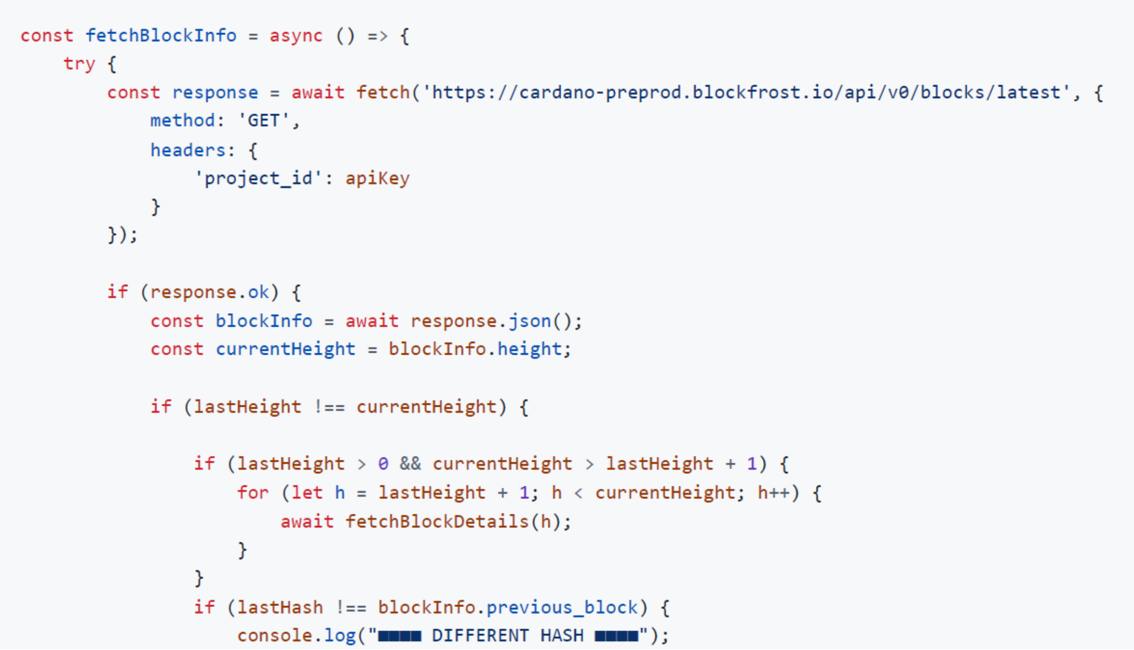
API KEY

：Define the key to access the Blockfrost API. A key for the preprod network is specified.

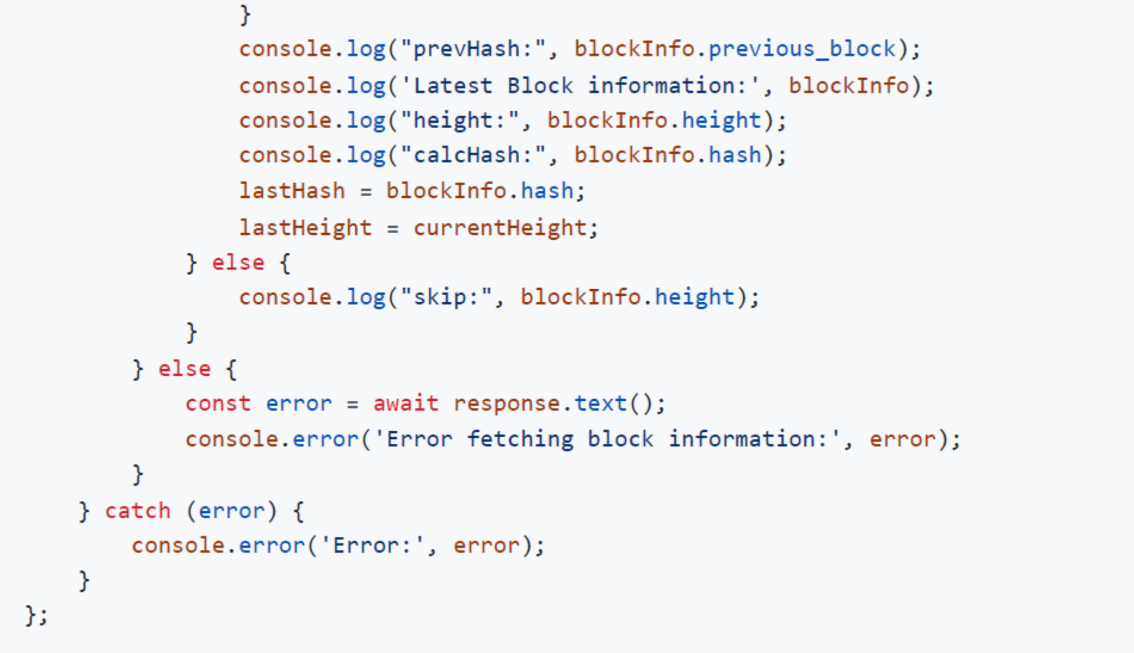
State Management Variables

: lastHeight and lastHash hold the height and hash of the previously retrieved block.

Obtaining and Monitoring Block Information



```
const fetchBlockInfo = async () => {
    try {
        const response = await fetch('https://cardano-preprod.blockfrost.io/api/v0/blocks/latest', {
            method: 'GET',
            headers: {
                'project_id': apiKey
            }
        });

        if (response.ok) {
            const blockInfo = await response.json();
            const currentHeight = blockInfo.height;

            if (lastHeight !== currentHeight) {

                if (lastHeight > 0 && currentHeight > lastHeight + 1) {
                    for (let h = lastHeight + 1; h < currentHeight; h++) {
                        await fetchBlockDetails(h);
                    }
                }
                if (lastHash !== blockInfo.previous_block) {
                    console.log("■■■■ DIFFERENT HASH ■■■■");
```



```
                }
                console.log("prevHash:", blockInfo.previous_block);
                console.log('Latest Block information:', blockInfo);
                console.log("height:", blockInfo.height);
                console.log("calcHash:", blockInfo.hash);
                lastHash = blockInfo.hash;
                lastHeight = currentHeight;
            } else {
                console.log("skip:", blockInfo.height);
            }
        } else {
            const error = await response.text();
            console.error('Error fetching block information:', error);
        }
    } catch (error) {
        console.error('Error:', error);
    }
};
```

fetchBlockInfo Function

: Retrieves the latest block information and compares it with the previously retrieved block.

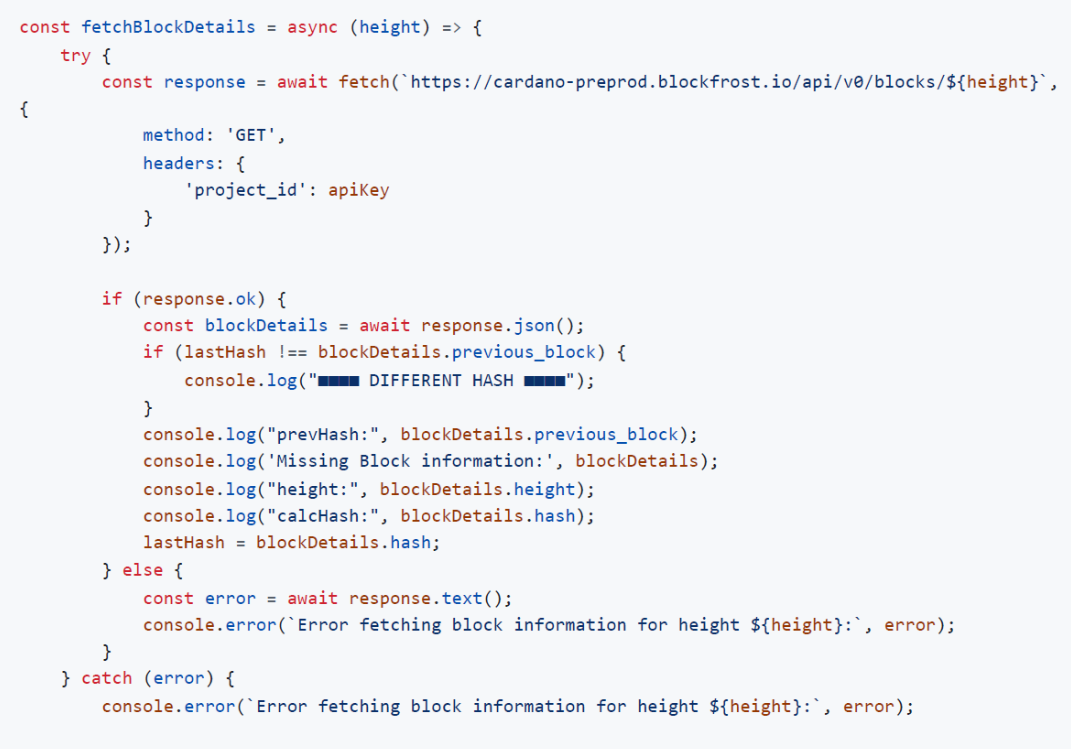
Inconsistency Check

: Logs any inconsistencies if the block height or the previous block’s hash does not match.

Gap Check

: If there is a gap between the previously retrieved block height and the latest block height, the blocks in between are retrieved individually to check the details.

Getting Blocks with Gaps

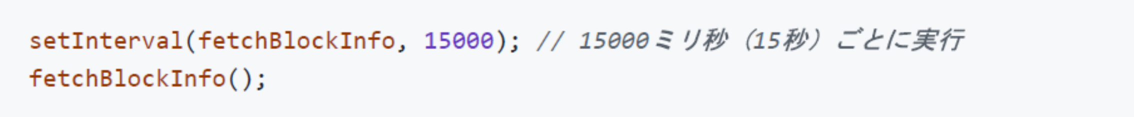


```
const fetchBlockDetails = async (height) => {
    try {
        const response = await fetch(`https://cardano-preprod.blockfrost.io/api/v0/blocks/${height}`,
{
            method: 'GET',
            headers: {
                'project_id': apiKey
            }
        });

        if (response.ok) {
            const blockDetails = await response.json();
            if (lastHash !== blockDetails.previous_block) {
                console.log("■■■■ DIFFERENT HASH ■■■■");
            }
            console.log("prevHash:", blockDetails.previous_block);
            console.log('Missing Block information:', blockDetails);
            console.log("height:", blockDetails.height);
            console.log("calcHash:", blockDetails.hash);
            lastHash = blockDetails.hash;
        } else {
            const error = await response.text();
            console.error(`Error fetching block information for height ${height}:`, error);
        }
    } catch (error) {
        console.error(`Error fetching block information for height ${height}:`, error);
```

fetchBlockDetails 関数

：Retrieve block information corresponding to the specified block height and compare it with the previous hash.

Periodic Block Information Getting



```
setInterval(fetchBlockInfo, 15000); // 15000ミリ秒（15秒）ごとに実行
fetchBlockInfo();
```

setInterval

：Execute the fetchBlockInfo function every 15 seconds to periodically monitor the latest block information.

4.1.4 Research Results

This code retrieves the latest block information on the Cardano preprod network and monitors block continuity and consistency. Every 15 seconds, the latest block information is retrieved, and the following details are checked:

- **Block Height**: Compare with the height of the previously retrieved block to check for gaps.
- **Block Hash**: Compare with the previous block hash to check for inconsistencies.

4.2 Research on Rollback

4.2.1 Research Objective

Verifying a Practical Process for Maintaining the Integrity of the Cardano Blockchain and Building a Reliable Monitoring System.
By introducing hash verification and reconciliation functions using past data, this enables swift troubleshooting in the event of inconsistencies.

<Advanced Integrity Maintenance of the Cardano Blockchain>
The integrity of the Cardano blockchain is ensured by the continuity of blocks and the consistency of hashes. The purpose of this research is to build advanced monitoring functions that detect and address inconsistencies, such as hash mismatches or missing blocks on the blockchain. In particular, by introducing the storage of past block hashes and reconciliation (integrity recovery) functions, the continuity of the blockchain is maintained, further strengthening its reliability.

4.2.2 Libraries used and environment

There are no libraries to be used.

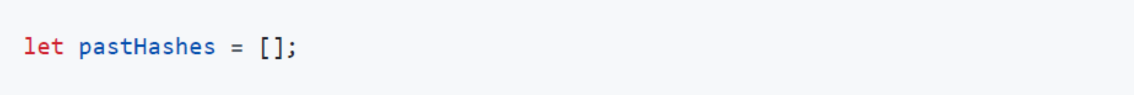
4.2.3 Basic Research Content

Functionality Overview

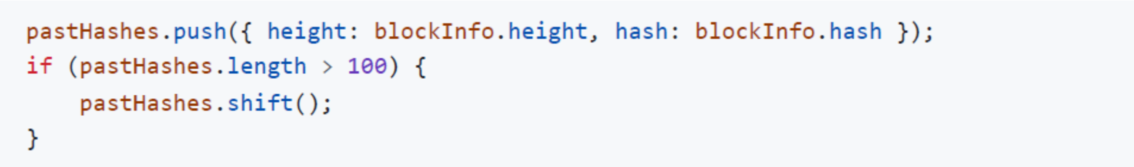
The following new features have been added:

- **Hash History Storage**: Save the hashes of previously retrieved blocks and use them during integrity checks.
- **Block Reconciliation**: A function that, when a hash mismatch is detected on the blockchain, goes back through previous blocks to find the matching block.

Hash History Storage



```
let pastHashes = [];
```

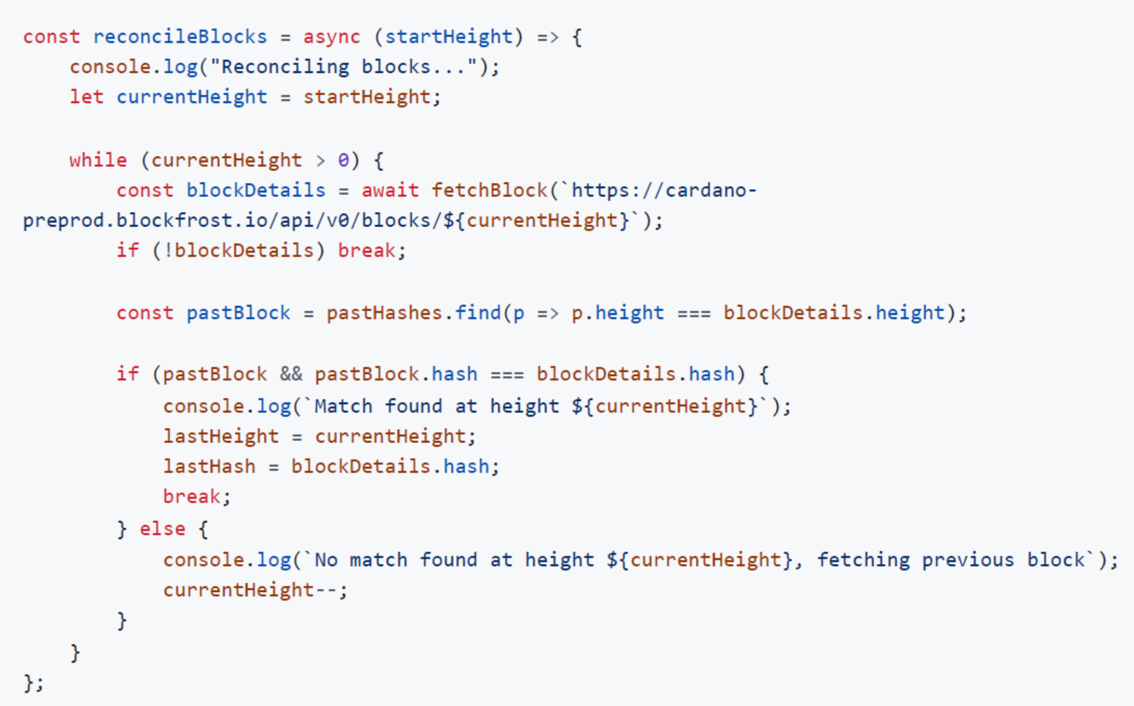


```
pastHashes.push({ height: blockInfo.height, hash: blockInfo.hash });
if (pastHashes.length > 100) {
    pastHashes.shift();
}
```

Managing Past Hashes
: Introduce a pastHashes list to store the hashes and heights of previously retrieved blocks. This list holds up to 100 hash entries, and older entries are deleted when this limit is exceeded. This provides a foundation for comparing the hashes of past blocks with the current block.

