Network Security

Professor Gaurav S. Kasbekar

Department of Electrical Engineering

Indian Institute of Technology, Bombay

Week - 11

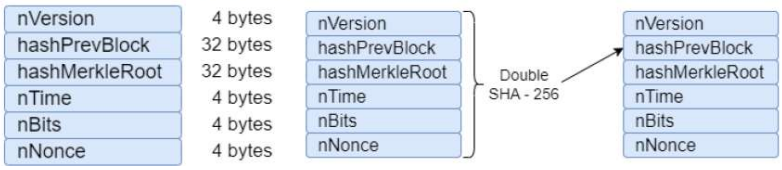
Lecture - 62

The Bitcoin Cryptocurrency: Part 2

Hello, in the previous lecture, we started our discussion of the Bitcoin cryptocurrency. We now continue that discussion. We discussed last time that each block includes a block header and a list of transactions. We now discuss the structure of the block header. So, each block in the blockchain begins with an 80-byte block header, which is shown in this figure.

The different fields of a block header are nVersion, then hashPrevBlock, hashMerkleRoot, nTime, nBits, and nNonce. We'll discuss these different fields in this lecture and the next few lectures. The sizes of the different fields of the header are shown here. So, nVersion is 4 bytes, hashPrevBlock is 32 bytes, and so on and so forth. So, this prefix 'n' of the 4-byte fields—namely nVersion, nTime, nBits, and nNonce— this prefix 'n' indicates that they are integer variables.

- Each block in the blockchain begins with a 80-byte block header (see fig.)
- The prefix "n" of the 4-byte fields indicates that they are integer variables
- The prefix "hash" of the 32-byte fields indicates that they store hash function outputs
- "nVersion" field specifies version of block
  - as Bitcoin system evolved, changes were proposed to fix bugs or add new features
- "hashPrevBlock" field in block header contains double SHA-256 hash of the block header of the previous block in the blockchain
- Since the genesis block has no previous block, its "hashPrevBlock" field was set to all zeros



Similarly, the prefix 'hash' of the 32-byte fields—namely hashPrevBlock and hashMerkleRoot— this prefix indicates that they store hash function outputs. The nVersion

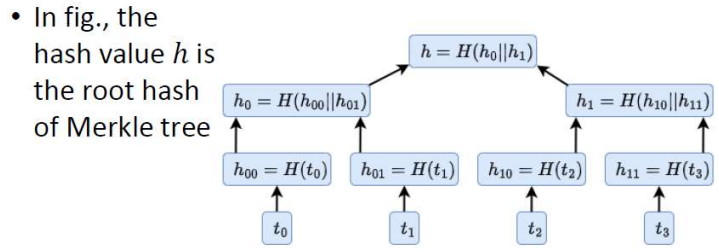
field specifies the version of the block. We have discussed that there are different versions of the Bitcoin system. So, this nVersion stores the particular version of this block. As the Bitcoin system evolved, changes were proposed to fix bugs or to add new features.

So, the version kept on changing each time. 'hashPrevBlock' contains a double SHA-256 hash of the block header of the previous block in the blockchain. That is shown over here. So, this is the block header of a particular block, and the double SHA-256 hash of this header is computed, and that is stored in this hashPrevBlock of the next block in the blockchain. Recall that the blockchain is a linked list of several blocks.

So, for a particular block, we find a double SHA-256 hash of the block header and store the value in the hashPrevBlock field of the next block. By double SHA-256 hash, we mean the following. Let H denote SHA-256 hash. In that case, to find the double SHA-256 hash of a particular value x, we find H(H(x)). So, that is, it's another cryptographic hash function.

So, we'll see that throughout the Bitcoin, as the cryptographic hash function, it uses double SHA-256. It's a measure to increase the randomness even further than computing the hash just once. Since the Genesis block has no previous block, its hashPrevBlock was set to all zeros. Recall that Genesis block is the very first block, and its hashPrevBlock field was set to zeros because there was no block before it. So, next we discuss the hashMerkleRoot field that is this one.

So, that is also cryptographic hash value. 'hashMerkleRoot' field stores the root hash of a Merkle tree that is formed using the transactions in the block. So, we'll discuss the concept of a Merkle tree. So, this shows a Merkle tree. The transactions in the block are these: t_0, t_1, t_2, and t_3.



