IEICE TRANSACTIONS

on Information and Systems

VOL. E105-D NO. 2
FEBRUARY 2022

The usage of this PDF file must comply with the IEICE Provisions on Copyright.
The author(s) can distribute this PDF file for research and educational (nonprofit) purposes only.
Distribution by anyone other than the author(s) is prohibited.

A PUBLICATION OF THE INFORMATION AND SYSTEMS SOCIETY

The Institute of Electronics, Information and Communication Engineers
Kikai-Shinko-Kaikan Bldg., 5-8, Shibakoen 3 chome, Minato-ku, TOKYO, 105-0011 JAPAN

PAPER

Trail: An Architecture for Compact UTXO-Based Blockchain and Smart Contract

Ryunosuke NAGAYAMA[†*], *Nonmember*, **Ryohei BANNO**[†**], *and* **Kazuyuki SHUDO**[†a)], *Members*

SUMMARY In Bitcoin and Ethereum, nodes require a large storage capacity to maintain all of the blockchain data such as transactions. As of September 2021, the storage size of the Bitcoin blockchain has expanded to 355 GB, and it has increased by approximately 50 GB every year over the last five years. This storage requirement is a major hurdle to becoming a block proposer or validator. We propose an architecture called Trail that allows nodes to hold all blocks in a small storage and to generate and validate blocks and transactions. A node in Trail holds all blocks without transactions, UTXOs or account balances. The block size is approximately 8 kB, which is 100 times smaller than that of Bitcoin. On the other hand, a client who issues transactions needs to hold proof of its assets. Thus, compared to traditional blockchains, clients must store additional data. We show that proper data archiving can keep the account device storage size small. Then, we propose a method of executing smart contracts in Trail using a threshold signature. Trail allows more users to be block proposers and validators and improves the decentralization and security of the blockchain.
key words: *blockchain, storage consumption, accumulator*

1. Introduction

A blockchain is a distributed system with Byzantine fault tolerance. Because blockchain technology can manage a distributed ledger without a centralized system and makes tampering with past data difficult, it is used as the core technology for cryptocurrencies. In major blockchains such as Bitcoin [1], [2] and Ethereum [3], [4], a node validates each received transaction and generates a block from the transaction if it is valid. The node then broadcasts the block, and the receiving node validates the block and the transactions contained in it. In the transaction validation process, the nodes validate that the sender has the balance to be remitted in the transaction.

The security and decentralization of a blockchain improve as the number of nodes increases. Today's blockchains require much computational resources to operate a node and it is the obstacle to boot up a new node. A node must equip considerable hash calculation power to support a Proof-of-Work blockchain such as Bitcoin. A Proof-of-Stake blockchain such as Ethereum 2.0 does not require much calculation power but still require a large amount of storage. The data size of a blockchain is enormous because it includes all transactions or balances of all accounts. As of September 2021, the storage size of the Bitcoin blockchain has expanded to 355 GB, and it has increased by approximately 50 GB every year over the last five years [5]. The situation of Ethereum is similar to Bitcoin and the full data size of Ethereum reached 980 GB. Such a storage requirement is too high for embedded computers and smart devices. A blockchain has to overcome the storage problem to obtain number of node beyond server-class computers.

We propose the Trail architecture in which account balances are managed in the same way as unspent transaction outputs (UTXOs) using the TXO approach. TXOs are stored in a data structure called a TXO tree. The TXO tree is used to manage whether a TXO is used or unused and transactions contain Merkle proofs for TXOs; thus, nodes do not require past transactions and TXOs for validation. Furthermore, a node can generate a block from only the parent block and new transactions; therefore, the storage size for nodes is small.

The Trail architecture has the following advantages:

- Nodes do not store transactions, UTXOs and account balances; they store only blocks.
- A user can prove its balance to the other users without relying on nodes.
- The block size is only 8 kB and is constant regardless of the number of transactions.
- Trail does not depend on a specific consensus algorithm or any kind of fork choice rule.

Note that Trail itself does not fix performance, scalability and security including incentive model, though a consensus algorithm used in a Trail based blockchain dominates them.

This paper is an extended version of our previous work [6]. This paper shows such TXOs are treated in case the TXOs are approved and used in the same block in Sect. 6.1. In addition, we propose a method of executing smart contracts based on the Trail architecture.

The rest of this paper is organized as follows: Sect. 2 introduces related work on reducing blockchain storage. Section 3 describes major approaches to asset ownership management in blockchain. In Sect. 4, an overview of the Trail architecture is presented, and in the following three sections from 5 to 7, details are presented. Section 5 describes the synchronization of the blockchain. Section 6 presents methods of reducing the size of broadcast data. Section 7 describes data archiving to reduce client stor-

Manuscript received June 24, 2021.
Manuscript revised September 27, 2021.
Manuscript publicized November 9, 2021.
†The authors are with the School of Computing, Tokyo Institute of Technology, Tokyo, 152–8552 Japan.
*Presently, with Yahoo Japan Corporation.
**Presently, with Kogakuin University.
a) E-mail: shudo@is.titech.ac.jp
DOI: 10.1587/transinf.2021EDP7139

Copyright © 2022 The Institute of Electronics, Information and Communication Engineers

age demands. Section 8 describes extensions for executing smart contracts based on the Trail architecture. Finally, Sect. 9 presents the conclusion of this study.

2. Related Work

This section describes existing research that has attempted to reduce the storage size for blockchain.

Nakamoto [1] proposed pruning the Merkle tree. In Bitcoin, transactions that have been buried under a sufficient number of blocks are difficult to overturn. Therefore, nodes can save storage space by summarizing old transactions into a hash value of a parent node and discarding the transactions themselves. L. Quan et al. [7] analysed the distribution of the period from the approval of a UTXO to its use and, based on that analysis, proposed a method of properly discarding transactions. In these methods, a node needs to hold transactions for a certain period of time, and the node decides when to discard them. In Trail, nodes do not need to store any transactions, and balance management and data discarding are the responsibility of the owner of the balance.