Adding Hashes and Managing the List
: Add the information of newly retrieved blocks to pastHashes. If the list exceeds 100 entries, the oldest entry is removed to manage memory usage.

Reconciliation in Case of Hash Mismatch



```
const reconcileBlocks = async (startHeight) => {
    console.log("Reconciling blocks...");
    let currentHeight = startHeight;

    while (currentHeight > 0) {
        const blockDetails = await fetchBlock(`https://cardano-
preprod.blockfrost.io/api/v0/blocks/${currentHeight}`);
        if (!blockDetails) break;

        const pastBlock = pastHashes.find(p => p.height === blockDetails.height);

        if (pastBlock && pastBlock.hash === blockDetails.hash) {
            console.log(`Match found at height ${currentHeight}`);
            lastHeight = currentHeight;
            lastHash = blockDetails.hash;
            break;
        } else {
            console.log(`No match found at height ${currentHeight}, fetching previous block`);
            currentHeight--;
        }
    }
};
```

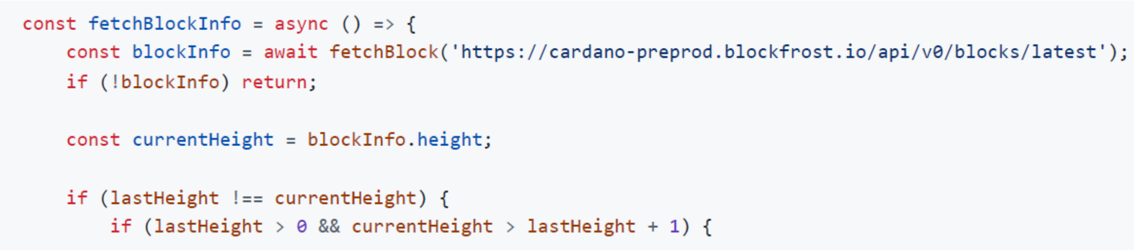
Reconciliation Function:

: This function is called when a hash mismatch is detected. It retrieves block information by going back from the current block height and compares it with the data stored in pastHashes. If a block with a matching hash is found, that block is used as the basis for restoring consistency.

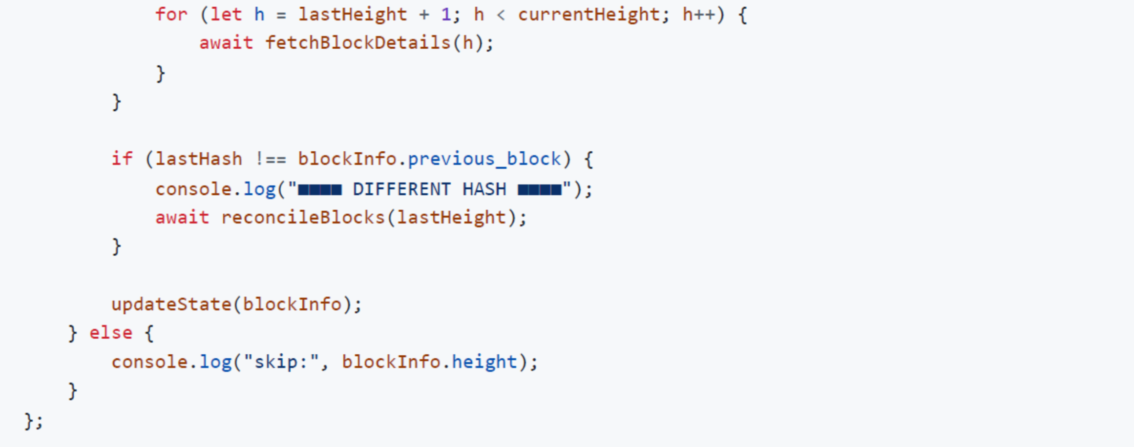
Utilizing Past Block Information:

: By comparing with the past block information stored in the pastHashes list, block integrity can be efficiently verified, and the point of inconsistency can be identified.

Improved Block Monitoring Functionality



```
const fetchBlockInfo = async () => {
    const blockInfo = await fetchBlock('https://cardano-preprod.blockfrost.io/api/v0/blocks/latest');
    if (!blockInfo) return;

    const currentHeight = blockInfo.height;

    if (lastHeight !== currentHeight) {
        if (lastHeight > 0 && currentHeight > lastHeight + 1) {
```



```
            for (let h = lastHeight + 1; h < currentHeight; h++) {
                await fetchBlockDetails(h);
            }
        }

        if (lastHash !== blockInfo.previous_block) {
            console.log("■■■■ DIFFERENT HASH ■■■■");
            await reconcileBlocks(lastHeight);
        }

        updateState(blockInfo);
    } else {
        console.log("skip:", blockInfo.height);
    }
};
```

Enhanced Inconsistency Handling:

: When block hashes do not match, instead of just displaying a warning, the reconcileBlocks function is called to attempt to restore consistency. This process ensures the continuity of the blockchain and improves the system’s reliability.

Benefit from improved code

Improved Reliability:

: When block inconsistencies occur, referencing past block data helps restore appropriate consistency, thereby enhancing the reliability of the blockchain.

Efficient Monitoring:
: By storing past hashes and using them in the reconciliation process, troubleshooting becomes easier when inconsistencies are detected.

4.2.4 Research Results

With this code, the Cardano blockchain monitoring system has gained advanced functionality to maintain block hash integrity and address potential inconsistencies. The addition of storing past block hashes and the reconciliation function further enhances the reliability of the blockchain and strengthens real-time monitoring.

4.3 Research on Transaction Route

4.3.1 Research Objective

The purpose of this research is to understand and apply the construction of a Merkle Tree and the use of the Blake2b hash function to verify the integrity of transaction data on the Cardano blockchain.

<Practical Application of Efficient Data Verification Using Merkle Tree>
The Merkle Tree is widely used as a method for efficiently verifying large datasets. In this research, we will confirm how to construct a Merkle Tree using transaction IDs on the Cardano blockchain, hashed with the Blake2b hash function.

<Application and Understanding of the Blake2b Hash Function>
In blockchain technology, hashing data is essential for ensuring security and integrity. The goal of this research is to understand how to hash transaction IDs using the Blake2b hash function and apply the results to construct a Merkle Tree. Blake2b is known for being a fast and secure hash function, and by applying it in practice, we will confirm its advantages in data processing on the blockchain.

4.3.2 Libraries used and environment

＜Libraries used＞

axios

：Used to make HTTP requests and send transactions to the Blockfrost API.

dotenv

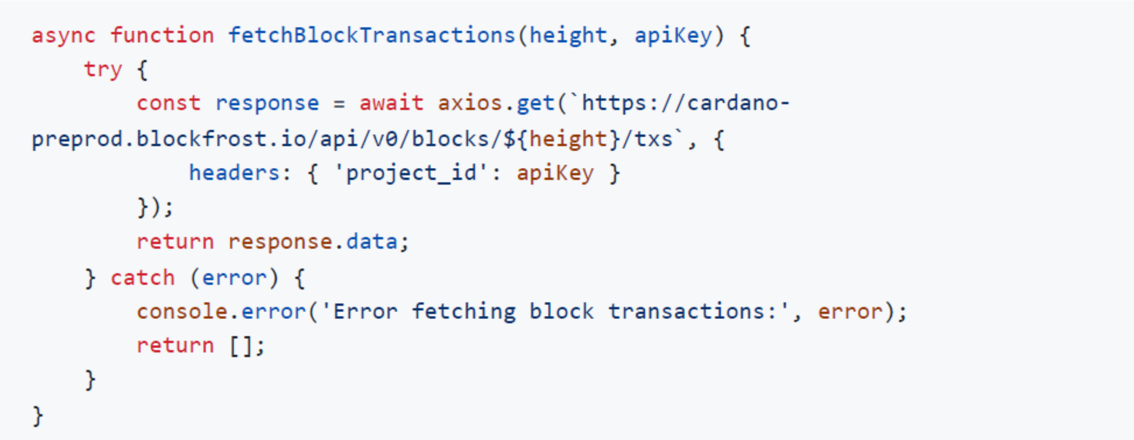
：Used to load environment variables.。

＜environment＞

This is JavaScript code that runs in the Node.js environment.
An API key is required to use the Blockfrost API.
The blakejs library is required to use the Blake2b hash function.

4.3.3 Basic Research Content

Getting Transaction



```
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 [];
    }
}
```

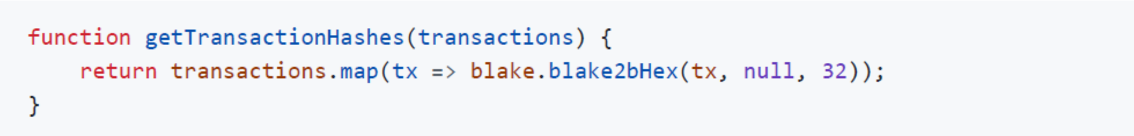
fetchBlockTransactions Function

: Retrieves transaction IDs using the Blockfrost API based on the specified block height. 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.

トランザクションIDのハッシュ計算

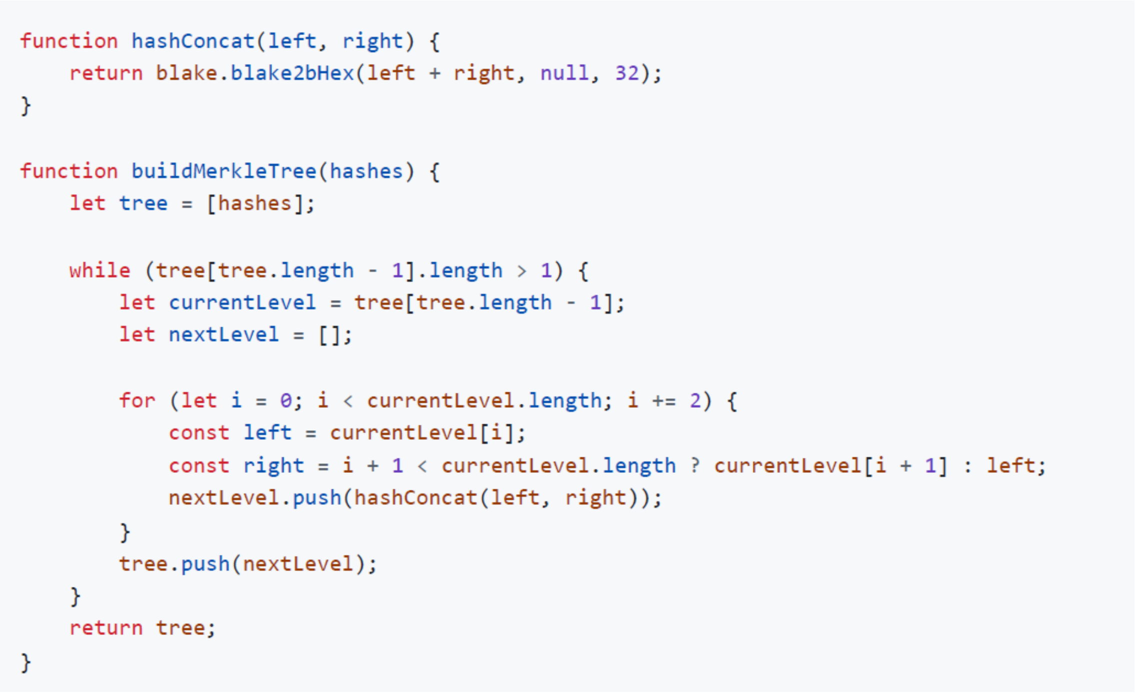


```
function getTransactionHashes(transactions) {
    return transactions.map(tx => blake.blake2bHex(tx, null, 32));
}
```

getTransactionHashes Function

: Hashes the retrieved transaction IDs using the Blake2b hash function, generating 32-byte hash values. This sets the foundation for constructing a Merkle Tree from the transaction IDs.

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;
}
```

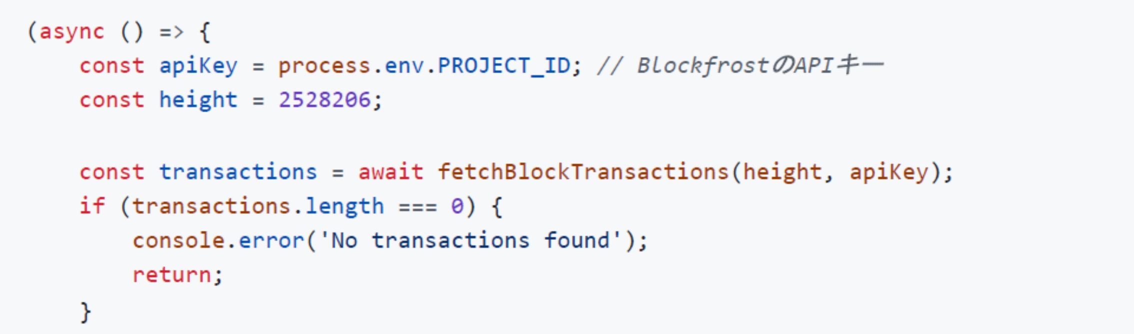
hashConcat Function

: 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.

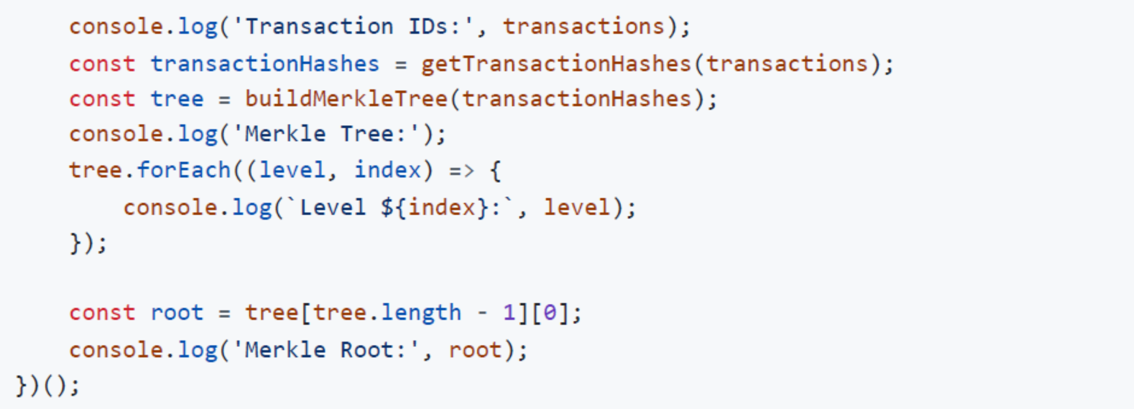
buildMerkleTree Function

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

Getting Transactions and Constructing a Merkle Tree



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

    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);
    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);
})();
```

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.

