EE 374: Blockchain Foundations **Stanford, Spring 2022**

Lecture 10: Accounts Model and Merkle Trees

April 29, 2022

Lecturer: Dr. Dionysis Zindros Scribe: Yifan Yang

1 Accounts Model

1.1 Accounts Model Compared with UTXO Model

Recall the previous UTXO model: we store a set of unspent transaction outputs (UTXOs). When a transaction occurs, UTXOs corresponding to the transaction's inputs are removed and UTXOs corresponding to the transaction's outputs are added into the UTXO set to produce a new UTXO set, as shown in Figure 1.

$$\begin{tikzcd}
\boxed{\text{UTXO Set}} \arrow[r, "\text{tx}"] & \boxed{\text{UTXO Set}} \arrow[r, "\text{tx}"] & \boxed{\text{UTXO Set}} \\
\text{State 0} & \text{State 1} & \text{State 2}
\end{tikzcd}$$

Figure 1: State transitions in the UTXO model

The accounts model is another model of transactions. In the accounts model, transactions contain 1) the account that sends balance (from), 2) the account that receives balance (to), 3) the value of the transaction (val), 4) the transaction fee (fee), and 5) the signature on the transaction (σ).

From	To	Val	Fee	σ

Figure 2: Structure of a transaction in the accounts model

For the accounts model, the state is maintained by accounts (public keys) and balances, as shown in Figure 3 and Figure 4.

$$\begin{tikzcd}
\boxed{\text{Balances}} \arrow[r, "\text{tx}"] & \boxed{\text{Balances}} \arrow[r, "\text{tx}"] & \boxed{\text{Balances}} \\
\text{State 0} & \text{State 1} & \text{State 2}
\end{tikzcd}$$

Figure 3: State transitions in the accounts model

The function that takes in a state and a transaction and returns a new state is called a **transition function**. It has a general form of:

$$\delta(st, tx) = \begin{cases} st' & \text{if } tx \text{ valid w.r.t. } st \\ \perp & \text{otherwise} \end{cases} \tag{1}$$

Balances State	
Alice	5 bu
Bob	100 bu
Dionysis	1 bu

Figure 4: Balance State

Specifically, the transition function of UTXO model is given by:

$$\delta_{UTXO}(st, tx) = \begin{cases} st \setminus tx_{in} \cup tx_{out} & \text{if } tx \text{ valid w.r.t. } st \\ \perp & \text{otherwise} \end{cases} \tag{2}$$

Where tx_{in} is the set of unspent outputs in the "inputs" field of tx, and tx_{out} is the set of newly generated outputs in the "outputs" field of tx.

The transaction validation process of UTXO model is

- Check σ
- Check conservation
- Check inputs are in st

Similarly, the transition function of the accounts model could be written as:

$$\delta_{acc}(st, tx) = \begin{cases} st' & \text{where } st'[tx.from] = st[tx.from] - tx.value, \\ & st'[tx.to] = st[tx.to] + tx.value, \text{ if } tx \text{ valid w.r.t. } st \\ \perp & \text{otherwise} \end{cases} \tag{3}$$

The transaction validation process of the accounts model is

- Check σ
- Check $st[tx.from] \geq tx.value$

1.2 Accounts Model Replay Attack

Here comes a problem. In the accounts model, if the same transaction is sent to the network twice, should the second transaction be included or not? For example, one morning, Bob bought a cup of coffee from Starbucks. The next morning, he bought a cup of coffee again. These two transactions have the same fields, even the signature.

If the network decides to accept transactions that are the same, the following replay attack could happen: an adversarial coffee shop could replay the transaction even if Bob didn't buy a coffee. However, if the network decided not to include transactions that are same, then Bob could only buy a coffee once.

The solution is to add a nonce field to transactions. The nonce is an 256-bit integer per source account which is incremented every new transaction. The transaction structure now looks like Figure 5.

From	To	Val	Fee	**nonce**	σ

Figure 5: Structure of tx in accounts model

And therefore, while validating transactions, an additional step of validating the nonce should be included. Transactions in which the nonce has already been used is rejected. This means that the state contains the current nonce for each account, in addition to the balance. The state transition function must also update the nonce for the “from” account of the transaction.

A side by side comparison between the two models of transactions is shown in Figure 6.

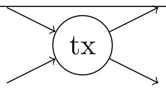
	UTXO	Accounts
Real System	Bitcoin	Ethereum
Transaction tx		From | To | Val | Fee | **nonce** | σ
Transistion δ	Remove consumed outputs and add produced outputs	Update balances $st'[from] := st[from] - value$ $st'[to] := st[to] + value$
Validation	Signature, Law of Conservation, Inputs exist in st.	Signature, Sufficient balance, Nonce unique.
Genesis State	$\varnothing$	$\{\}$

Figure 6: Side by Side Comparison of Two Models

2 State Machine Replication

We talk briefly about State Machine Replication (SMR). A state machine consists of a state, inputs and a transition function. The machine has an initial state. Based on its inputs and the state transition function, the machine updates its state. In SMR, multiple nodes in the network run a state machine in a distributed manner. The term “replication” signifies that each node in the network maintains the state of the machine and runs its transition functions locally. The goal of SMR is that each node runs the same set of state transitions and in the same order so that there is agreement or consensus on the state of the machine.