Simplified payment verification (SPV) [1] and the light client protocol [8] are methods being researched by the Bitcoin and Ethereum communities. These methods allow clients to validate blocks using the Merkle proofs of transactions. However, these methods still require block proposers to store transactions and account balances, whereas in Trail, block proposers do not need to keep transactions and account balances. Furthermore, clients in Trail keep the Merkle proofs of their balances, so they can validate blocks in the same way as in these methods.

The stateless client concept [9] refers to a method in which validators need to keep only blocks containing the tree root. In current research, stateless clients can validate blocks but cannot generate blocks. Trail makes it possible for nodes to validate and generate blocks without storing data other than blocks by adding the information required for the verification and generation of a block to the transactions.

Utreexo [10] is a method of managing account assets using a Merkle forest, in which the leaf nodes are TXOs. Including the Merkle proof of a TXO in a transaction allows the TXO to be added to/deleted from the Merkle forest and enables nodes to validate transactions and generate blocks. In Utreexo, only unused TXOs become leaf nodes, so TXOs approved and used in the same block are not recorded in any blocks. On the other hand, Trail records all past TXOs.

Omniledger [11] reduces storage demand and improves throughput by means of sharding. Each shard is randomly assigned to validators periodically. When a client issues a transaction between different shards, the associated shards either commit or abort the transaction, and an aborted transaction is rolled back on each shard. Ethereum will also implement sharding [12]. Validators are randomly assigned, and each shard periodically commits blocks to the beacon chain, the blockchain that manages all shards.

In the pegged sidechain approach [13], another blockchain called a sidechain is created, which issues currency that can be exchanged with the Bitcoin blockchain. A transaction is issued in both blockchains, and the transaction is validated using SPV. This method reduces the storage demand by dividing the ledger and improves throughput, similar to sharding.

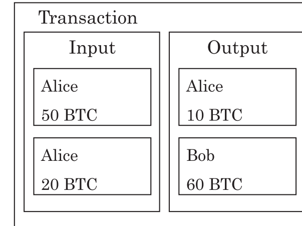

Fig. 1 A transaction sending 60 BTC from Alice to Bob.

Vault [14] is a method of reducing storage demand through sharding. In Vault, the balance of each account is stored in a Merkle tree, and each node is assigned to hold a part of the Merkle tree. Adding a Merkle proof to a transaction allows a node to validate and generate a block even if the node possesses only part of the Merkle tree. However, Vault requires the nodes to hold some of the leaf nodes and intermediate nodes of the Merkle tree. In Trail, nodes do not store transactions: they store the root and only one Merkle proof of the Merkle tree per block by using the TXO tree.

3. Asset Ownership Management in Blockchain

Our proposal, Trail is a UTXO-based blockchain. This section describes UTXO-based blockchain briefly to present required background.

There are two main methods of ownership management of assets in a blockchain: account-based and UTXO-based management.

Ethereum is an account-based blockchain. Account-based blockchains record the assets of all accounts in each block: an account and its assets are recorded in the block as one node in the Merkle tree. Validators check past blocks to confirm whether an account holds the balance sent in the current transaction.

By contrast, Bitcoin is a UTXO-based blockchain. A UTXO records the address of the owner and the number of coins. An account holds one or more UTXOs, and the total amount of the UTXOs represents the total assets of the account.

As shown in Fig. 1, a transaction consumes one or more UTXOs as input and creates one or more new UTXOs as output. The depicted transaction consumes a 50 BTC UTXO and a 20 BTC UTXO from Alice and creates a 10 BTC UTXO belonging to Alice and a 60 BTC UTXO belonging to Bob. If the transaction is approved, the assets of Alice decrease from 70 BTC to 10 BTC, and the assets of Bob increase by 60 BTC. As a result, 60 BTC is sent from Alice to Bob. The approved transaction is recorded in the block as one node in the Merkle tree. A validator checks past blocks to confirm whether any UTXO in the transac-

tion input has been used as a transaction input in the past; it is illegal to use a UTXO as a transaction input multiple times.

Trail is also a UTXO-based blockchain; however, it records whether a UTXO was used as an input in the past in a different way than Bitcoin. Bitcoin creates a Merkle tree with transactions as leaf nodes and records the tree in blocks, while Trail records Merkle trees with UTXOs as leaf nodes in blocks.

4. Overview of the Trail Architecture

This section describes the design of the Trail architecture. In the following, we use the terms "Trail node" and "client". A client simply issues a transaction. A Trail node validates transactions and blocks and generates blocks. A Trail node is usually also a client. Moreover, the nodes in the tree structure are simply called "nodes".

Since clients include proofs of their assets in transactions, Trail allows Trail nodes to validate transactions and generate blocks with minimal data storage. A simulator implementing the Trail architecture has been released on GitHub[†].

4.1 TXO Tree

First, we describe the TXO tree, which is the core concept of Trail. The TXO tree is a Merkle tree with leaf nodes that store the hash values of all TXOs approved in the past blocks in the blockchain in such a way that it can be determined whether a TXO has been used. In Bitcoin, a new Merkle tree is created for each block, while Trail updates a single TXO tree through the blockchain and records a root of the current TXO tree in the current block: no Trail node store the entire TXO tree. A TXO and its Merkle proof are held only by the client who is the owner of the TXO, and each Trail node holds only the root of the TXO tree, one hash value and one Merkle proof. As shown Fig. 2, a client holds only one part of the TXO tree that is related to its own TXO, and a Trail node holds only the root of the TXO tree, the hash value of the rightmost leaf node and its Merkle proof.

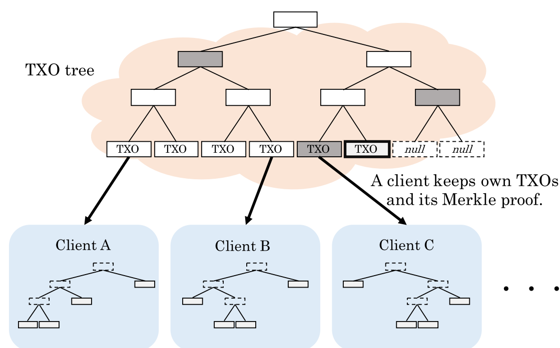

Fig. 2 Concept of the Trail architecture: The node with a thick border in the TXO tree is the rightmost leaf node, and the grey nodes are Merkle proofs for the rightmost leaf node.