Constructing the Merkle Tree

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

Displaying the Merkle Root

: Display the root hash (Merkle Root) of the constructed Merkle Tree.

4.3.4 Research Results

This code retrieves transactions from the specified block, hashes the transaction IDs using the Blake2b hash function, and constructs a Merkle Tree. Finally, it calculates the Merkle Root and outputs the result.

Transaction Retrieval

: Transaction IDs are retrieved from the specified block and used for hashing.

Merkle Tree Construction

: Each transaction ID is hashed, and these are used to progressively construct the Merkle Tree.

Merkle Root Calculation

: The hashes at the top level are concatenated, and the final Merkle Root is obtained.

5 Data Structure for IPDC Transmission（2-c）

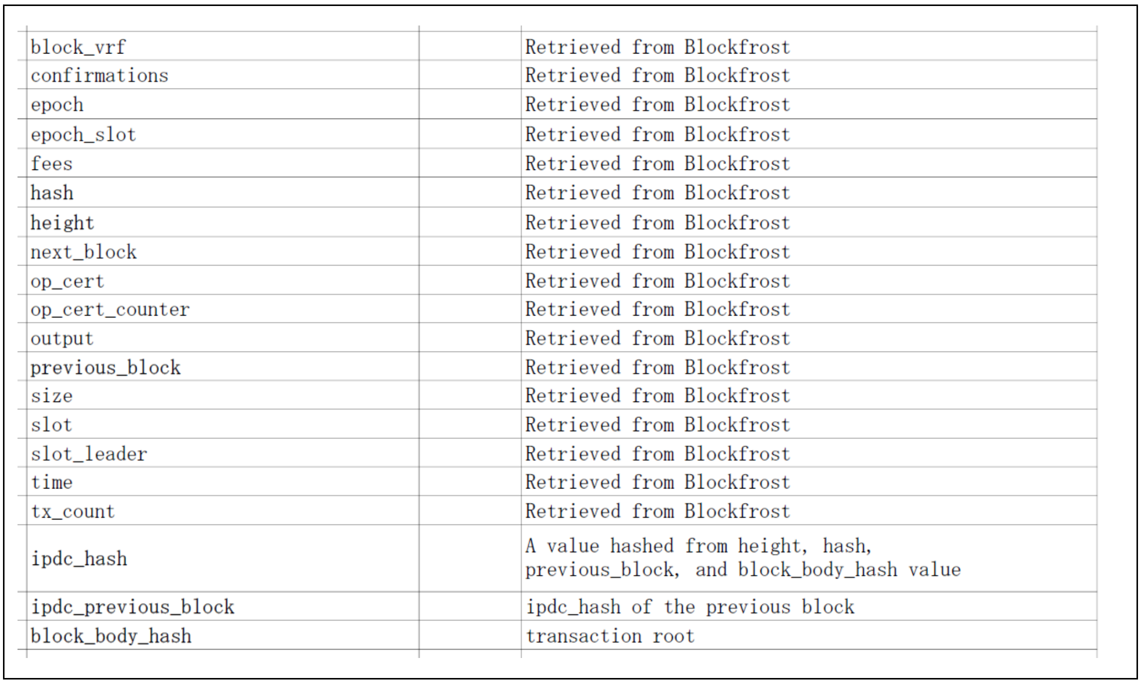
Research on the Data Structure and Entry Encoding Required for IPDC Transmission.

5.1 Data structure and transmission data

5.1.1 Data structure and transmission data

The data to be transmitted via IPDC will consist of block header information as shown below. Each parameter is obtained from the block information retrieved from the Cardano blockchain and structured as custom JSON data. The file name follows the rule head_{block height}.json, ensuring a unique name within the system. In IPDC transmission, it is assumed that if the latest information is available, it will be sent.

Table : Structure of Block Header Information



block_vrf		Retrieved from Blockfrost
confirmations		Retrieved from Blockfrost
epoch		Retrieved from Blockfrost
epoch_slot		Retrieved from Blockfrost
fees		Retrieved from Blockfrost
hash		Retrieved from Blockfrost
height		Retrieved from Blockfrost
next_block		Retrieved from Blockfrost
op_cert		Retrieved from Blockfrost
op_cert_counter		Retrieved from Blockfrost
output		Retrieved from Blockfrost
previous_block		Retrieved from Blockfrost
size		Retrieved from Blockfrost
slot		Retrieved from Blockfrost
slot_leader		Retrieved from Blockfrost
time		Retrieved from Blockfrost
tx_count		Retrieved from Blockfrost
ipdc_hash		A value hashed from height, hash, previous_block, and block_body_hash value
ipdc_previous_block		ipdc_hash of the previous block
block_body_hash		transaction root

5.2 IPDC Transmission

5.2.1 IPDC Transmission Protocol and Transmission Process

The IPDC protocol follows the model below. This time, we will use the yellow parts. The block header JSON information is divided into UDP packets using the Flute protocol, and the split IP packets are encapsulated in ULE (Unidirectional Lightweight Encapsulation). They are then transmitted using the broadcasting protocol (MPEG-2 TS format).

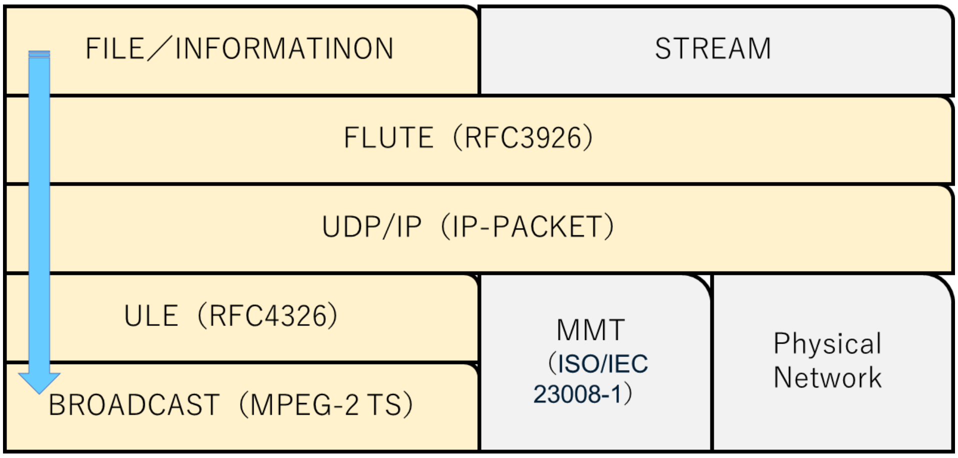


FIGURE: IPDC PROTOCOL STACK (SEND DIRECTION)

<Transmission Process>

JSON information is generated from the block header retrieved from the blockchain. This JSON information is fragmented using FLUTE and encapsulated as IP packets, which are then transmitted through the digital broadcasting system. In Japan, this is realized using terrestrial television, which is the most widely used broadcasting medium.

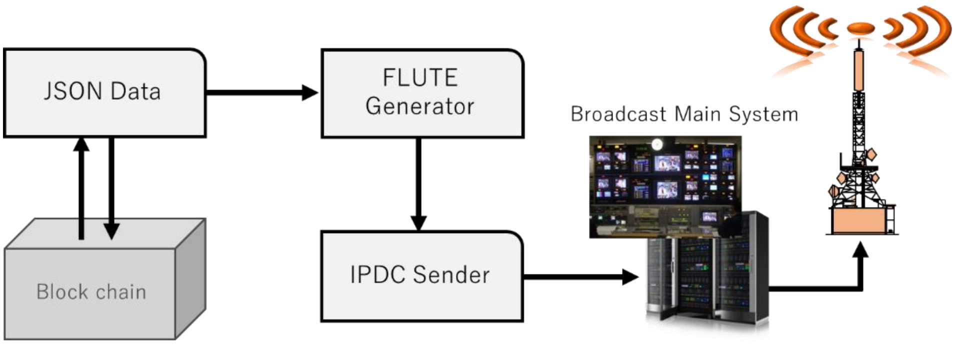


FIGURE : BLOCK DIAGRAM FOR IPDC TRANSMISSION

5.2.2 IPDC Transmission Process Flow

The process flow for IPDC transmission is shown in the following diagram. The IPDC transmission support tool continuously generates JSON information from the latest block information on the Cardano blockchain. The IPDC transmission system retrieves this JSON and broadcasts it as IPDC. ＊In the future, a more secure method will be used to retrieve the information instead of FTP.

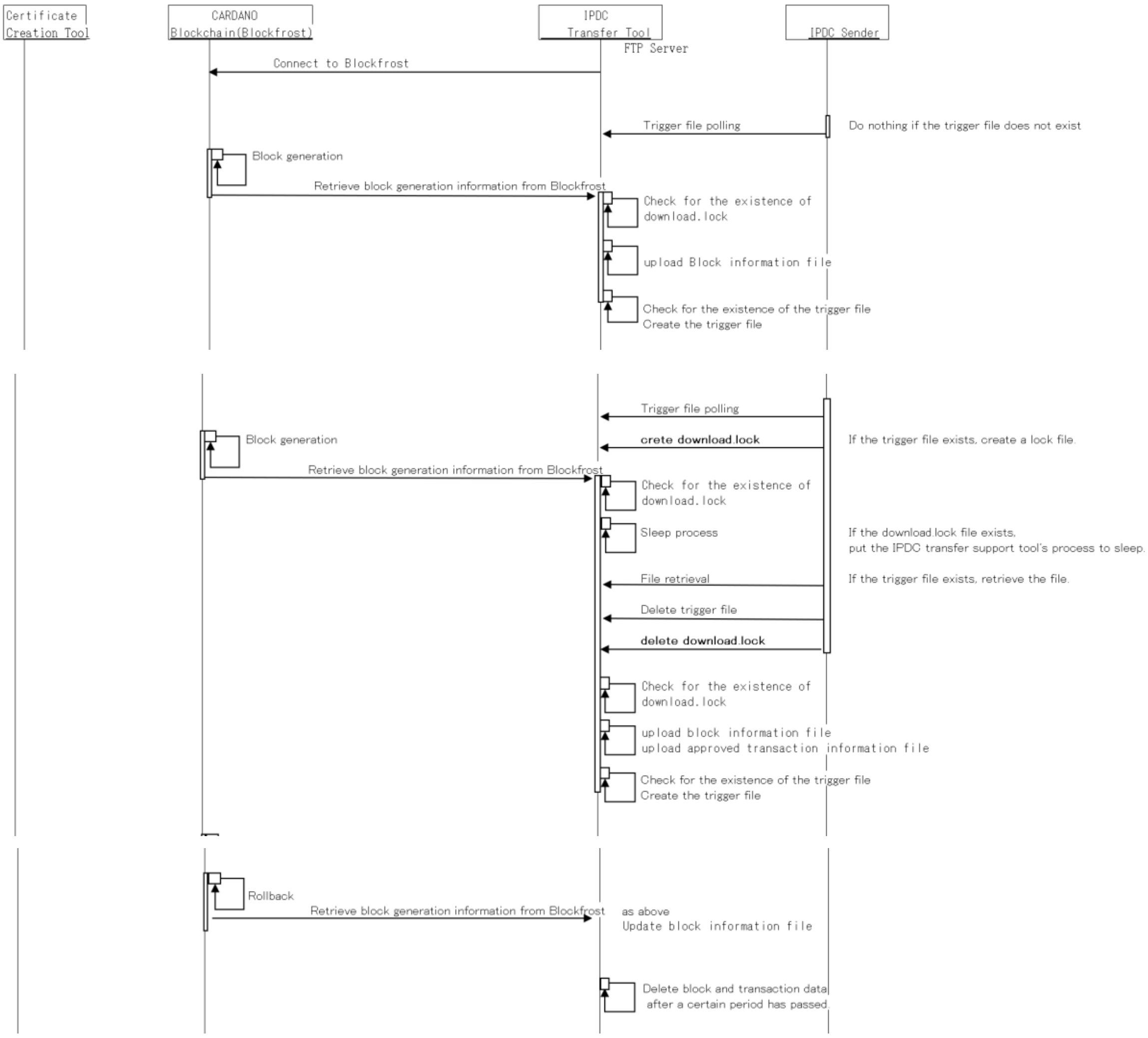

Certificate Creation Tool
CARDANO Blockchain(Blockfrost)
IPDC Transfer Tool
FTP Server
IPDC Sender
Connect to Blockfrost
Trigger file polling
Do nothing if the trigger file does not exist
Block generation
Retrieve block generation information from Blockfrost
Check for the existence of download.lock
upload Block information file
Check for the existence of the trigger file
Create the trigger file
Trigger file polling
Block generation
crete download.lock
If the trigger file exists, create a lock file.
Retrieve block generation information from Blockfrost
Check for the existence of download.lock
Sleep process
If the download.lock file exists, put the IPDC transfer support tool's process to sleep.
File retrieval
If the trigger file exists, retrieve the file.
Delete trigger file
delete download.lock
Check for the existence of download.lock
upload block information file
upload approved transaction information file
Check for the existence of the trigger file
Create the trigger file
Rollback
Retrieve block generation information from Blockfrost
as above
Update block information file
Delete block and transaction data after a certain period has passed

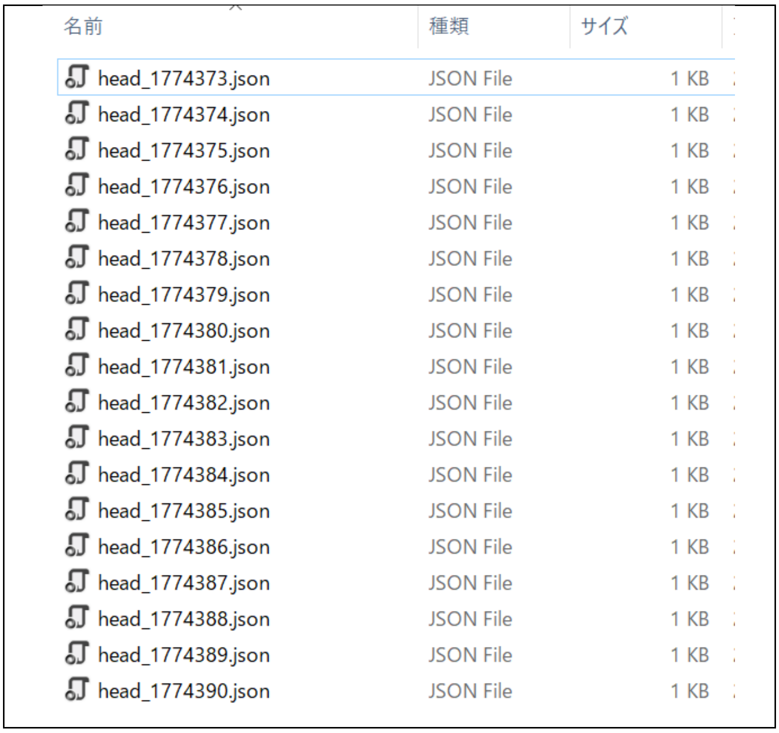
FIGURE : DATA INPUT INTERFACE AND PROCESSING FLOW FOR IPDC TRANSMISSION

6 Data structure after receiving IPDC（3-a)

6.1 Data Structure and Received Data

6.1.1 Received Data and Its Structure

The JSON is obtained via an IPDC dedicated receiver. The received JSON files are managed in a specific folder. In the case where the disaster area lacks internet access and cannot retrieve block information from the Cardano blockchain, the block header information received via IPDC broadcast is used to verify and validate certificates.