A blockchain can be considered as a distributed replicated database. A blockchain can help us run SMR. We have seen two examples of state machines that the blockchain can run — the accounts model and the UTXO model. In both cases, there is a state st, state transition functions δ, and inputs (which are transactions in this case). The initial state is specified by the genesis state.

3 Light Clients

How to run a blockchain node efficiently? Efficiency has multiple dimensions: storage, communication, and computation. For most application scenarios, the blockchain node has limited resources.

For example, if we store all the data of the chain, it would take gigabytes of storage. Validating every transaction in the network would be very heavy work for a phone. Therefore, a light client is needed for these resource-limited nodes.

3.1 Storage Efficiency: Merkle Trees

For a light client, it is better to save the data at a server and retrieve data at usage. However, we need to prove the integrity of the retrieved data. Hash functions are useful in this case. Suppose that we wanted to store a file on a server and verify that we receive the correct file from the server. We could hash the file and store the hash (checksum) locally. When we request files from the data server, we validate the checksum of the retrieved file to verify that it is the exact file we saved on the server. However, this requires clients to retrieve the entire file to validate its integrity even if only a 1 kilobyte chunk is needed.

We can also split the file into chunks and hash each chunk. This reduces the communication complexity: clients only need the chunk to be transferred. However, this requires more hash key storage for the client. The client needs to store one hash per chunk, making the storage complexity linear in the size of the file. There is a trade off between communication complexity and storage complexity: with large chunks, comes high communication complexity and with small chunks, comes high storage complexity.

Our goal is to achieve low storage and low communication. Specifically, storage with $O(1)$ complexity and communication with $O(\log n)$ complexity where n is the number of chunks of the file. And this is done with a data structure called Merkle tree.

3.2 Data Structure: Merkle Tree

Files are split into n data chunks.

$$D : D[0], D[1], ..., D[n-1]$$

A binary tree of depth μ is created, where there are $2^\mu = n$ leaves (for simplicity, assume that n is a power of 2). Each node in the binary tree stores a hash h which is the hash of its children concatenated.

$$h \coloneqq H(h[\text{left}] \parallel h[\text{right}])$$

Nodes on the leaves store the hash of the corresponding data chunk. The client stores the Merkle tree root (MTR) h_ϵ. When a data chunk is requested, the server sends the data chunk, along with every sibling hash value to the clients as shown in Figure 7. For example, when data chunk at index j is requested, the server sends $D[j]$, π_0, π_1, π_2, and π_3 to the client. The client walks from the received data chunk all the way up to the root to check if the hash values are intact. From calculating e_0 by hashing the data chunk, to the top level $e_{\mu+1}$, the client calculates $e_k = H(e_{k-1} \parallel \pi_{k-1})$ or $e_k = H(\pi_{k-1} \parallel e_{k-1})$(left child first). In this example, the client computes the values $e_0 = H(D[j])$, $e_1 = H(e_0 \parallel \pi_0)$, $e_2 = H(\pi_1 \parallel e_1)$, $e_3 = H(e_2 \parallel \pi_2)$, and $e_4 = H(\pi_3 \parallel e_3)$ and then compares e_4 with h_ϵ.

With this data structure, the data transferred is a list of π values and the data chunk of fixed size, which gives $|\pi| = O(\log n)$ succinct communication and $O(1)$ constant storage.