[†]https://github.com/nagayamaryu/trail_simulator

A TXO tree is a perfect binary Merkle tree, and the leaf nodes of the TXO tree store the hash values of TXOs or null hash values. When new TXOs are accepted, leaf nodes that are still null are assigned to these TXOs from left to right. A null value for a leaf node indicates that a TXO has not yet been assigned to this leaf node. If the new TXO is unused, the fixed-length hash value hash(TXO) of the TXO is stored in the corresponding leaf node. On the other hand, when a TXO is used, the corresponding leaf node contains the hash value hash(TXO^2), which is the hash value of a concatenated binary of the TXO. Specifically, hash(b^2) represents the hash value of the binary formed by concatenating two copies of the binary of b. If the binary of a TXO is n bytes, then when the TXO is used, the value obtained by multiplying the hash function by a binary of $2n$ bytes, that is, the concatenation of two TXO binaries, is stored in the leaf node.

The root of the TXO tree, the index of the rightmost leaf node to which a TXO is assigned, and its hash value and Merkle proof are recorded in the block.

A node in the TXO tree is uniquely determined by an identifier called the branchID. The branchID is a concatenation of the height and the index from the left in the height. For example, if the height of the TXO tree is $2^8 = 256$, the branchID is 33 bytes; the first 1 byte represents the height of the node, and the remaining 32 bytes represent the index from the left in that height. Let branchID(h, i) be the branchID of the ith node from the left in height h.

4.2 Generation of a Transaction by a Client

The data structures of a TXO and a transaction are shown in Tables 1 and 2, respectively.

Clients store their own TXOs and the update histories of their Merkle proofs. A transaction must be generated by clients who consider the same block to be the latest block.

Table 1 Data fields of a transaction output (TXO).

Field	Description	Size
Index	Index of the corresponding leaf node.	32 bytes
ParentBlock	Hash value of the parent block of the block that added this TXO to the TXO tree.	32 bytes
OwnerAddress	Address of the owner of this TXO.	32 bytes
Balance	Asset amount.	32 bytes

Table 2 Data fields of a transaction.

Field	Description	Size
BlockHash	Hash value of the block on which proofs of this transaction are based.	32 bytes
Inputs	List of unused TXOs and their Merkle proofs. A Merkle proof is an array of 255 hash values.	
Outputs	List of new TXOs.	
Sigs	Signatures of accounts who own TXOs in Inputs or Outputs.	

Algorithm 1: Validation of a transaction.

```
Input: A transaction tx, the latest block B_latest, valid
       transactions Tx_valid
if tx.BlockHash ≠ hash(B_latest) then
  | return False
end
if tx.Inputs ∩ ⋃^{Tx_valid} tx_v.Inputs ≠ ∅ then
  | return False
end
for TXO, Proof ∈ tx.Inputs do
  | if Proof is empty then
  |   | if TXO ∉ ⋃^{Tx_valid} tx_v.Outputs then
  |   |   | return False
  |   | end
  | else
  |   | if Proof is invalid then
  |   |   | return False
  |   | end
  | end
end
if Σ^{tx.Inputs} TXO.Balance < Σ^{tx.Outputs} TXO.Balance then
  | return False
end
return True
```

Otherwise, nodes cannot validate the transaction and the transaction is not approved. A client can check the Merkle proof of another client if it has the root of the TXO tree of the latest block. For example, when checking whether a certain TXO is unused, a client checks whether the root value of the Merkle tree calculated from the hash value of the TXO and its Merkle proof matches the root value of the TXO tree of the latest block. If it does not match, then either the TXO has been used, the TXO has not yet been approved, or the Merkle proof is invalid. However, a Merkle proof is not required when using an unapproved TXO as an input TXO.

If all TXOs in Inputs are unused and the Merkle proof is correct, then the client creates Outputs such that the total amount is less than or equal to the total amount of Inputs minus the fee. At this time, no one knows which leaf node of the TXO tree the TXO will be assigned to, so the Index of the TXO in Outputs is empty.

Afterwards, the client broadcasts the transaction.

4.3 Validation of a Transaction by Trail Nodes

Trail nodes validate received transactions and generate blocks from valid transactions. A node validates a transaction by checking the following four conditions as shown in Algorithm 1.

- BlockHash of the transaction is the same as the block that the node considers as the latest block.
- TXOs in Inputs of the transaction are not included in Inputs of other valid transactions.
- The root value calculated based on hash values and Merkle proofs of TXOs in Inputs of the transaction is the same as the root value of the TXO tree of the latest block.
- The total amount of Outputs of the transaction is less than or equal to the total amount of Inputs of the transaction minus the fee.

The validator does not need all past blocks to validate whether a TXO was previously used.

4.4 Update of the Merkle Proof in a Transaction by Trail Nodes

When a new block is created, the TXO tree may be updated before a valid transaction is included in the block. In this case, the Merkle proof in the transaction will no longer be valid and cannot be included in the block.

For example, consider two transactions containing the Merkle proof of the TXO tree of block n, T_1 and T_2. If T_1 is contained in child block $n+1$ of block n, then the node of the TXO tree will be updated. Therefore, the root of the TXO tree of block $n+1$ and the root of the Merkle tree calculated from the TXO and its Merkle proof in transaction T_2 will not match because the root of the Merkle tree calculated from the TXO and its Merkle proof in transaction T_2 correspond to the root of the TXO tree of block n. Therefore, T_2 cannot be included in block $n+2$ because it is not a valid transaction for block $n+1$.

Thus, for a pending transaction, the Trail node needs to update the Merkle proof in the transaction. The proof can be updated based on the information of the approved transaction in the blocks generated during the pending transaction because the hash value of the updated node of the TXO tree can be calculated from the information contained in those transactions.

The client does not include the Merkle proof when signing a transaction, as the Trail node may update the Merkle proof.

The process of the update of the Merkle proof by a Trail node is similar to the method proposed for Vault [14].

4.5 Generation of a Block by a Trail Node

The data structure of a block is shown in Table 3. The block size is 8288 bytes, which is approximately one hundredth of the block size in Bitcoin [15].