名前	種類	サイズ
head_1774373.json	JSON File	1 KB
head_1774374.json	JSON File	1 KB
head_1774375.json	JSON File	1 KB
head_1774376.json	JSON File	1 KB
head_1774377.json	JSON File	1 KB
head_1774378.json	JSON File	1 KB
head_1774379.json	JSON File	1 KB
head_1774380.json	JSON File	1 KB
head_1774381.json	JSON File	1 KB
head_1774382.json	JSON File	1 KB
head_1774383.json	JSON File	1 KB
head_1774384.json	JSON File	1 KB
head_1774385.json	JSON File	1 KB
head_1774386.json	JSON File	1 KB
head_1774387.json	JSON File	1 KB
head_1774388.json	JSON File	1 KB
head_1774389.json	JSON File	1 KB
head_1774390.json	JSON File	1 KB

FIGURE : FOR EXAMPLE OF FILES(BLOCK HEADER JSON)

6.2 IPDC Receiver

6.2.1 IPDC Receiving Protocol and Receiving Processing

The IPDC protocol is the following model. When receiving, the yellow part is used in reverse to the sender. IP packets received using the broadcast protocol (MPEG-2 TS format) are reconstructed by Flute and can be obtained as the original file. This file can be stored at a wide range of receiving points that can receive broadcast radio waves, making it effective.

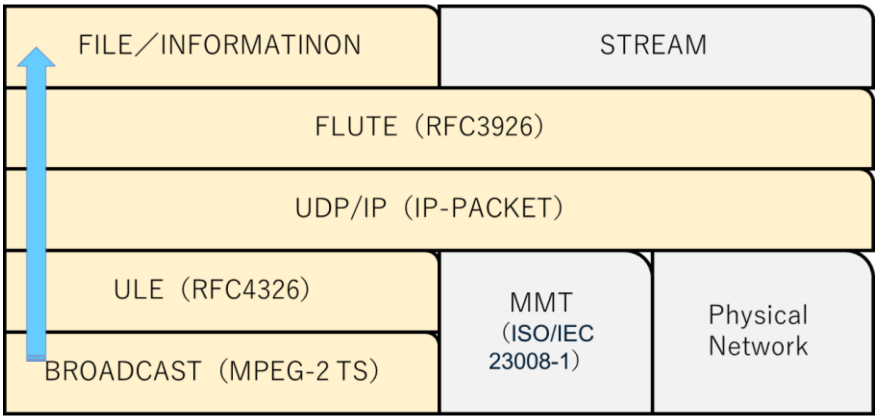


FIGURE: IPDC PROTOCOL STACK (RECEIVE DIRECTION)

<Reception Process>

To receive the broadcast, an IPDC receiver is used. Within the receiver, FLUTE is configured as the IP layer, so the FLUTE packets are reassembled to reconstruct the original file. This allows the file to be retrieved as it was at the time of transmission, and this file is then used to perform verification processes.

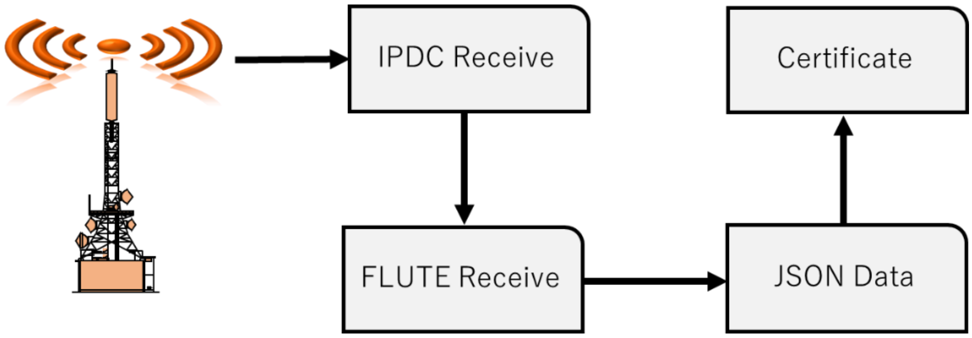


FIGURE : BLOCK DIAGRAM FOR IPDC RECEPTION

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

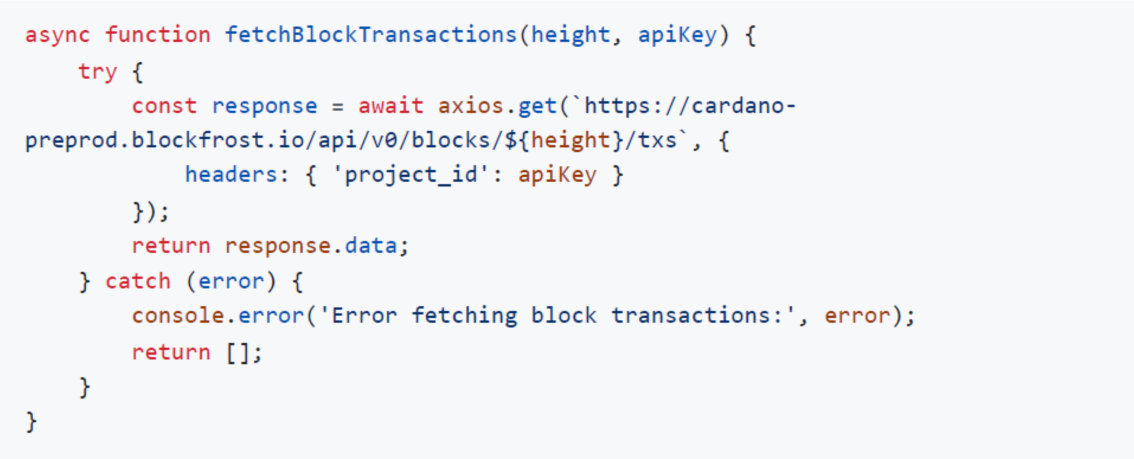
: Using the Node.js standard library for file system operations, the generated Merkle Path is saved as a JSON file.

<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 [];
    }
}
```

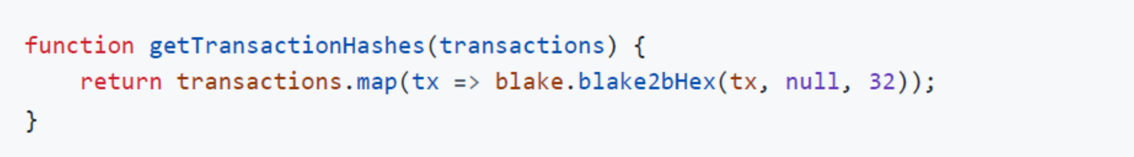
fetchBlockTransactions Function

: 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

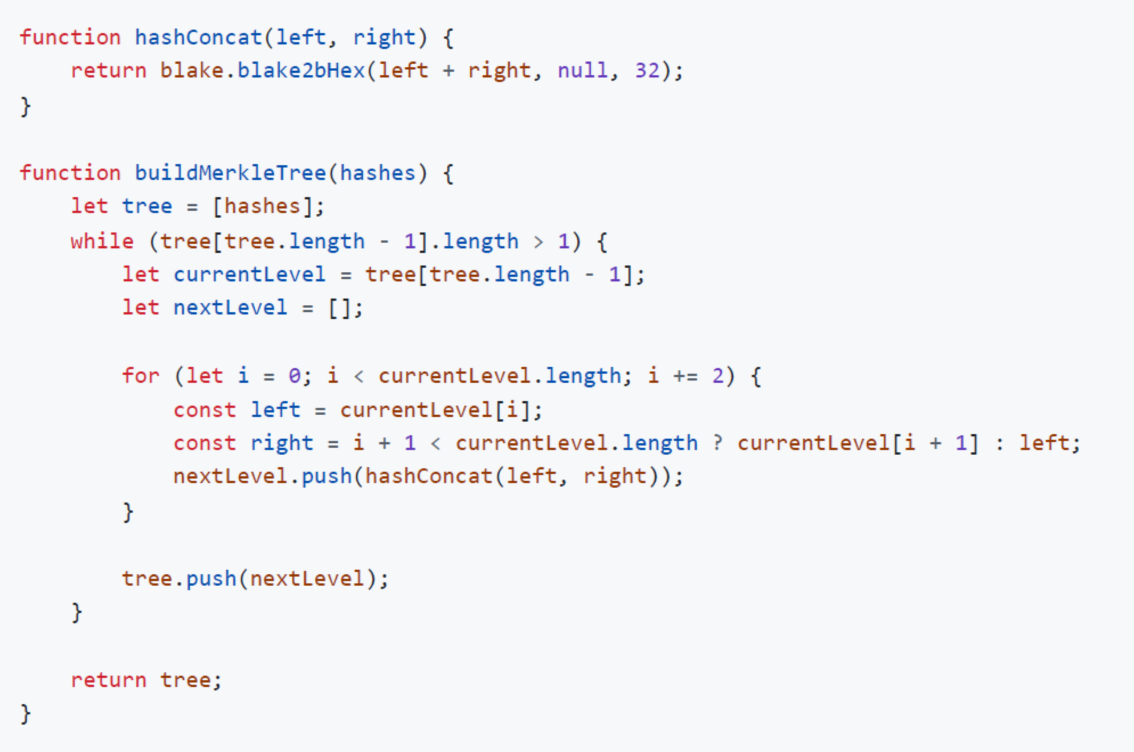


```
function getTransactionHashes(transactions) {
    return transactions.map(tx => blake.blake2bHex(tx, null, 32));
}
```

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;
}
```

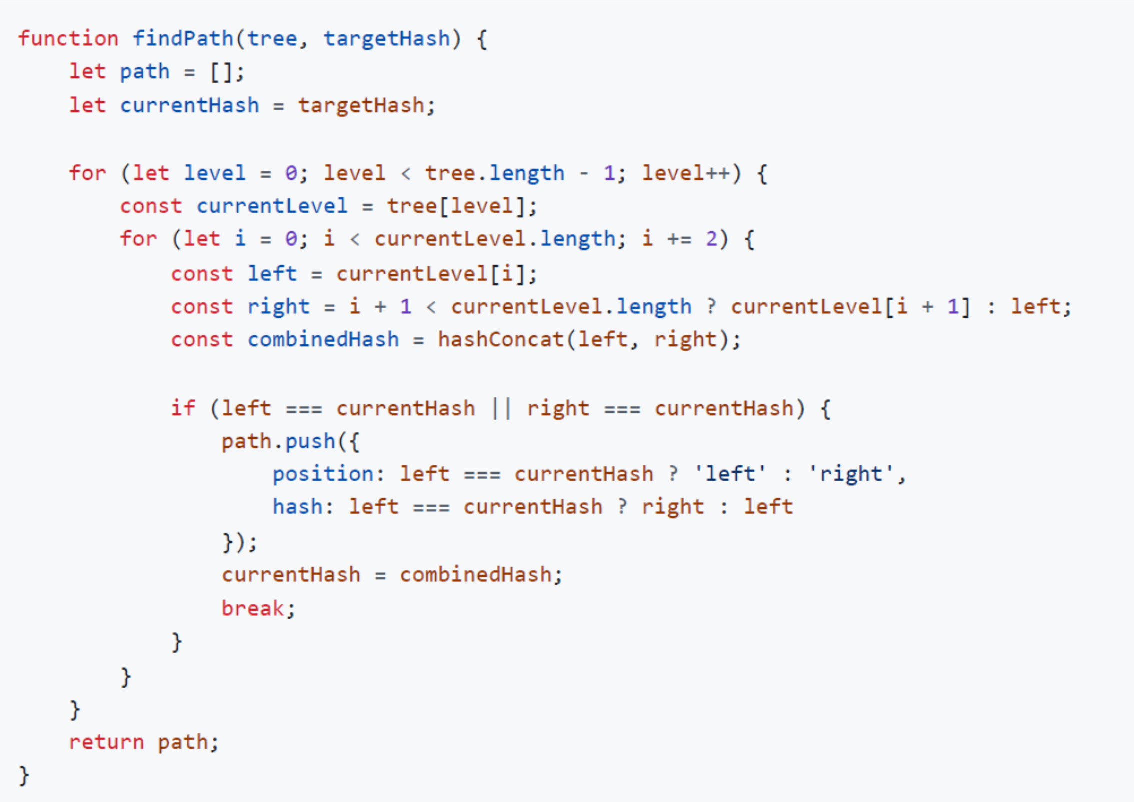
hashConcat Function

: 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

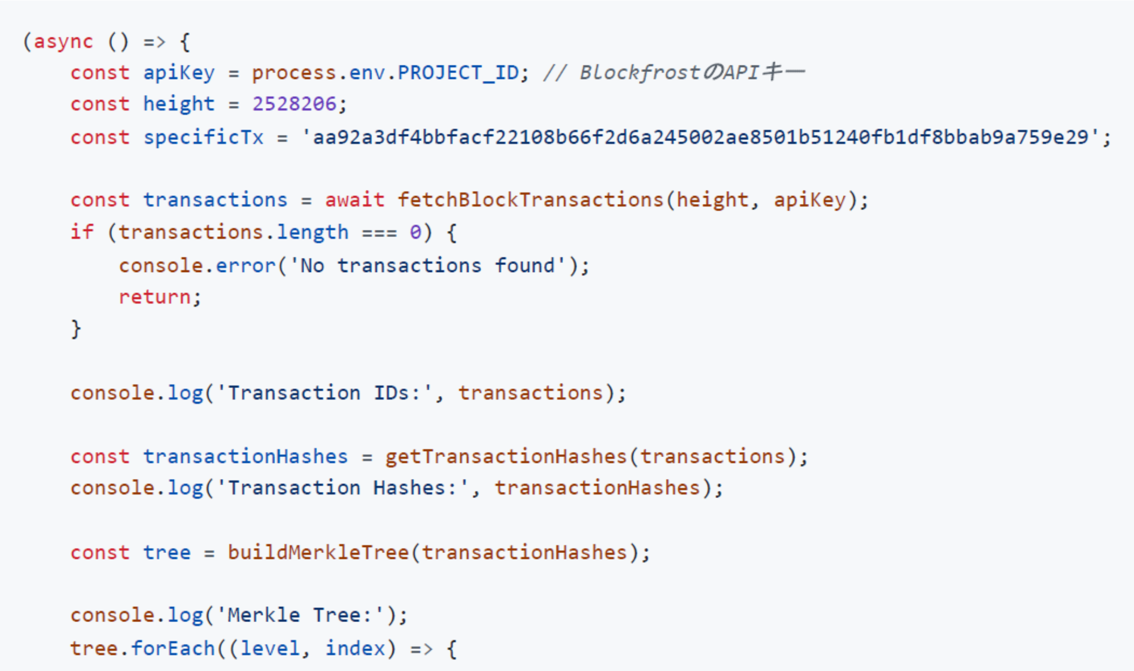


```
function findPath(tree, targetHash) {
    let path = [];
    let currentHash = targetHash;

    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;
}
```