So, each of these transactions encodes the transfer of a certain number of Bitcoins from a sender to a receiver. For example, t_1 may record that Alice transfers one Bitcoin to Bob.

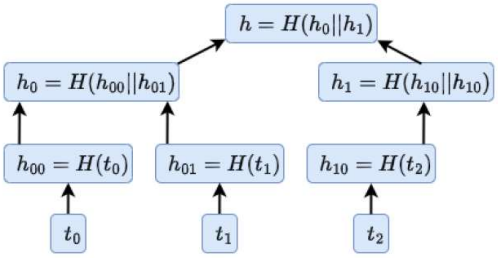
So, each of these is the transactions. We find the hash value of each of these transactions. So, h_{00} is the $H(t_0)$.

Then h_{01} is the $H(t_1)$. h_{10} is the $H(t_2)$ and h_{11} is the $H(t_3)$. Then we concatenate these two hashes, h_{00} and h_{01}, and find the hash of the resulting value. So, $h_0 = H(h_{00} \parallel h_{01})$.

And similarly, $h_1 = H(h_{10} \parallel h_{11})$. And then we concatenate these two values, h_0 and h_1, and the resulting value is h. So, these transactions are arranged so that they are the leaves of a binary tree. And then we keep computing the values of the intermediate nodes of the binary tree. So, the transactions are arranged as a list, as shown here, and the double SHA-256 hash of each is computed. The hash function H here denotes a double SHA-256 hash.

So, each H here is a double SHA-256 hash. Using these hashes as leaves, a binary tree is created where each node is associated with a double SHA-256 hash of the concatenation of its child hashes. For example, we concatenate h_0 and h_1, and the hash of that—that is, the double hash of that—is the hash of the parent of these two nodes. In this figure, the hash value h is the root hash of the Merkle tree. For a particular block, consider the transactions in the block; these are all placed such that they are the leaves of a binary tree. Then we keep on computing hashes in this manner, and when we get to the root of the tree, the hash value that we have computed for the root of this tree, that is, the root hash of the Merkle tree, and that is the hashMerkleRoot field stored in the block header.

- When the number of transactions is not a power of two, some nodes in Merkle tree have only one child
- In this case the hash of the single child is concatenated with itself and then hashed to derive the hash value associated with parent
- Fig. shows case where there are three transactions



So, in the previous slide, we considered the case where the number of transactions in the block is a power of 2. So, it is of the form 2 to the power n for some integer n, but that may not always be the case. So, when the number of transactions is not a power of 2, some nodes in the Merkle tree have only one child. So, we see that in that case, this binary tree

is not complete. So, some of the nodes in this binary tree will have only one child. An example is shown here.

In this case, the number of transactions is three, so it's not a power of 2. So, the Merkle tree is shown here. These two transactions are concatenated, and the hash is computed and it is stored in this node. But what about this transaction? There isn't another transaction that can be concatenated with it.

So, in that case, the hash of the single child, such as this t_2, is concatenated with itself and then hashed to derive the hash value associated with the parent. So, we can see here that h_{10} is concatenated with itself, and we find out the hash of the concatenation of h_{10} with itself, and that is the value h_1, which is stored corresponding to this node in the Merkle tree. So, this figure shows the case where there are three transactions. This is an example where the number of transactions is not a power of 2. So, what is the reason for using the hashMerkleRoot field?

So, that is used for tamper resistance. If any one of the transactions in the block is modified, then once we compute the hashMerkleRoot field, it will not match the actual value. So, any change to a transaction in a block will result in a change in the hashMerkleRoot field. So, that prevents someone from modifying one of the transactions in the block. A change in the hashMerkleRoot field, in turn, results in a change in the double SHA-256 hash of the block header.

Recall that the hashPrevBlock field of a block header contains the double SHA-256 hash of the block header of the previous block. We mentioned this during our discussion of hashPrevBlock. Hence, the hashPrevBlock field of a block depends on all the transactions in all the previous blocks all the way up to the genesis block. So, if any one of the transactions in the blockchain is changed, then the hashPrevBlock field of all subsequent blocks will change. So, that is the reason the blockchain has the property of tamper resistance.