In the block generation process, the Trail node computes the new root of the TXO tree. First, hash values are assigned to leaf nodes (height 0) of the TXO tree in the following order: Merkle proofs in Inputs; RightmostProof; RightmostHash; hash values of TXOs in Outputs, hash(TXO); and hash values of TXOs in Inputs, hash(TXO^2).

The TXOs in Outputs of all valid transactions are assigned in order from the leaf node of RightmostIndex+1 without gaps. At this time, the Index values of the TXOs in Outputs are fixed. Therefore, the hash value of each TXO in Outputs, including the Index value, is assigned to a leaf

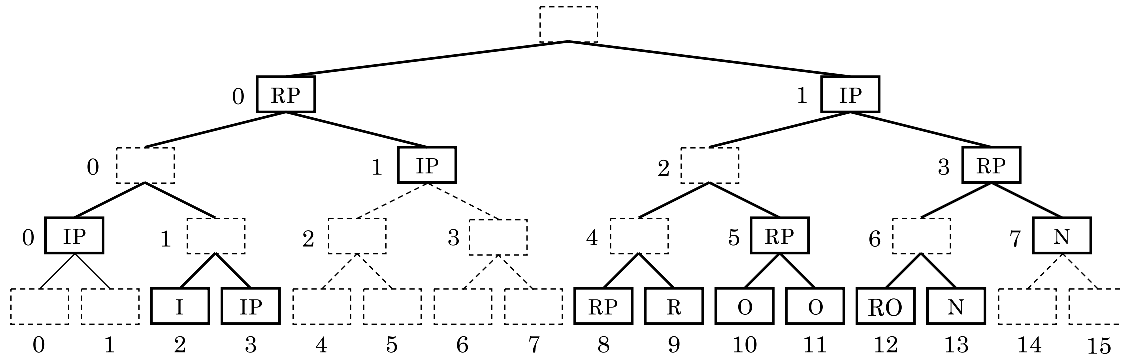

Fig. 3 Example of assigning hash values to a TXO tree from transaction and parent block data: Node I is assigned the hash value of a TXO in Inputs. Node IP is assigned the hash value of the corresponding Merkle proof in Inputs. Node R is assigned the RightmostHash of the parent block. Node RP is assigned the RightmostProof of the parent block. Nodes O are assigned the hash values of TXOs in Outputs. Node RO is the rightmost node that is assigned the hash value of a TXO in Outputs. Node N is assigned a null hash value. The nodes indicated by dashed boxes are initially assigned no hash value. The block proposer computes the hash values of the parent nodes along the thick lines between nodes. The numbers beside the nodes are the indexes of the nodes at that height.

Table 3 Data fields of a block.

Field	Description	Size
Parent	Hash value of the parent block.	32 bytes
Root	Root of the TXO tree.	32 bytes
RightmostIndex	Index of the rightmost leaf node to which a TXO is assigned.	32 bytes
RightmostHash	Hash value of the leaf node corresponding to RightmostIndex.	32 bytes
RightmostProof	Merkle proof of the leaf node corresponding to RightmostIndex.	255×32 bytes

node. Since the TXOs in Inputs are used, the hash value flagging a TXO as used, hash(TXO^2), is assigned to each corresponding leaf node. Furthermore, RightmostHash is assigned to the leaf node whose index is RightmostIndex.

Note that hash values are assigned to only a portion of the leaf nodes. The block proposer computes the hash value of the parent node from the hash values assigned to the leaf nodes. Hash values are always assigned to the sibling nodes of leaf nodes to which hash values are assigned, except the rightmost leaf node. If a hash value is not assigned to the sibling node of the rightmost node, then the hash value of the parent node is computed under the assumption that a null hash value is assigned to that sibling node.

Then, hash values are assigned to the nodes at height 1 in the following order: Merkle proofs in Inputs, RightmostProof, and hash values computed from the leaf nodes. Sibling nodes of nodes to which hash values have been assigned, except the rightmost node, are always assigned hash values. If a hash value is not assigned to the sibling node of the rightmost node, then hash(hash($null$)^2) is assigned, and the Trail node computes the hash value of the parent node.

Similarly, hash values are assigned to the nodes at height $h+1$ in the following order: Merkle proofs in Inputs, RightmostProof, and hash values computed from the nodes at height h. Additionally, the block proposer computes the hash values of the nodes at height $h+1$. Finally, the block proposer obtains a new root of the TXO tree.

The block proposer generates a new block with the index of the rightmost leaf node to which the newly added TXO is assigned as RightmostIndex, the hash value of that node as RightmostHash, and the Merkle proof of that node as RightmostProof. Then, the block proposer broadcasts the new block and the approved transaction to other Trail nodes.

In Fig. 3, the leaf nodes at indexes 2, 3, 8, 9, 10, 11, 12, and 13 are assigned hash values. The node at index 2 is assigned the hash value of a TXO in Inputs, hash(TXO^2), and the node at index 3 is assigned the corresponding Merkle proof in Inputs. The node at index 8 is assigned RightmostProof, and the node at index 9 is assigned RightmostHash; that is, the RightmostIndex of the parent block is 9. The nodes at indexes 10, 11, and 12 are assigned the hash values of TXOs in Outputs, hash(TXO). Since the node at index 12 is the rightmost node assigned a hash value, the RightmostIndex of the new block will be 12.

The sibling nodes of the nodes at indexes 2, 3, 8, 9, 10, and 11 are assigned hash values; however, the sibling node of the node at index 12 has not yet been assigned a TXO. Therefore, the node at index 13 is assigned a null hash value.

At this time, since the sibling nodes of the leaf nodes to which hash values have been assigned have all been assigned hash values, the block proposer can compute the hash values of the parent nodes.

Then, the block proposer assigns hash values to nodes at height 1. The node at index 0 is assigned the Merkle proof in Inputs, and the node at index 5 is assigned RightmostProof. Furthermore, the nodes at indexes 1, 4, and 6 are assigned hash values computed from the leaf nodes. The sibling nodes of the nodes at indexes 0, 1, 4, and 5 are also assigned hash values. However, since the sibling node of the node at index 6 has not been assigned a hash value, the node at index 7 is assigned a null hash value, hash(hash($null$)^2).