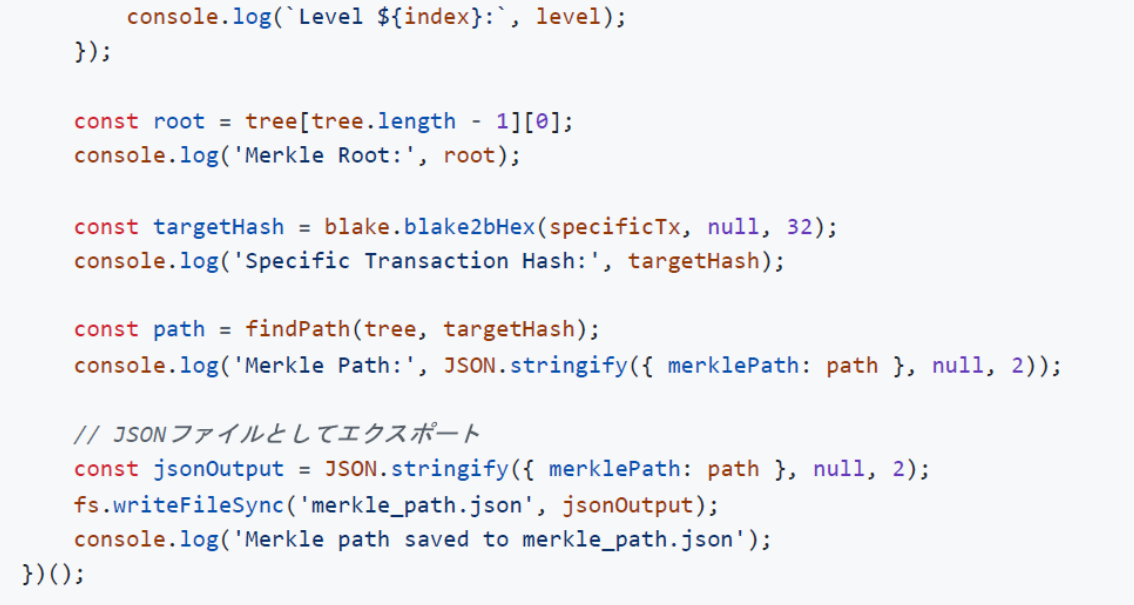
findPath Function

: 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

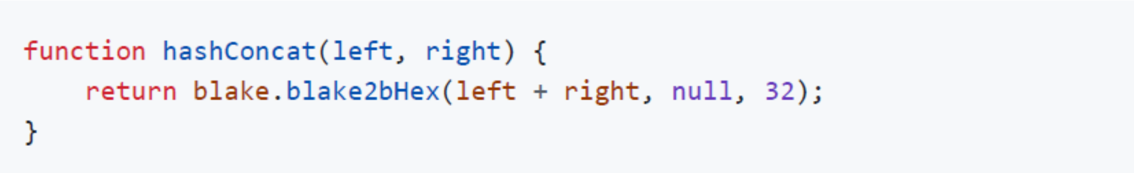
: Using the Blake2b hash function to calculate the transaction hash and verify the Merkle Path.

<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

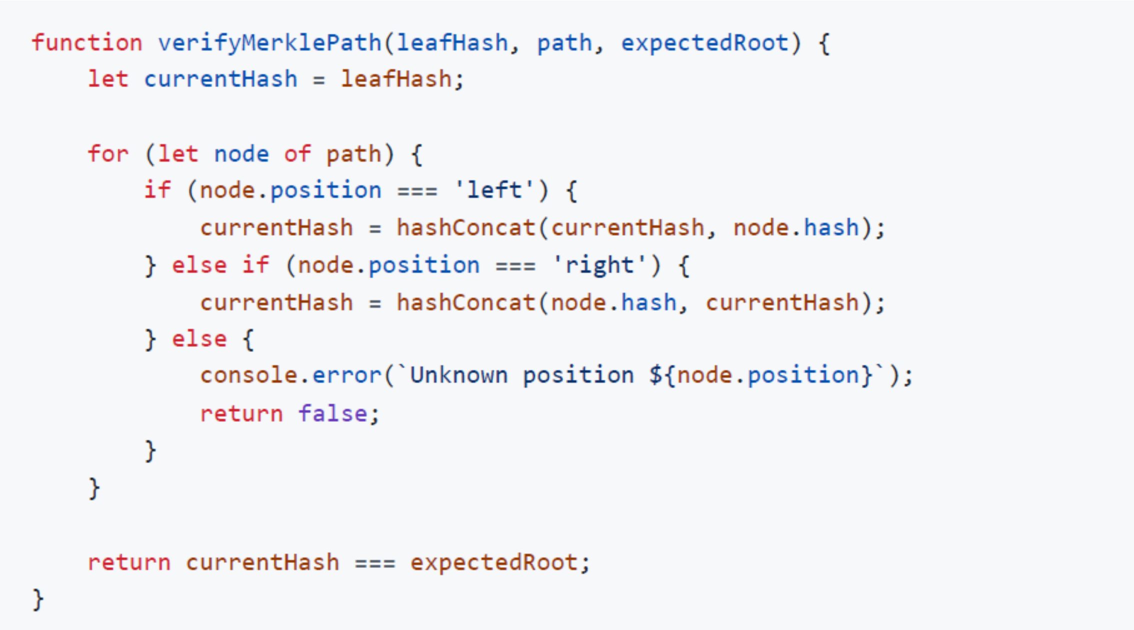


```
function hashConcat(left, right) {
    return blake.blake2bHex(left + right, null, 32);
}
```

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

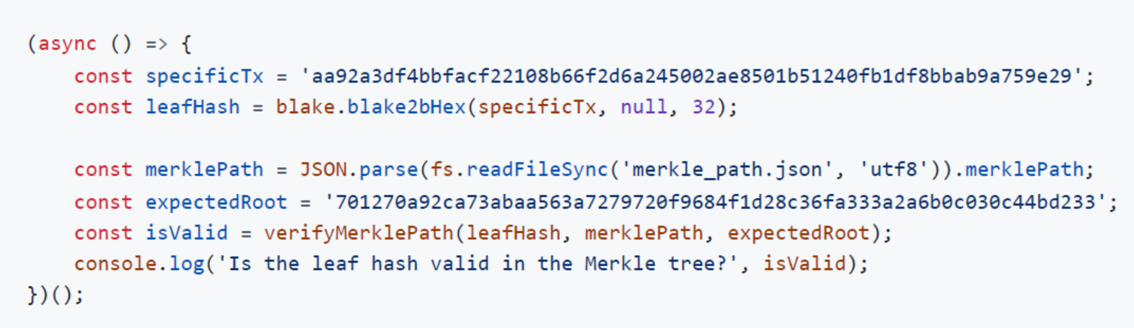


```
function verifyMerklePath(leafHash, path, expectedRoot) {
    let currentHash = leafHash;

    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.

8 Prototype Verification for Certificate Issuance（1）

8.1 Account Generation

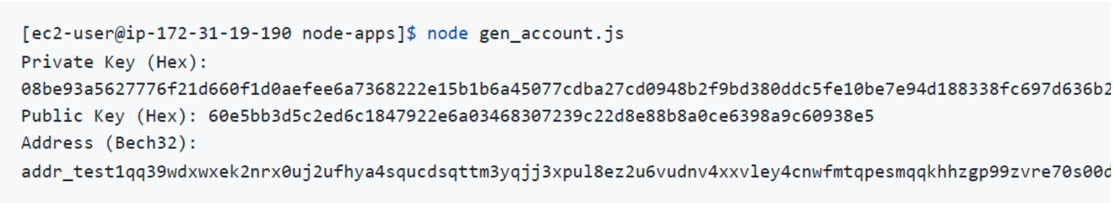
8.1.1 Account Generation

Using the Cardano Blockchain to Generate Keys. The private and public key pair is used for signing and authenticating transactions.

Prototype : Source Code（gen_account.js）

```
const CardanoWasm = require('@emurgo/cardano-serialization-lib-nodejs');
const crypto = require('crypto');

// Generate 32 bytes of entropy
const entropy = crypto.randomBytes(32);

// Generate a private key (BIP32 format)
const rootKey = CardanoWasm.Bip32PrivateKey.from_bip39_entropy(entropy, '');

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

// Derive address key and generate address
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();

// Display private key, public key, and address
console.log("Private Key (Hex):",
Buffer.from(privateKey.as_bytes()).toString('hex'));
console.log("Public Key (Hex):",
Buffer.from(publicKey.as_bytes()).toString('hex'));
console.log("Address (Bech32):", baseAddress);
```

Execution result



```
[ec2-user@ip-172-31-19-190 node-apps]$ node gen_account.js
Private Key (Hex):
08be93a5627776f21d660f1d0aefee6a7368222e15b1b6a45077cdba27cd0948b2f9bd380ddc5fe10be7e94d188338fc697d636b2
Public Key (Hex): 60e5bb3d5c2ed6c1847922e6a03468307239c22d8e88b8a0ce6398a9c60938e5
Address (Bech32):
addr_test1qq39wdxwxek2nrx0uj2ufhya4squcdsqttm3yqjj3xpul8ez2u6vudnv4xxvley4cnwfmtqpesmqqkhhzgp99zvre70s00d
```

Private Key (Hex):

08be93a5627776f21d660f1d0aefee6a7368222e15b1b6a45077cdba27cd0948b2f9bd380d

dc5fe10be7e94d188338fc697d636b20a4d9706284f2f3665468a1

Public Key (Hex):

60e5bb3d5c2ed6c1847922e6a03468307239c22d8e88b8a0ce6398a9c60938e5

Address (Bech32):

addr_test1qq39wdxwxek2nrx0uj2ufhya4squcdsqttm3yqjj3xpul8ez2u6vudnv4xxvley4cnwfmtqp

esmqqkhhzgp99zvre70s00d88q

8.2 Transactions and Certificate Issuance

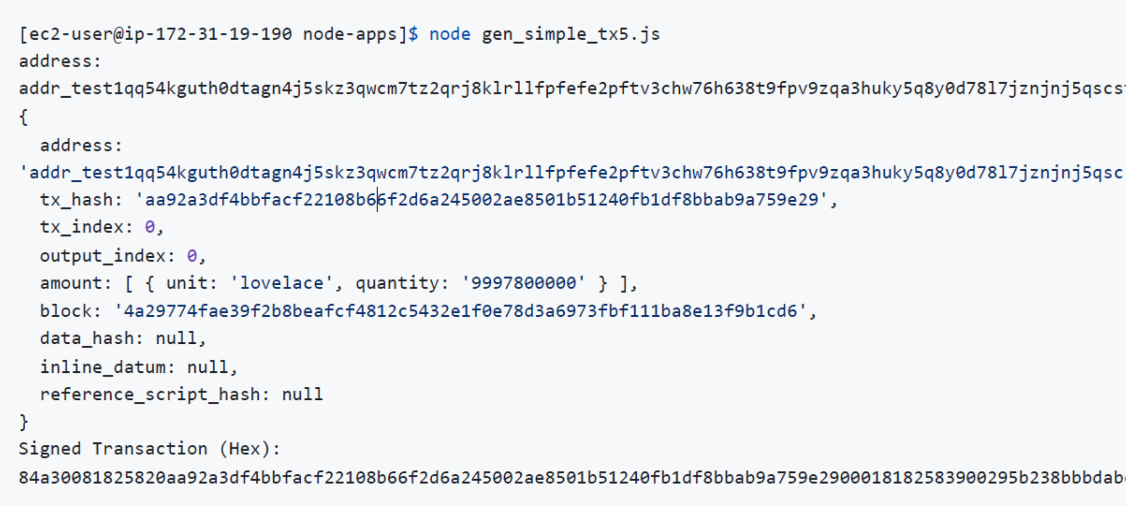
8.2.1 Transaction Creation

Using the Cardano Blockchain, Generate a Transaction with the Generated Private Key and Obtain the Serialized Transaction ID

Prototype : Source Code （gen_simple_tx5.js）

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

const privateKeyHex = process.env.PRIVATE_KEY_HEX;
const projectId = process.env.PROJECT_ID;
const apiBase = "https://cardano-preprod.blockfrost.io/api/v0";

//const address =
"addr_test1qq54kguth0dtagn4j5skz3qwcm7tz2qrj8klrllfpfefe2pftv3chw76h638t9fpv9z
qa3huky5q8y0d78l7jznjnj5qscstrm";


async function main() {

    const privateKey =
CardanoWasm.PrivateKey.from_extended_bytes(Buffer.from(privateKeyHex, 'hex'));

    // Generate address from private key
    const generatedPublicKey = privateKey.to_public();
    const address = CardanoWasm.BaseAddress.new(
        0, // Testnet = 0, Mainnet = 1
        CardanoWasm.StakeCredential.from_keyhash(generatedPublicKey.hash()),
        CardanoWasm.StakeCredential.from_keyhash(generatedPublicKey.hash())
    ).to_address().to_bech32();

    console.log("address:",address);

    const maxUTXO = await getMaxUTXO(address);
    const txHash = maxUTXO.txHash;
    const txIndex = maxUTXO.txIndex;
    const utxoAmount = parseInt(maxUTXO.amount);
    const fee = 200000; // Transaction fee (Lovelace)
    const sendAmount = utxoAmount - fee; //Transfer amount(after deducting
fees)
    const protocolParams = await getProtocolParameters();
    const txBuilder = CardanoWasm.TransactionBuilder.new(protocolParams);

    txBuilder.add_input(
        CardanoWasm.Address.from_bech32(address),
        CardanoWasm.TransactionInput.new(
            CardanoWasm.TransactionHash.from_bytes(Buffer.from(txHash,
"hex")),
            txIndex
        ),
        CardanoWasm.Value.new(CardanoWasm.BigNum.from_str(utxoAmount.toString(
)))
    );

    txBuilder.add_output(
        CardanoWasm.TransactionOutput.new(
            CardanoWasm.Address.from_bech32(address),
            CardanoWasm.Value.new(CardanoWasm.BigNum.from_str(sendAmount.toStr
ing()))
        )
    );

    txBuilder.set_fee(CardanoWasm.BigNum.from_str(fee.toString()));
    const txBody = txBuilder.build();
    const txHex = CardanoWasm.hash_transaction(txBody);
    const witnessSet = CardanoWasm.TransactionWitnessSet.new();
    const vkeys = CardanoWasm.Vkeywitnesses.new();
    vkeys.add(CardanoWasm.Vkeywitness.new(
        CardanoWasm.Vkey.new(privateKey.to_public()),
        privateKey.sign(txHex.to_bytes())
    ));
    witnessSet.set_vkeys(vkeys);

    // Build and serialize the transaction
    const signedTx = CardanoWasm.Transaction.new(txBody, witnessSet);
    const signedTxHex = Buffer.from(signedTx.to_bytes()).toString(‘hex’);
    console.log(“Signed Transaction (Hex):”, signedTxHex);
}

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.”);
        }
        // Select the largest UTXO
        let maxUTXO = utxos[0];
        utxos.forEach(utxo => {
            if (parseInt(utxo.amount[0].quantity) >
parseInt(maxUTXO.amount[0].quantity)) {
                maxUTXO = utxo;
            }
        });
        console.log(maxUTXO);
        return {
            txHash: maxUTXO.tx_hash,
            txIndex: maxUTXO.tx_index,
            amount: maxUTXO.amount[0].quantity
        };
    } catch (error) {
        console.error('Error fetching UTXOs:', error);
        throw error;
    }
}

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;
    }
}

main().catch(console.error);
```