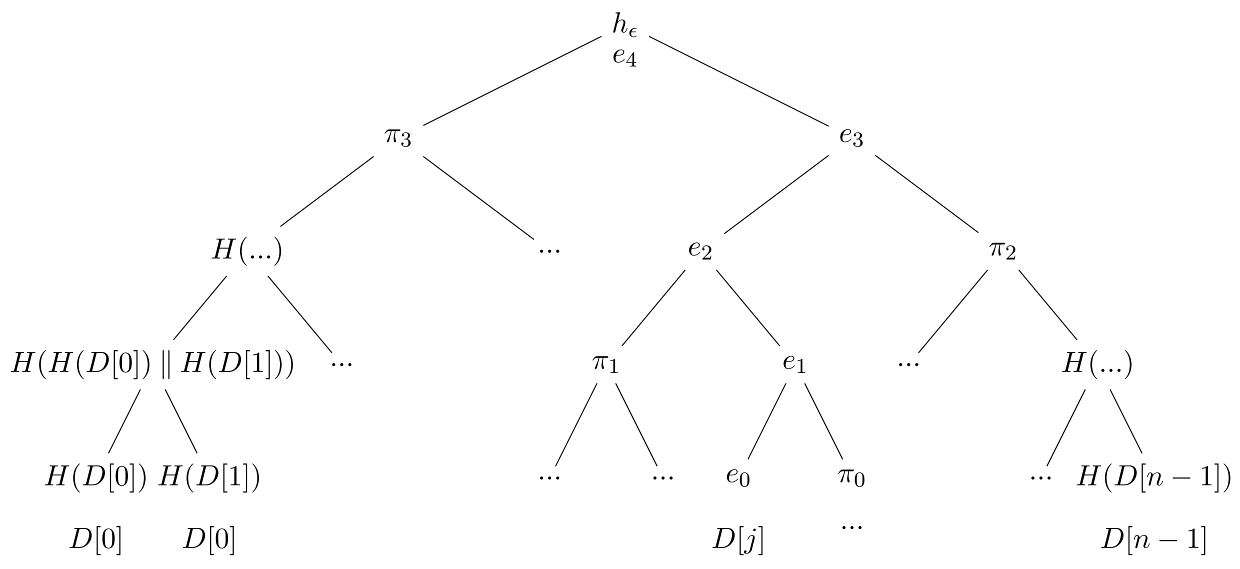


Figure 7: Merkle Tree

The Merkle tree structure is described by the functions

$$\text{compress}(D) \to h_\epsilon, \tag{4}$$

$$\text{prove}(D, j) \to \pi, \text{ and} \tag{5}$$

$$\text{verify}(h_\epsilon, d, j, \pi) \to \begin{cases} \text{true} & \text{if valid} \\ \text{false} & \text{otherwise} \end{cases}. \tag{6}$$

The correctness of the Merkle tree is specified as:

$$\forall D, \forall j, \text{verify}(\text{compress}(D), D[j], j, \text{prove}(D, j)) = \text{true} \tag{7}$$

3.3 Security of Merkle Trees

MT-security means that if the client outputs true after verifying the received data chunk and proof, then the received data must be the same data that was originally stored. To define security of Merkle trees formally, we create the following game that lets an adversary try to break the protocol.

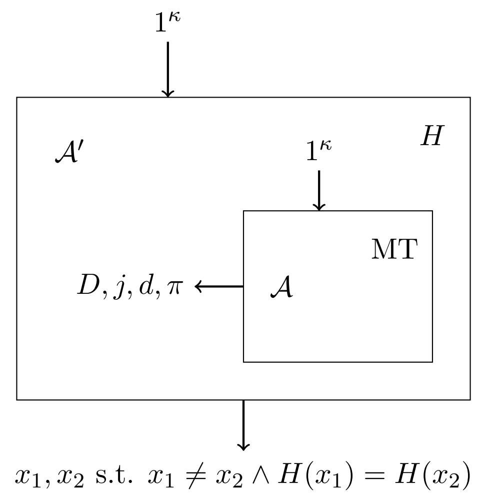
```
$\text{MERKLE}_\mathcal{A}(\kappa):$
    $D, \pi, j, d \leftarrow \mathcal{A}(1^\kappa)$
    return $\text{verify}(\text{compress}(D), d, j, \pi) \wedge d \neq D[j]$
```

Our goal is to prove that

$$\forall \text{ PPT } \mathcal{A} : Pr[\text{MERKLE}_\mathcal{A}(\kappa) = 1] \leq negl(\kappa)$$

Theorem 3.1. *Let H be a collision-resistant hash function. Then Merkle trees constructed with H are MT-secure.*

Proof. Suppose for contradiction, $\mathcal{A}$ breaks MT-security. We will construct an adversary $\mathcal{A}'$ that breaks collision-resistance of H.



We use e for the hash value calculated by the client, h for the expected hash value in the correct Merkle tree, and π for the hash values returned by the server.

Consider the event that $\mathcal{A}$ succeeds, i.e. verify(compress(D), $d, j, \pi) = 1 \wedge d \neq D[j]$.

$\mathcal{A}'$ works as follows:

Given that $\mathcal{A}$ succeeds, the returned list of π is used to calculate hashes of the nodes to verify the returned data chunk, which involves calculating e values by concatenating the children of the nodes. The hash of the root is the same ($h_\epsilon = e_{top}$) and the hash of the data chunk is different ($e_0 \neq h_0$). (If not, $\mathcal{A}'$ has already found a collision because different data chinks have the same hash.) Therefore, there must exists a node, some level k in the tree, such that $e_k = h_k$ but its children e_{k-1} or π_{k-1} not equal to the expected h values. (These must exist because roots are the same, but leaves are different). In this case, we have two different inputs that hash to the same value.

Then, adversary $\mathcal{A}'$ returns children of e_k from the verifier tree at level k as x_1 and children of h_k from real tree at the corresponding position as x_2. Then, x_1 and x_2 satisfy $x_1 \neq x_2 \wedge H(x_1) = H(x_2)$.

Therefore, the probability of breaking the Merkle tree protocol is the same as the probability of breaking the collision-resistant hash function H.

$$Pr[\text{MERKLE}_{\mathcal{A}}(\kappa) = 1] = Pr[\text{Collision}_{\mathcal{A}'}(\kappa) = 1]$$

But $Pr[\text{Collision}_{\mathcal{A}'}(\kappa) = 1]$ is negligible by assumption, which means the probability of breaking Merkle tree protocol is also negligible.

□