At this time, since the sibling nodes of the nodes at height 1 to which hash values have been assigned have all been assigned hash values, the block proposer can compute the hash values of the parent nodes.

The nodes at heights 2 and 3 are assigned hash values in the same way. The hash values of nodes other than those to which hash values have been assigned are not updated when computing the new root of the TXO tree.

In this case, the new RightmostIndex is 12, the new RightmostHash is the hash value of the node at branchID(0, 12), and the new RightmostProof consists of the hash values of the nodes at branchID(0, 13), branchID(1, 7), branchID(2, 2), and branchID(3, 0).

4.6 Updating Client Data

Finally, we describe the procedure for updating the data of a client. When a block is created, the client receives the updated hash values of the nodes in the TXO tree to update the Merkle proof it holds.

Here, it is assumed that there is no fork or churn and that messages do not disappear during communication with peers, that is, the client receives all updates for the blocks and the TXO tree; we consider the case in which the client cannot receive some portion of the updates in Sect. 5.

The client receives the new block, the new TXOs, the used TXOs, and the hash values of the nodes that have been assigned hash values when computing the new root of the TXO tree. If the client has its own TXO among the new TXOs, then the client keeps that TXO as an unused TXO. If the client has its own TXO among the used TXOs, then the client marks that TXO as used.

If the Merkle proof of the unused TXOs is updated or the client has not yet obtained it, then the client recodes the hash value of the Merkle proof together with the branchID and the hash value of the new block.

5. Data Synchronization

In the previous section, we assumed that there were no forks and that the client was always connected to the network. However, in an actual blockchain network, forks will occur, and clients will repeatedly connect to and disconnect from the network. In this case, a client will need to obtain the blockchain data and synchronize its own data with the blockchain.

Furthermore, a Trail node may have newly joined or left the network and may not have obtained some blocks. In this case, the Trail node will need to obtain the blockchain data and synchronize its own data.

5.1 Full Node

For data synchronization, a full node with all transactions is required. Full nodes are a subclass of Trail nodes. Normal Trail nodes do not need to hold transactions approved in past blocks, and a Trail node does not need to be a full node for block generation and transaction validation. A full node requires more storage space than a normal Trail node but approximately the same storage space as a full node in Bitcoin.

When a fork or churn occurs, a client obtains TXO tree updates that have not been obtained from a full node. Algorithm 2 shows how the client can obtain TXO tree updates from a full node. It is sufficient for the client to obtain only the latest update for each node of the TXO tree related to its own TXOs rather than all the updates.

The Trail node requests the transactions approved by the missing blocks from the full node. The blocks themselves can be obtained from other Trail nodes that are not full nodes, but the Trail node needs to obtain the transactions from a full node to validate the missing blocks.

Algorithm 2: Downloading of latest TXO tree updates from a full node.

```
Input: List of branchIDs ids; hash values of blocks from,
       to
Output: Map that maps the hash values of nodes to
        branchIDs and hash values of blocks
begin
  if from is not an ancestor of to then
    return error
  else
    nodes ← nodes for which branchID is in ids;
    return latest updates of nodes between from and
      to
  end
end
```

Table 4 Data fields of a client.

Field	Description
LatestBlock	Hash value of the latest block.
Blocks	Map that maps the hash values of parent blocks to hash values of received blocks.
Unused	Map that maps the list of unused TXOs in that block to the hash value of the latest block of each fork.
Used	Map that maps the TXOs to the hash values of blocks with used TXOs.
Updates	Map that maps the hash values of nodes to branchIDs and hash values of blocks.

5.2 Data Synchronization for a Client

Table 4 shows the data fields held by a client. The client keeps the hash values of blocks received, its own used TXOs, its own unused TXOs, and the update history of the Merkle proofs of the TXOs. When a new block is generated, client *c* receives the new block *b*, the used TXOs *usedTXOs*, the newly added TXOs *newTXOs*, and the node hash values *nodeHashes*. When a client receives a new block, the client first downloads missing blocks and TXO tree updates from a full node. Then, the client updates its own data with *b*, *usedTXOs*, *newTXOs*, and *nodeHashes*.

6. Broadcast

Since transactions in Trail include the TXO Merkle proofs, the data size of a transaction is large and wastes network resources. This section describes a technique for reducing the data size of a transaction using the characteristics of the TXO tree. Furthermore, during block propagation, the data size to be broadcast can be reduced using the existing blockchain method. Finally, we describe the process through which a client obtains new blocks, TXOs, and TXO tree updates.

6.1 Transaction

A transaction includes the TXOs and Merkle proofs in Inputs. For each TXO, the size of the transaction increases by $128 + 32 \times 255 = 8288$ bytes. Therefore, Trail attempts to reduce the number of TXOs in Inputs by determining the transaction fee in accordance with the number of TXOs in Inputs. Thus, to reduce the Inputs, a client will prefer to manage its balance with only a small number of TXOs instead of keeping the balance in a large number of small TXOs. If a client generates only one transaction per block, then the client needs to have only one unused TXO.

Furthermore, multiple Merkle proofs included in a transaction can be combined to reduce the data size of the transaction. Let the gap between the maximum and minimum indexes of the input TXOs in a transaction be δ. As shown in Fig. 4, all nodes that are higher than a height $\log_2 \delta$ with respect to the Merkle proofs in Inputs are the same.

Moreover, let the gap between the RightmostIndex of the latest block and the minimum index of the input TXOs in a transaction be Δ. Then, nodes at a height higher than $\log_2 \Delta$ with respect to the Merkle proofs in Inputs are the same as nodes at a height higher than $\log_2 \Delta$ with respect to the RightmostProof of the latest block. Therefore, the Merkle proofs higher than $\log_2 \Delta$ can be removed from the transaction by adding a flag to the transaction.

If a client uses a TXO within at least b blocks and every block approves n TXOs, then $\Delta < nb$ and $log_2 \Delta < log_2(nb)$. In the case of $b = 100$ and $n = 10000$ txo/block, $\log_2(nb)$ is less than 20. Therefore, the increase in the transaction size per TXO in Inputs is $128 + 20 \times 32 = 768$ bytes.

The transaction size is then $32 + i \times 768 + o \times 128$ bytes, where the number of TXOs in Inputs is i and the number of TXOs in Outputs is o.