Execution result



```
[ec2-user@ip-172-31-19-190 node-apps]$ node gen_simple_tx5.js
address:
addr_test1qq54kguth0dtagn4j5skz3qwcm7tz2qrj8klrllfpfefe2pftv3chw76h638t9fpv9zqa3huky5q8y0d78l7jznjnj5qscs
{
  address:
'addr_test1qq54kguth0dtagn4j5skz3qwcm7tz2qrj8klrllfpfefe2pftv3chw76h638t9fpv9zqa3huky5q8y0d78l7jznjnj5qsc
  tx_hash: 'aa92a3df4bbfacf22108b66f2d6a245002ae8501b51240fb1df8bbab9a759e29',
  tx_index: 0,
  output_index: 0,
  amount: [ { unit: 'lovelace', quantity: '9997800000' } ],
  block: '4a29774fae39f2b8beafcf4812c5432e1f0e78d3a6973fbf111ba8e13f9b1cd6',
  data_hash: null,
  inline_datum: null,
  reference_script_hash: null
}
Signed Transaction (Hex):
84a30081825820aa92a3df4bbfacf22108b66f2d6a245002ae8501b51240fb1df8bbab9a759e2900018182583900295b238bbbdab
```

Signed Transaction (Hex):

84a30081825820aa92a3df4bbfacf22108b66f2d6a245002ae8501b51240fb1df8bbab9a759e2900018182583900295b238bbbdabea275952161440ec6fcb1280391edf1ffe90a729ca8295b238bbbdabea275952161440ec6fcb1280391edf1ffe90a729ca81b0000000253e74500021a00030d40a100818258203745f60982871f6d49d0eb0cf1b02b4eafd128479933311292387df6ed86fcad58404770c9eb8bc2da34356650d35d2ad3e5fa56b1ec6babe31b88204df0f20bbe39941b758d708687898e0d1a683478107f7ddc5afe9df3970efb0f2c6509687f08f5f6

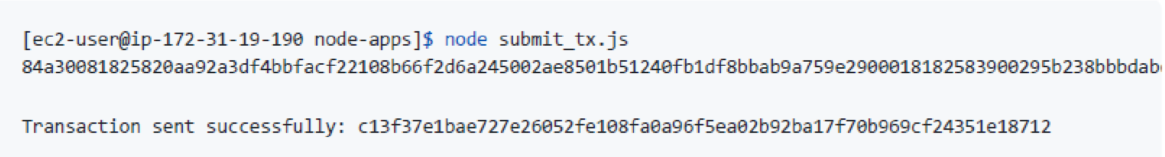
8.2.2 Transaction Sending

Using the Cardano Blockchain to Submit a Transaction.
The transaction is submitted using the transaction ID via the Blockfrost API, and a response is received.

Prototype : Source Code (submit_tx.js)

```
const axios = require('axios');
require('dotenv').config();// Load environment variables

const apiKey = process.env.PROJECT_ID; // Blockfrost API key
const signedTxHex = process.argv[2];// Get signedTxHex from command-line arguments

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

// Function to send a transfer transaction
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);
    }
}
sendTransaction();
```

Execution result



```
[ec2-user@ip-172-31-19-190 node-apps]$ node submit_tx.js
84a30081825820aa92a3df4bbfacf22108b66f2d6a245002ae8501b51240fb1df8bbab9a759e29000181825839002 95b238bbbdab

Transaction sent successfully: c13f37e1bae727e26052fe108fa0a96f5ea02b92ba17f70b969cf24351e18712
```

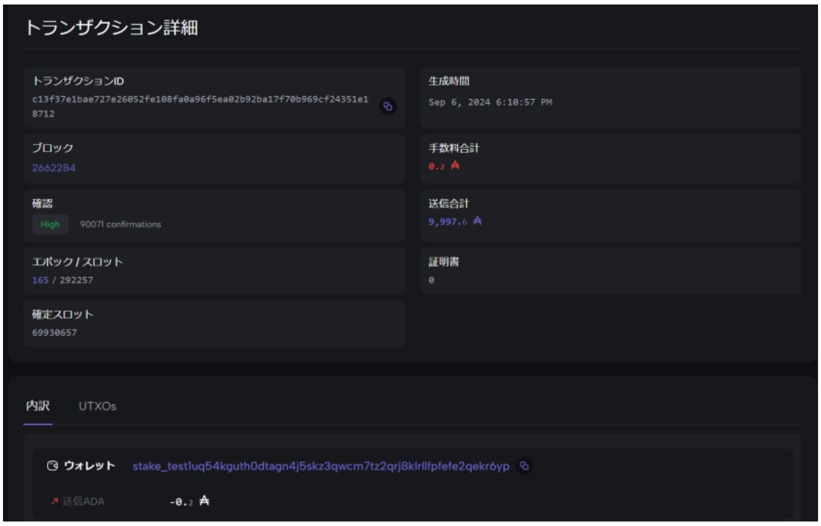
Transaction sent successfully:

c13f37e1bae727e26052fe108fa0a96f5ea02b92ba17f70b969cf24351e18712

Reference: Check with the Explorer

```
https://preprod.cardanoscan.io/transaction/c13f37e1bae727e26052fe108fa0a96f5ea02b92ba17f70b969cf24351e18712
```

Tranzaction Detail



トランザクション詳細

トランザクションID
c13f37e1bae727e26052fe108fa0a96f5ea02b92ba17f70b969cf24351e18712

生成時間
Sep 6, 2024 6:10:57 PM

ブロック
2662284

手数料合計
0.2 ₳

確認
High 90071 confirmations

送信合計
9,997.6 ₳

エポック / スロット
165 / 292257

証明書
0

確定スロット
69930657

ウォレット stake_test1uq54kguth0dtagn4j5skz3qwcm7tz2qrj8klrllfpfefe2qekr6yp

送信ADA -0.2 ₳

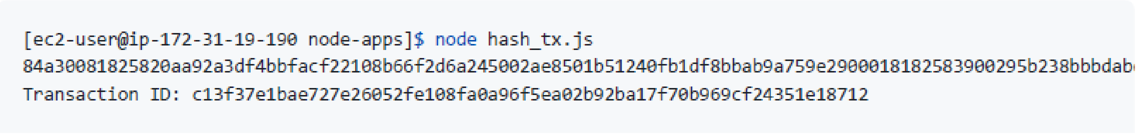
8.2.3 Transaction Hash

Using the Cardano Blockchain, Retrieve the Transaction ID from the Serialized Transaction.

Prototype : Source Code（hash_tx.js)

```
const serializedTxHex = process.argv[2]; // Get signedTxHex from command-line arguments
const CardanoWasm = require('@emurgo/cardano-serialization-lib-nodejs');

// Convert hexadecimal string to byte array
const txBytes = Buffer.from(serializedTxHex, 'hex');

// Deserialize the transaction
const transaction = CardanoWasm.Transaction.from_bytes(txBytes);

// Get the transaction body
const txBody = transaction.body();

// Get the transaction ID
const txHash = CardanoWasm.hash_transaction(txBody);

// Output the transaction ID in hexadecimal format
console.log(”Transaction ID:”, Buffer.from(txHash.to_bytes()).toString('hex'));
```

Execution result



```
[ec2-user@ip-172-31-19-190 node-apps]$ node hash_tx.js
84a30081825820aa92a3df4bbfacf22108b66f2d6a245002ae8501b51240fb1df8bbab9a759e29000181825839002095b238bbbdab
Transaction ID: c13f37e1bae727e26052fe108fa0a96f5ea02b92ba17f70b969cf24351e18712
```

Transaction ID:

c13f37e1bae727e26052fe108fa0a96f5ea02b92ba17f70b969cf24351e18712

8.3 Block Information Construction

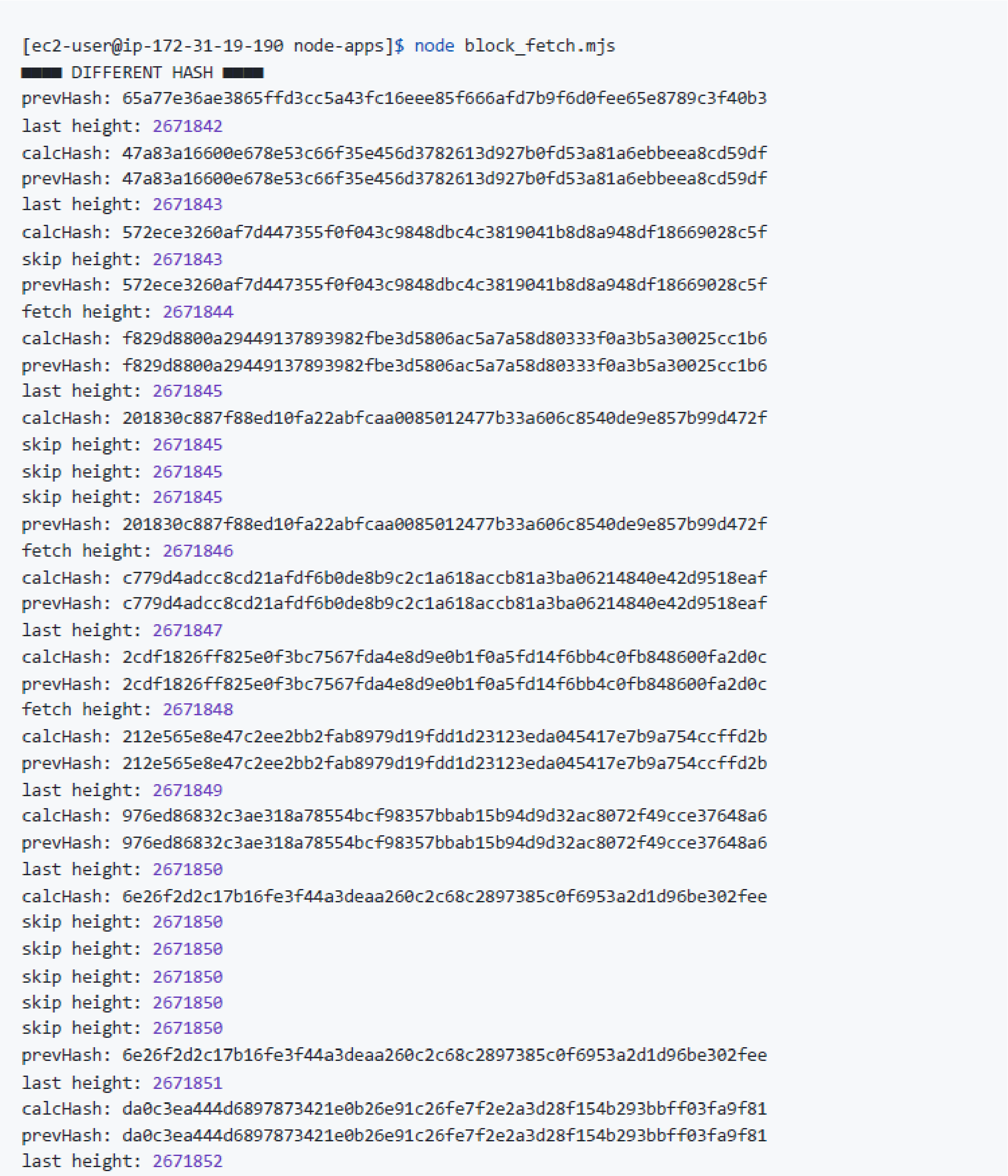
8.3.1 Research on Block Validation

Using the Cardano Blockchain, Retrieve Block Information from the Latest Block and Use the Block Height as a Key to Retrieve Block Information, then Compare It with the Previous Hash Value.

Prototype : Source Code （block_fetch.mjs）

```
import fetch from 'node-fetch';
import dotenv from "dotenv";

const res = dotenv.config()
const apiKey = process.env.PROJECT_ID; // Blockfrost API key
let lastHeight = 0;
let lastHash = "";

// Function to fetch the latest block information
const fetchBlockInfo = async () => {
    try {
        const response = await fetch('https://cardano-
preprod.blockfrost.io/api/v0/blocks/latest', {
            method: 'GET',
            headers: {
                'project_id': apiKey
            }
        });

        if (response.ok) {
            const blockInfo = await response.json();
            const currentHeight = blockInfo.height;
            // Execute processing only if the height of the last confirmed
block is different
            if (lastHeight !== currentHeight) {

                if (lastHeight > 0 && currentHeight > lastHeight + 1) {
                    // Fetch information for missing blocks
                    for (let h = lastHeight + 1; h < currentHeight; h++) {
                        await fetchBlockDetails(h);
                    }
                }
                if(lastHash !== blockInfo.previous_block){
                    console.log(”■■■■ DIFFERENT HASH ■■■■”);
                }
                console.log(”prevHash:”, blockInfo.previous_block);
//                console.log(’Latest Block information:’, blockInfo);
                console.log(”last height:”, blockInfo.height);
                console.log(”calcHash:”, blockInfo.hash);
                lastHash = blockInfo.hash;
                lastHeight = currentHeight;
            }else{
                console.log(”skip height:”, blockInfo.height);
            }
        } else {
            const error = await response.text();
            console.error(’Error fetching block information:’, error);
        }
    } catch (error) {
        console.error(’Error:’, error);
    }
};

// Function to fetch block details for the specified height
const fetchBlockDetails = async (height) => {
    try {
        const response = await fetch(`https://cardano-
preprod.blockfrost.io/api/v0/blocks/${height}`, {
            method: 'GET',
            headers: {
                'project_id': apiKey
            }
        });

        if (response.ok) {
            const blockDetails = await response.json();
            if(lastHash !== blockDetails.previous_block){
                console.log("■■■■ DIFFERENT HASH ■■■■");
            }
            console.log("prevHash:", blockDetails.previous_block);
            console.log("fetch height:", blockDetails.height);
            console.log("calcHash:", blockDetails.hash);
            lastHash = blockDetails.hash;
        } else {
            const error = await response.text();
            console.error(`Error fetching block information for height
${height}:`, error);
        }
    } catch (error) {
        console.error(`Error fetching block information for height
${height}:`, error);
    }
};

// Execute every 15,000 milliseconds (15 seconds)
setInterval(fetchBlockInfo, 15000);
fetchBlockInfo()
```

Execution result



```
[ec2-user@ip-172-31-19-190 node-apps]$ node block_fetch.mjs
■■■■ DIFFERENT HASH ■■■■
prevHash: 65a77e36ae3865ffd3cc5a43fc16eee85f666afd7b9f6d0fee65e8789c3f40b3
last height: 2671842
calcHash: 47a83a16600e678e53c66f35e456d3782613d927b0fd53a81a6ebbeea8cd59df
prevHash: 47a83a16600e678e53c66f35e456d3782613d927b0fd53a81a6ebbeea8cd59df
last height: 2671843
calcHash: 572ece3260af7d447355f0f043c9848dbc4c3819041b8d8a948df18669028c5f
skip height: 2671843
prevHash: 572ece3260af7d447355f0f043c9848dbc4c3819041b8d8a948df18669028c5f
fetch height: 2671844
calcHash: f829d8800a29449137893982fbe3d5806ac5a7a58d80333f0a3b5a30025cc1b6
prevHash: f829d8800a29449137893982fbe3d5806ac5a7a58d80333f0a3b5a30025cc1b6
last height: 2671845
calcHash: 201830c887f88ed10fa22abfcaa0085012477b33a606c8540de9e857b99d472f
skip height: 2671845
skip height: 2671845
skip height: 2671845
prevHash: 201830c887f88ed10fa22abfcaa0085012477b33a606c8540de9e857b99d472f
fetch height: 2671846
calcHash: c779d4adcc8cd21afdf6b0de8b9c2c1a618accb81a3ba06214840e42d9518eaf
prevHash: c779d4adcc8cd21afdf6b0de8b9c2c1a618accb81a3ba06214840e42d9518eaf
last height: 2671847
calcHash: 2cdf1826ff825e0f3bc7567fda4e8d9e0b1f0a5fd14f6bb4c0fb848600fa2d0c
prevHash: 2cdf1826ff825e0f3bc7567fda4e8d9e0b1f0a5fd14f6bb4c0fb848600fa2d0c
fetch height: 2671848
calcHash: 212e565e8e47c2ee2bb2fab8979d19fdd1d23123eda045417e7b9a754ccffd2b
prevHash: 212e565e8e47c2ee2bb2fab8979d19fdd1d23123eda045417e7b9a754ccffd2b
last height: 2671849
calcHash: 976ed86832c3ae318a78554bcf98357bbab15b94d9d32ac8072f49cce37648a6
prevHash: 976ed86832c3ae318a78554bcf98357bbab15b94d9d32ac8072f49cce37648a6
last height: 2671850
calcHash: 6e26f2d2c17b16fe3f44a3deaa260c2c68c2897385c0f6953a2d1d96be302fee
skip height: 2671850
skip height: 2671850
skip height: 2671850
skip height: 2671850
skip height: 2671850
prevHash: 6e26f2d2c17b16fe3f44a3deaa260c2c68c2897385c0f6953a2d1d96be302fee
last height: 2671851
calcHash: da0c3ea444d6897873421e0b26e91c26fe7f2e2a3d28f154b293bbff03fa9f81
prevHash: da0c3ea444d6897873421e0b26e91c26fe7f2e2a3d28f154b293bbff03fa9f81
last height: 2671852
```

8.3.2 Research on Rollback Process

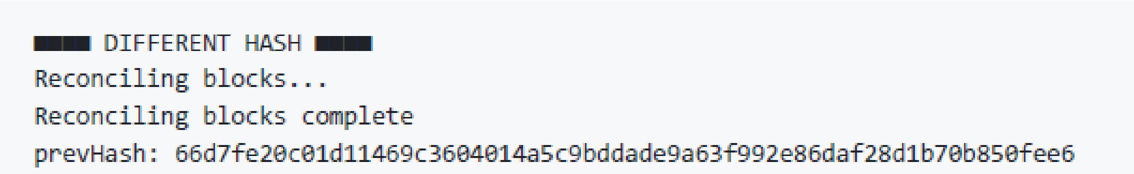
Using the Cardano Blockchain to Perform Repeated Rollback Processes

Prototype : Source Code （block_rollback.mjs）

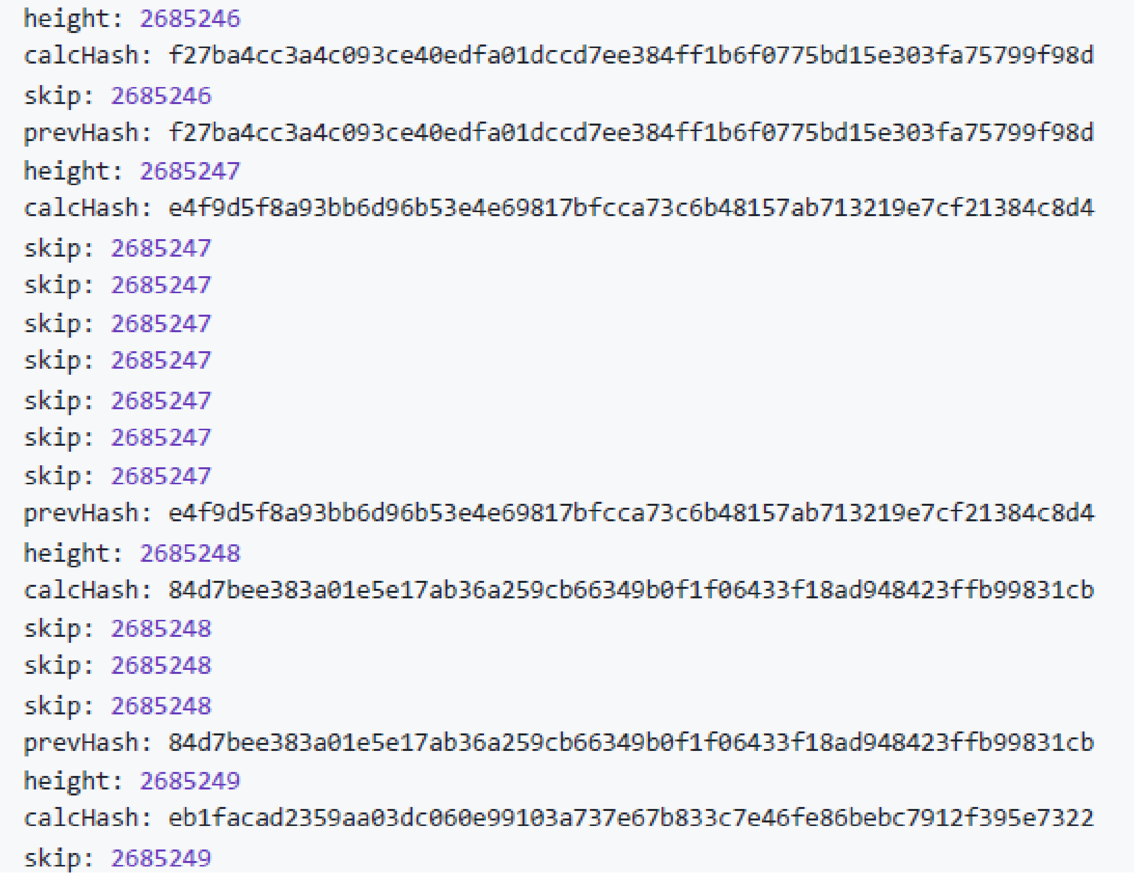
```
import fetch from 'node-fetch';
import dotenv from "dotenv";
dotenv.config()

const apiKey = process.env.PROJECT_ID; // Blockfrost API key
let lastHeight = 0;
let lastHash = "";
let pastHashes = [];

// Function to fetch block information from the specified endpoint
const fetchBlock = async (endpoint) => {
    try {
        const response = await fetch(endpoint, {
            method: 'GET',
            headers: {
                'project_id': apiKey
            }
        });

        if (response.ok) {
            return await response.json();
        } else {
            const error = await response.text();
            console.error(`Error fetching block information from ${endpoint}:`,
error);
            return null;
        }
    } catch (error) {
        console.error(`Error fetching block information from ${endpoint}:`, error);
        return null;
    }
};

// Fetch the latest block information and update the state
const fetchBlockInfo = async () => {
    const blockInfo = await fetchBlock('https://cardano-preprod.blockfrost.io/api/v0/blocks/latest');
    if (!blockInfo) return;

    const currentHeight = blockInfo.height;

    if (lastHeight !== currentHeight) {
        if (lastHeight > 0 && currentHeight > lastHeight + 1) {
            // Fetch information for missing blocks
            for (let h = lastHeight + 1; h < currentHeight; h++) {
                await fetchBlockDetails(h);
            }
        }

        if (lastHash !== blockInfo.previous_block) {
            console.log("■■■■ DIFFERENT HASH ■■■■");
            await reconcileBlocks(lastHeight);
        }

        updateState(blockInfo);
    } else {
        console.log("skip:", blockInfo.height);
    }
};
```

```
// Fetch the details of the block at the specified height
const fetchBlockDetails = async (height) => {
    const blockDetails = await fetchBlock(`https://cardano-
preprod.blockfrost.io/api/v0/blocks/${height}`);
    if (!blockDetails) return;

    if (lastHash !== blockDetails.previous_block) {
        console.log("■■■■ DIFFERENT HASH ■■■■");
        await reconcileBlocks(height - 1);
    }
    updateState(blockDetails);
};

// Function to reconcile block information
const reconcileBlocks = async (startHeight) => {
    console.log("Reconciling blocks...");
    let currentHeight = startHeight;

    while (currentHeight > 0) {
        const blockDetails = await fetchBlock(`https://cardano-
preprod.blockfrost.io/api/v0/blocks/${currentHeight}`);
        if (!blockDetails) break;

        updateState(blockDetails);
        const pastBlock = pastHashes.find(p => p.height === blockDetails.height);

        if (pastBlock && pastBlock.hash === blockDetails.hash) {
            console.log(`Match found at height ${currentHeight}`);
            lastHeight = currentHeight;
            lastHash = blockDetails.hash;
            break;
        } else {
            console.log(`No match found at height ${currentHeight}, fetching previous block`);
            currentHeight--;
        }
    }
    console.log("Reconciling blocks complete");

};

// Update block information and update the state
const updateState = (blockInfo) => {
    console.log("prevHash:", blockInfo.previous_block);
//    console.log('Block information:', blockInfo);
    console.log("height:", blockInfo.height);
    console.log("calcHash:", blockInfo.hash);

    lastHash = blockInfo.hash;
    lastHeight = blockInfo.height;

    // Add to the list of previous hash values
    pastHashes.push({ height: blockInfo.height, hash: blockInfo.hash });
    if (pastHashes.length > 100) { // 保持するハッシュ値の数を制限 // Limit the number of stored hash values
        pastHashes.shift();
    }
};

setInterval(fetchBlockInfo, 15000);// Execute every 15,000 milliseconds (15 seconds)
fetchBlockInfo();
```

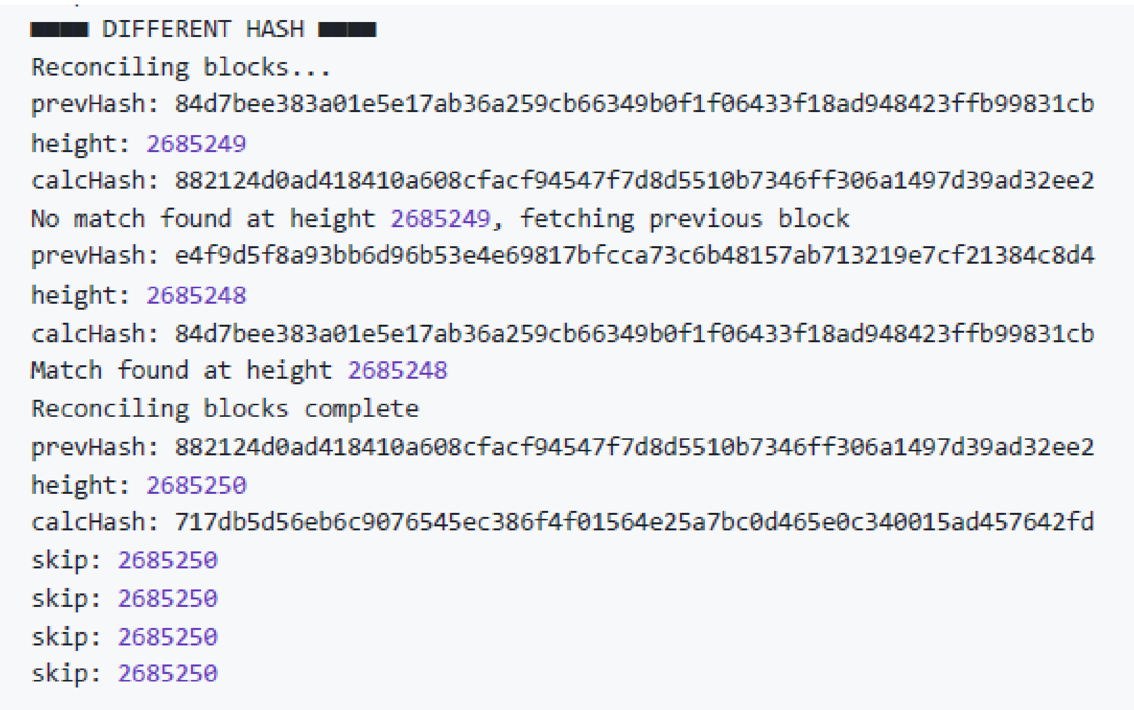
Execution result



```
■■■■ DIFFERENT HASH ■■■■
Reconciling blocks...
Reconciling blocks complete
prevHash: 66d7fe20c01d11469c3604014a5c9bddade9a63f992e86daf28d1b70b850fee6
```



```
height: 2685246
calcHash: f27ba4cc3a4c093ce40edfa01dccd7ee384ff1b6f0775bd15e303fa75799f98d
skip: 2685246
prevHash: f27ba4cc3a4c093ce40edfa01dccd7ee384ff1b6f0775bd15e303fa75799f98d
height: 2685247
calcHash: e4f9d5f8a93bb6d96b53e4e69817bfcca73c6b48157ab713219e7cf21384c8d4
skip: 2685247
skip: 2685247
skip: 2685247
skip: 2685247
skip: 2685247
skip: 2685247
skip: 2685247
prevHash: e4f9d5f8a93bb6d96b53e4e69817bfcca73c6b48157ab713219e7cf21384c8d4
height: 2685248
calcHash: 84d7bee383a01e5e17ab36a259cb66349b0f1f06433f18ad948423ffb99831cb
skip: 2685248
skip: 2685248
skip: 2685248
prevHash: 84d7bee383a01e5e17ab36a259cb66349b0f1f06433f18ad948423ffb99831cb
height: 2685249
calcHash: eb1facad2359aa03dc060e99103a737e67b833c7e46fe86bebc7912f395e7322
skip: 2685249
```



```
■■■■ DIFFERENT HASH ■■■■
Reconciling blocks...
prevHash: 84d7bee383a01e5e17ab36a259cb66349b0f1f06433f18ad948423ffb99831cb
height: 2685249
calcHash: 882124d0ad418410a608cfacf94547f7d8d5510b7346ff306a1497d39ad32ee2
No match found at height 2685249, fetching previous block
prevHash: e4f9d5f8a93bb6d96b53e4e69817bfcca73c6b48157ab713219e7cf21384c8d4
height: 2685248
calcHash: 84d7bee383a01e5e17ab36a259cb66349b0f1f06433f18ad948423ffb99831cb
Match found at height 2685248
Reconciling blocks complete
prevHash: 882124d0ad418410a608cfacf94547f7d8d5510b7346ff306a1497d39ad32ee2
height: 2685250
calcHash: 717db5d56eb6c9076545ec386f4f01564e25a7bc0d465e0c340015ad457642fd
skip: 2685250
skip: 2685250
skip: 2685250
skip: 2685250
```