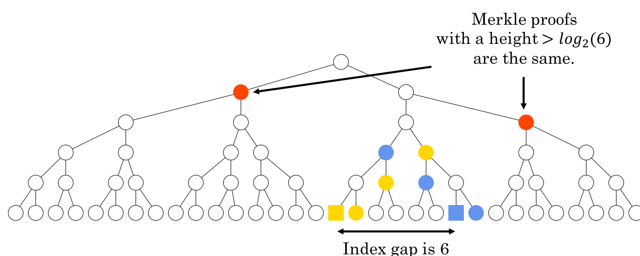

Fig. 4 Merkle proofs of TXOs referring to the same nodes.

6.2 Block

For block validation, the block proposer must broadcast the approved transactions with the new block; thus, the size of the data to be broadcast is substantial. If the total number of TXOs in Inputs approved by the block is 10000 and the total number of Outputs is 10000, the data size will be approximately $(960 + 128) \times 10000$ bytes ≈ 10 MB.

The data size can be reduced by omitting duplicate Merkle proofs, but it is expected that this problem can be solved using a protocol similar to compact block relay [16]. Compact block relay is a Bitcoin protocol in which only transaction IDs are included in the block broadcast data instead of sending entire transactions because the transactions are broadcast before the block is broadcast and other nodes in the network already have the transactions. By means of compact block relay, the transactions included in a block broadcast can be compressed to 8 bytes, so even if 10,000 transactions are approved, the data size is, at most, block size $+10000 \times 8 = 8288 + 80000$ bytes ≈ 90 kB.

6.3 New TXO, Used TXO, and TXO Tree Updates

When a block is generated, a client needs to receive the newly added TXOs, used TXOs, and TXO tree updates to update its own data. However, the client needs to update only its own TXOs and Merkle proofs; thus, the client does not need all these data.

In Trail, when a client receives a new block, the client sends a message to the node containing the hash value of the block, the address of the client, and the indexes of its own TXOs. The Trail node returns the newly added TXOs or used TXOs whose OwnerAddress matches the address in the message and the hash values of nodes with the branchID in the message.

7. Data Archiving

Trail assumes clients use mobile devices. Therefore, client devices do not store unnecessary data but rather archive the data for external storage, such as in the cloud or on an SSD.

Specifically, only the Merkle proofs for unused TXOs in the latest block are stored on a client device, and the hash values of other nodes are archived. Instead of *c.Updates*, the update data on the device of client c are represented by *c.Memory*, and the archived update data are represented by *c.Archive*. Algorithm 3 illustrates the archiving of unnecessary data to *c.Archive*.

Furthermore, the client archives old updates of the TXO tree. Let h_{latest} be the block height of the latest block, and let h_{del} be the threshold for archiving. The client archives the updates of the TXO tree in blocks with a block height of less than $h_{latest} - h_{del}$.

Let u be the number of unused TXOs for the latest block of each fork, and let f be the number of forks when

Algorithm 3: Update of the update history with archiving.

```
Input: client c, block b and map updatedHashes that
       maps the hash values of updated nodes in TXO
       tree of b to branchIDs
begin
  newMemory ← HashMap
  for t in c.Unused[hash(b)] do
    index ← t.Index
    for h ← 0 to 254 do
      if index is even then
        index ← index + 1
      else
        index ← index − 1
      end
      i ← branchID(h, index)
      if c.Memory[i] exists then
        updates ← c.Memory[i]
        if updatedHashes[i] exists then
          newHash ← updatedHashes[i]
          updates[hash(b)] ← newHash
        end
      else if updatedHashes[i] exists then
        newHash ← nodeHashes[i]
        if c.Archive[i] exists then
          updates ← c.Archive[i]
          updates[hash(b)] ← newHash
        else
          updates ← HashMap
          updates[hash(b)] ← newHash
        end
      else
        updates ← c.Archive[i]
      end
      newMemory[i] ← updates
      index ← index ≫ 1
    end
  end
  forall the i ∈ c.Memory.keys do
    if newMemory[i] not exists then
      Store c.Memory[i] to c.Archive[i]
    end
  end
  c.Memory ← newMemory
end
```

generating h_{del} blocks. Assume that unused TXOs are used within b blocks, that n TXOs are added for each block, and that the Merkle proofs for all unused TXOs are updated in every block.

At this time, in each fork, the Merkle proofs for the unused TXOs with heights greater than $\log_2 bn$ are the same. Therefore, the data size of the TXO tree updates stored on the client device is $\min(h_{del}, b) \times 32f\,(u(\lfloor\log_2 bn\rfloor + 1) + 255 - (\lfloor\log_2 bn\rfloor + 1))$ bytes. Additionally, the data size of unused TXOs is $128uf$ bytes. In the case of $u = 2$, $h_{del} = 100$, $b = 100$, $f = 2$, and $n = 10^4$, the data size on the device is approximately 1.76 MB. This amount of data is sufficiently small to store on a mobile device.

On the other hand, the size of the archived data increases as the blockchain lengthens. However, the data size can be reduced by deleting archived data at block height h_{del}. The client considers that blocks whose height is lower than $h_{latest} - h_{del}$ are finalized and will not be overwritten and deletes the update history for those blocks. If a block is overturned by an attack, the client will need to obtain the deleted data from a full node. h_{del} is considered to be larger than h_{del}, b. The size of archived TXO tree updates is at most $32\frac{h_{del}f}{b}\left(ub(\lfloor\log_2 bn\rfloor+1)+\frac{h_{del}}{b}(255-(\lfloor\log_2 bn\rfloor+1))\right)$ bytes. The size of archived TXOs is $128(\frac{h_{del}}{b})^2uf$ bytes.

8. Smart Contracts

A number of blockchains provide mechanisms for executing programs and recording the execution results in the blockchain. These mechanisms and programs are called smart contracts. This section describes how to execute smart contracts based on Trail and record their results in the blockchain.

8.1 Distributed Key Generation

Trail uses threshold signatures to prove that validators obtained consensus for the execution results of smart contracts. Note that Trail uses threshold signatures only for smart contracts and only smart contract execution nodes (Sect. 8.4) use them. A distributed key generation (DKG) protocol is a protocol for generating the private and public keys of threshold signatures without a trusted party.

Feldman proposed a verifiable secret sharing (VSS) protocol [17]. In Feldman's VSS protocol, a trusted party shares a secret via Shamir's Secret Sharing scheme [18] and a verification commitment. Each participant in secret sharing can verify the secret share received from the trusted party using the verification commitment.

Pedersen [19] proposed a (t, n)-threshold DKG protocol in which every participant runs Feldman's VSS protocol. Trail utilizes it.

8.2 Extension

TXOs and transactions are extended to the deployment and execution of smart contracts. For this purpose, a TXO (Table 1) has an additional Data field. The Data field contains smart contract code or a hash value of the execution result of a smart contract. Similarly, a transaction (Table 2) has an additional Script field. The Script field contains messages that can be understood as function calls. If the Script field is empty, this means that the transaction corresponds to a payment and not to the execution of a contract.

8.3 Deploying Smart Contracts

To deploy a smart contract, a client generates a transaction that outputs a TXO with the smart contract code in the Data field. The address of the owner of the TXO that

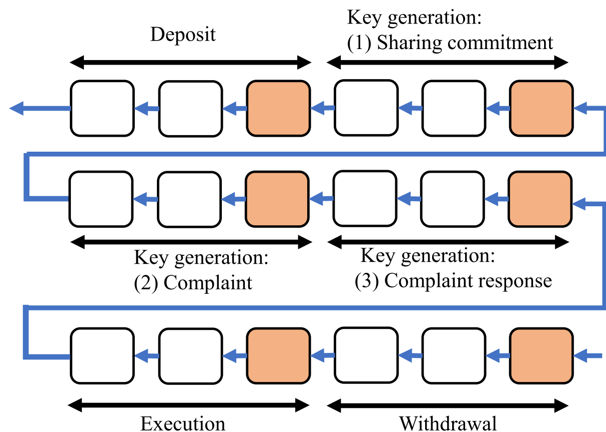

Fig. 5 Life cycle of a smart contract execution node (1 epoch = 3 blocks).

contains the smart contract code is the smart contract address. The smart contract address is hash(hash(*tx.Inputs*) || hash(*code*)), where *code* is the binary of the smart contract code. Since a TXO can be used only once as an input, the smart contract address is unique.

8.4 Smart Contract Execution Nodes

In Ethereum, the nodes that generate and verify the blockchain also execute smart contracts and maintain their states. On the other hand, in Trail, the set of nodes that manage the blockchain and the set of smart contract execution (SCE) nodes do not always match.

As shown in Fig. 5, the life cycle of a SCE node is divided into six phases: deposit, three phases for key generation, execution, and withdrawal. In turn, three phases for key generation are sharing commitment, complaint, and complaint response. We call the length of a phase *epoch, that is a system parameter and specified by the number of blocks.* SCE nodes can start smart contract execution every epoch and each phase takes place in a pipeline. Because of it, Trail allows six groups of SCE nodes to run in parallel at most, though a Trail-based blockchain can limit the number of the groups. Each phase is described below.

8.4.1 Deposit

Each participating user generates a deposit transaction that locks a certain deposit amount required to become an SCE node. The deposit will be forfeited if the user commits fraud. Furthermore, this deposit secures voting rights when obtaining consensus on the execution result of the smart contract.

8.4.2 Key Generation

In this phase, the SCE nodes generate the private and public keys of the $(\frac{n}{2}, n)$-threshold signature mechanism through the protocol described in Sect. 8.1. The key generation phase consists of three epochs.

In the first epoch, every SCE node chooses at random a polynomial $f_i(x)$ with a degree of at most $\frac{n}{2} - 1$, broadcasts the verification commitment and sends secret shares individually to other nodes. Each SCE node verifies the received secret shares and verification commitments. The blockchain records the verification commitments.

In the second epoch, if a node experiences failure in the verification of a secret share or verification commitment or did not received a secret share in the last epoch, the node broadcasts a complaint to the sender of that secret share.

In the third epoch, the senders of failed secret shares broadcast the correct secret shares. If all complaints concerning a verification commitment are resolved, that verification commitment is deemed valid. The group public key is generated from valid verification commitments. If the number of valid verification commitments is greater than $\frac{2n}{3}$, the SCE nodes can move to the execution phase. Otherwise, the nodes cannot move to the execution phase, and the deposit of the sender of each invalid verification commitment is forfeited.

8.4.3 Execution

The SCE nodes hold the smart contracts deployed in the blockchain and their TXOs and Merkle proofs. The SCE nodes are sorted by the hash values of the concatenation of the group public key and their own addresses. From the top of the sorted list, the leaders of the SCE nodes in each block of the epoch are selected. Subleaders are also selected to provide redundancy.

The clients send the execution transaction to the SCE nodes. The leader broadcasts the execution order of the transactions. Other nodes sign each transaction and send it back to the leader. When executing an execution transaction, an SCE node creates a TXO with the execution result recorded in the Data field and uses it as the output of the transaction.

When the leader has collected a number of signatures above the threshold, the leader creates the group signatures of the transactions and broadcasts them to the blockchain management nodes together with the executed transactions.

The execution process requires a single round-trip communication between the leader and other SCE nodes. It involves a broadcast to the nodes over the blockchain peer-to-peer network. Large latency can be a concern in case a blockchain network is very large. But even Bitcoin, that dozens of thousands of nodes participate, can broadcast a block just in about two second to 90 percentile of all the nodes [20].

8.4.4 Withdrawal

When the locking period of a deposit is exceeded, the corresponding node will be removed from among the SCE nodes.

9. Conclusion

We proposed the Trail architecture. In Trail, the assets of accounts are managed by updating one Merkle tree through the blockchain. The issuer of a transaction includes its own TXO and its Merkle proof in the transaction, which allows the nodes to validate and generate blocks without needing to hold the entire Merkle tree. As a result, the block size is only 8 kB and is constant regardless of the number of transactions.

We described techniques for reducing the sizes of the data broadcast to the network. The transaction data size can be reduced by omitting duplicate Merkle proofs. Furthermore, during block propagation, the data size is reduced using compact block relay.