8.3.3 Research on Transaction Root

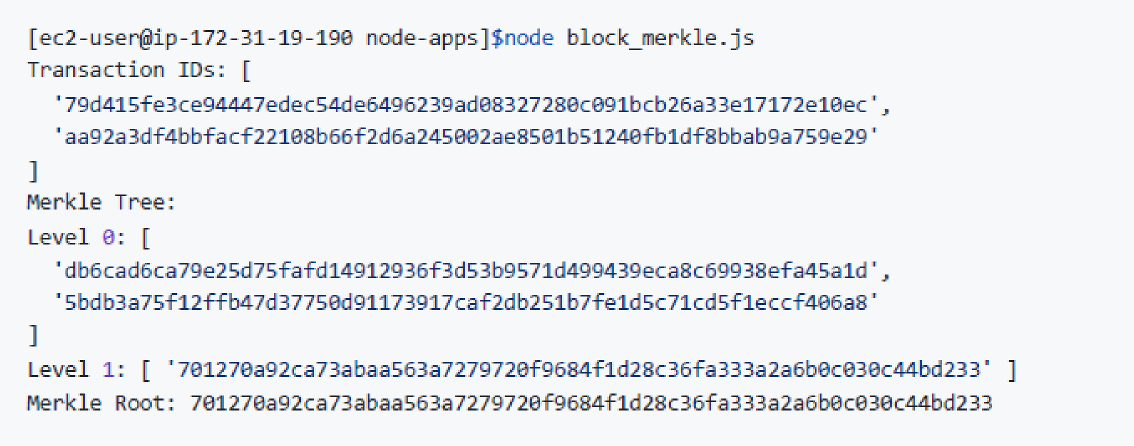
Investigating Transaction Roots Using the Cardano Blockchain

Prototype : Source Code （block_merkle.js）

```
const axios = require('axios');
const blake = require('blakejs');
require('dotenv').config();

// Fetch transactions for the specified block height from the Blockfrost API
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 [];
    }
}

// Generate BLAKE2b hashes from the array of transactions
function getTransactionHashes(transactions) {
    return transactions.map(tx => blake.blake2bHex(tx, null, 32));
}

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

```
}

// Build a Merkle tree
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;
}

(async () => {
    const apiKey = process.env.PROJECT_ID; // Retrieve the API key from
environment variables
    const height = 2528206; // Specify the block height

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

    console.log('Transaction IDs:', transactions);
    // Generate transaction hashes and build a Merkle tree
    const transactionHashes = getTransactionHashes(transactions);
    const tree = buildMerkleTree(transactionHashes);

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

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

Execution result



```
[ec2-user@ip-172-31-19-190 node-apps]$node block_merkle.js
Transaction IDs: [
  '79d415fe3ce94447edec54de6496239ad08327280c091bcb26a33e17172e10ec',
  'aa92a3df4bbfacf22108b66f2d6a245002ae8501b51240fb1df8bbab9a759e29'
]
Merkle Tree:
Level 0: [
  'db6cad6ca79e25d75fafd14912936f3d53b9571d499439eca8c69938efa45a1d',
  '5bdb3a75f12ffb47d37750d91173917caf2db251b7fe1d5c71cd5f1eccf406a8'
]
Level 1: [ '701270a92ca73abaa563a7279720f9684f1d28c36fa333a2a6b0c030c44bd233' ]
Merkle Root: 701270a92ca73abaa563a7279720f9684f1d28c36fa333a2a6b0c030c44bd233
```

Transaction IDs:

[’79d415fe3ce94447edec54de6496239ad08327280c091bcb26a33e17172e10ec’,
’aa92a3df4bbfacf22108b66f2d6a245002ae8501b51240fb1df8bbab9a759e29’]

Merkle Tree:

Level 0:

[’db6cad6ca79e25d75fafd14912936f3d53b9571d499439eca8c69938efa45a1d’,
’5bdb3a75f12ffb47d37750d91173917caf2db251b7fe1d5c71cd5f1eccf406a8’]

Level 1:

[’701270a92ca73abaa563a7279720f9684f1d28c36fa333a2a6b0c030c44bd233’]

Merkle Root:

701270a92ca73abaa563a7279720f9684f1d28c36fa333a2a6b0c030c44bd233

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

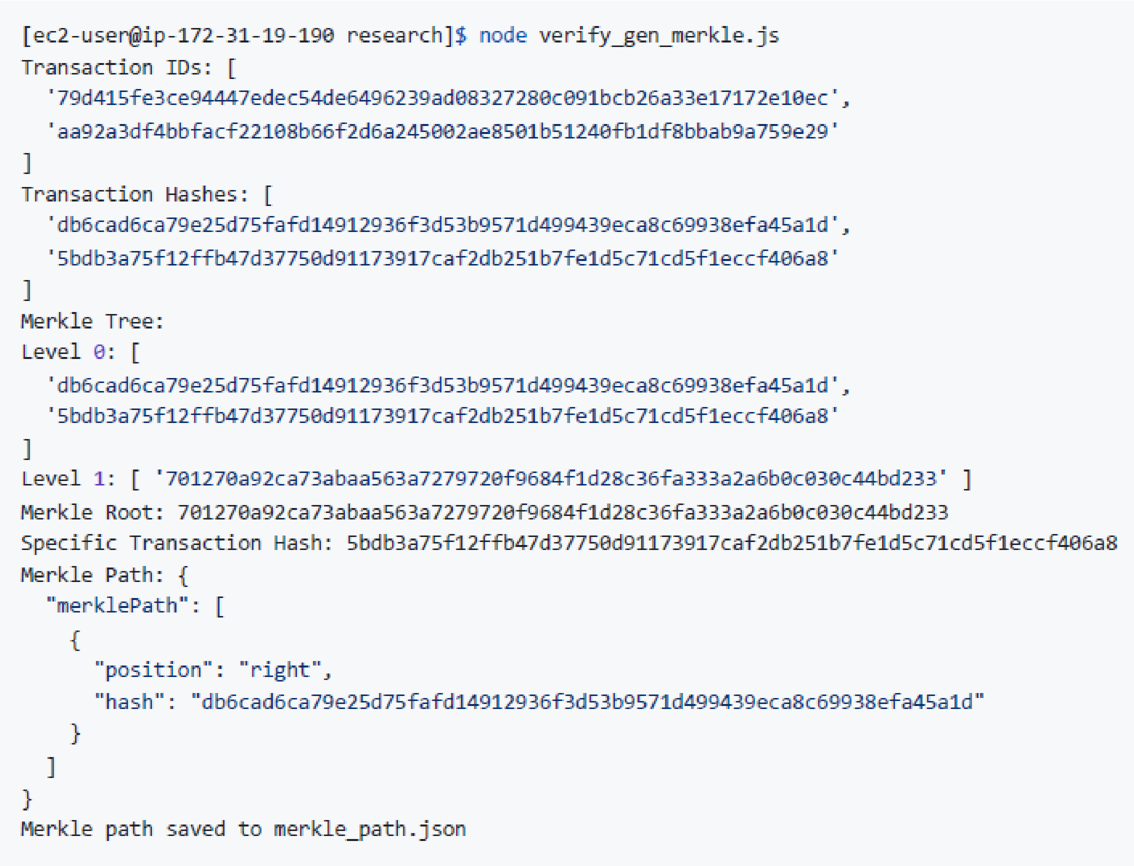
Prototype : Source Code（verify_gen_merkle.js)

```
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;
}
```

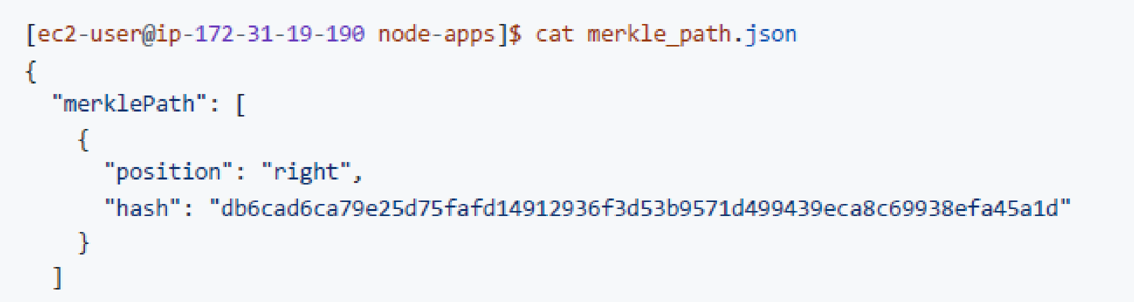
```
// Find the Merkle path for the specified transaction hash
function findPath(tree, targetHash) {
    let path = [];
    let currentHash = targetHash;

    // 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

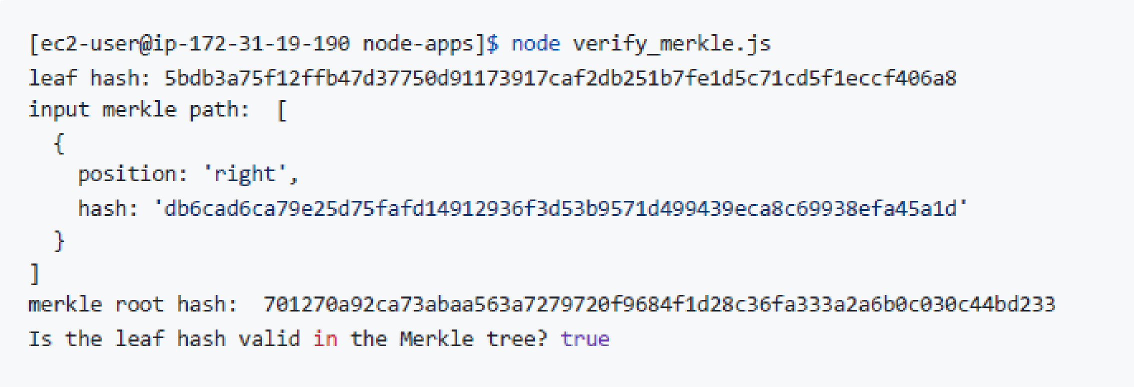
Using the Cardano Blockchain, Verify the Validity of Certificates with the Generated Information and Block Information Received via IPDC

Prototype : Source Code （verify_merkle.js）

```javascript
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

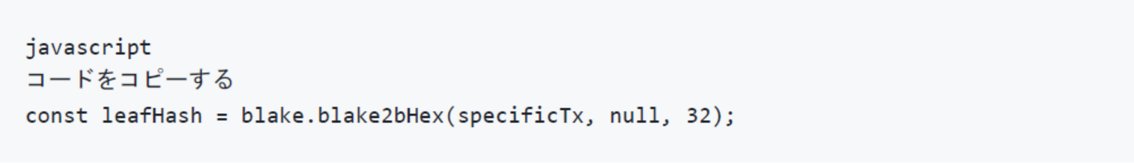
Is the leaf hash valid in the Merkle tree? true

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.

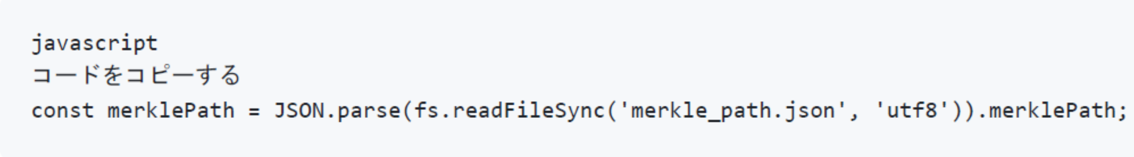


javascript
コードをコピーする

```
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.

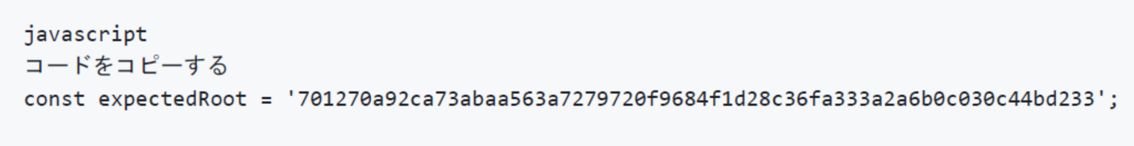


javascript
コードをコピーする

```
const merklePath = JSON.parse(fs.readFileSync('merkle_path.json', 'utf8')).merklePath;
```

9.1.3.3. Setting the Expected Merkle Root

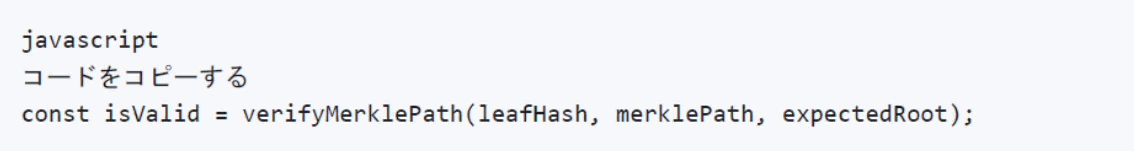
In the program, the previously calculated Merkle Root is set. This root is the topmost hash of the Merkle Tree and serves as the benchmark for verifying the validity.



```
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