We showed that by properly archiving and deleting data, the data stored on a client device can be reduced to approximately 1.76 MB, and the data to be archived can be reduced to 181 MB. Therefore, Trail can operate on mobile devices.

Finally, we proposed a method of executing smart contracts based on Trail. Nodes can prove that they have obtained consensus on the execution results of smart contracts by using threshold signatures and can record the execution results in the blockchain.

Trail significantly reduces the storage requirements for nodes and makes it easier for users to become block proposers or validators. As a result, Trail improves the decentralization and security of the blockchain.

Acknowledgments

Ryunosuke Nagayama and Kazuyuki Shudo were supported by SECOM Science and Technolgy Foundation. Kazuyuki Shudo was supported by JSPS KAKENHI Grant Number JP21H04872.

References

[1] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System." https://bitcoin.org/bitcoin.pdf, 2008.
[2] Bitcoin.org, "Bitcoin Core." https://github.com/bitcoin/bitcoin. Accessed on Sept. 26th, 2021.
[3] G. Wood, "Ethereum: a secure decentralised generalised transaction ledger." http://gavwood.com/paper.pdf, 2014.
[4] Ethereum, "Go-Ethereum." https://github.com/ethereum/go-ethereum. Accessed on Sept. 26th, 2021.
[5] Bitcoin.com, "Blockchain size." https://www.blockchain.com/charts/blocks-size. Accessed on Sept. 26th, 2021.
[6] R. Nagayama, R. Banno, and K. Shudo, "Trail: A Blockchain Architecture for Light Nodes," 25th IEEE Symposium on Computers and Communications (ISCC 2020), pp.1–7, 2020.
[7] L. Quan, Q. Huang, S. Zhang, and Z. Wang, "Downsampling Blockchain Algorithm," Proc. IEEE INFOCOM 2021 Workshops, pp.342–347, 2019.
[8] Ethereum, "Light client protocol." https://eth.wiki/en/concepts/light-client-protocol. Accessed on Sept. 26th, 2021.
[9] V. Buterin, "The Stateless Client Concept." https://ethresear.ch/t/the-stateless-client-concept/172. Accessed on Sept. 26th, 2021.
[10] T. Dryja, "Utreexo: A dynamic hash-based accumulator optimized for the Bitcoin UTXO set." https://eprint.iacr.org/2019/611. Accessed on Sept. 26th, 2021.
[11] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford, "OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding," 2018 IEEE Symposium on Security and Privacy (SP), 2018.
[12] V. Buterin, "Sharding FAQs." https://github.com/ethereum/wiki/wiki/Sharding-FAQs. Accessed on Sept. 26th, 2021.
[13] A. Back, M. Corallo, L. Dashjr, M. Friedenbach, G. Maxwell, A. Miller, A. Poelstra, J. Timon, and P. Wuille, "Enabling Blockchain Innovations with Pegged Sidechains," 2014.
[14] D. Leung, A. Suhl, Y. Gilad, and N. Zeldovich, "Vault: Fast Bootstrapping for the Algorand Cryptocurrency," NDSS, 2019.
[15] Blockchain.com, "Average block size." https://www.blockchain.com/charts/avg-block-size. Accessed on Sept. 26th, 2021.
[16] M. Corallo, "Compact Block Relay (BIP 152)." https://github.com/bitcoin/bips/blob/master/bip-0152.mediawiki. Accessed on Sept. 26th, 2021.
[17] P. Feldman, "A practical scheme for non-interactive verifiable secret sharing," 28th Annual Symposium on Foundations of Computer Science (SFCS 1987), pp.427–438, 1987.
[18] A. Shamir, "How to share a secret," Communication of the ACM, vol.21, no.11, pp.612–613, 1979.
[19] T.P. Pedersen, "Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing," Advances in Cryptology — CRYPTO '91, pp.129–140, 1992.
[20] T. Neudecker, Security and Anonymity Aspects of the Network Layer of Permissionless Blockchains, Ph.D. thesis, Karlsruher Instituts für Technologie, Nov. 2018.

Ryunosuke Nagayama received the B.E. and M.S. degrees from Tokyo Institute of Technology in 2019 and 2021, respectively. He is currently working with Yahoo Japan Corporation. His research interests include distributed systems and blockchain.

Ryohei Banno received the Bachelor of Engineering and Master of Information Science and Technology degrees from Hokkaido University, Japan, in 2010 and 2012 respectively, and the Ph.D. degree in science from Tokyo Institute of Technology, Japan, in 2018. From 2012 to 2018, he was a Researcher at NTT Network Innovation Laboratories. From 2018 to 2020, he was a Researcher at Tokyo Institute of Technology. Since 2020, he has been an Assistant Professor at Kogakuin University, Japan. His research interests include distributed systems and Internet of Things (IoT). Dr. Banno's awards and honors include the Outstanding Paper Award from Information Processing Society of Japan (IPSJ) in 2015, Inoue Research Award for Young Scientist from Inoue Foundation for Science in 2020, and Funai Research Award from Funai Foundation for Information Technology in 2020.

Kazuyuki Shudo received the B.E. degree in 1996, the M.E. degree in 1998, and the Ph.D. degree in 2001 all in computer science from Waseda University. He worked as a Research Associate at the same university from 1998 to 2001. He later served as a Research Scientist at National Institute of Advanced Industrial Science and Technology. In 2006, he joined Utagoe Inc. as a Director, Chief Technology Officer. Since December 2008, he currently serves as an Associate Professor at Tokyo Institute of Technology. His research interests include distributed computing, programming language systems and information security. Dr. Shudo has received the best paper award at SACSIS 2006, Information Processing Society Japan (IPSJ) best paper award in 2006, the Super Creator certification by Japanese Ministry of Economy Trade and Industry (METI) and Information Technology Promotion Agency (IPA) in 2007, IPSJ Yamashita SIG Research Award in 2008, Funai Prize for Science in 2010, The Young Scientists' Prize, The Commendation for Science and Technology by the Minister of Education, Culture, Sports, and Technology in 2012, and IPSJ Nagao Special Researcher Award in 2013. He is a member of IEEE, IEEE Computer Society, IEEE Communications Society and ACM.