So, we'll later see that this property is used for guaranteeing tamper resistance of the transaction data. So, if we change any of the transactions in a block, then the hashMerkleRoot field will change, and hence, the hashPrevBlock fields of the next block as well as all following blocks will change. So, why do we use a Merkle tree of the transactions for achieving tamper resistance? So, here is an alternative (simpler field) that also achieves tamper resistance. So, an alternative (simpler) field that could be included in the block header in place of the hashMerkleRoot field is the following.

We could have just hashed the concatenation of all the transactions in a block and put the resulting hash value in the block header. For example, if the transactions in a block are t_0, t_1, up to t_{n-1}, then we could have just concatenated them to get t_0, t_1, and so on up to t_{n-1} and found out the hash value of the result and stored that in the block header. So, this would also have guaranteed tamper resistance of the transaction data. So, then why do we use such a complicated structure like Merkle tree? So, Merkle tree achieves tamper resistance, but it also achieves one more important property.

- we could have hashed the concatenation of all the transactions in a block and put the resulting hash value in the block header
- i.e., if the transactions in a block are $t_0, t_1, \ldots, t_{n-1}$, then we could have included the hash value $H(t_0||t_1||\ldots||t_{n-1})$ in the block header
- this would also have guaranteed tamper resistance of the transaction data

The reason for using the Merkle tree is the following. It enables efficient membership proofs of transactions within a block. So, we'll discuss what this means. There are some nodes in the Bitcoin network called simple payment verification nodes. Earlier, we discussed that there are different kinds of nodes in a Bitcoin network.

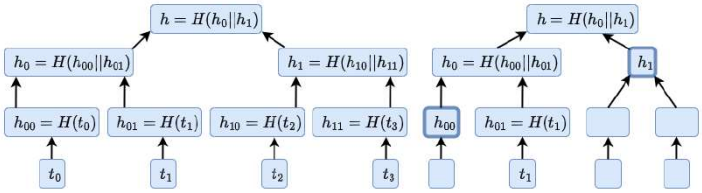
One kind is full nodes, which store the entire block. Another kind of node is simple payment verification (SPV) nodes, which store only the block headers and not the full blocks like full nodes. Suppose an SPV node, Bob, needs to verify whether a transaction, say t_1, involving Bob, belongs to a block or not. For example, the transaction may say that Alice transferred x bitcoins to Bob. Hence, Bob is concerned about whether the transaction t_1 is there in a block or not.

Bob has the header of the block but not the full block. So, how does Bob efficiently verify whether the transaction t_1 belongs to a particular block or not? Bob contacts a full node, and the full node must efficiently convince Bob that the transaction t_1 indeed belongs to the block. By efficiently, we mean that the full node must just send only a small amount of data to Bob, but by examining this small amount of data, Bob must still be convinced that the transaction t_1 indeed belongs to the block. How can the full node transfer such a small amount of data to Bob to convince him that the transaction t_1 belongs to the block?

So, the full node sends a so-called Merkle branch to Bob to prove the existence of transaction t_1 in the block. So, we'll discuss what is meant by a Merkle branch. It consists of certain hash values in a Merkle tree. So, here's an example. Suppose we want to prove that transaction t_1 was involved in the calculation of the root hash 'h' in the figure on the left.

So, ‘h’ is the root hash of the Merkle tree, and we want to prove that the transaction t_1 was involved in the calculation of this root hash ‘h’. Recall that the root hash of the Merkle tree is stored in the block header. So, an SPV node such as Bob will have the block header and hence has the hash h. But we need to convince Bob that the transaction t_1 was involved in the calculation of this root hash ‘h’. So, what information must a full node transfer to Bob to convince him that t_1 is involved in the calculation of this root hash h? We only need to provide the Merkle branch consisting of these hashes, which are highlighted in the figure on the right, namely h_{00} and h_1. Now Bob knows the value of t_1, and Bob has been provided this Merkle branch, which consists of the hashes h_{00} and h_1. Now let’s see what Bob can do.

- Suppose we want to prove that the transaction t_1 was involved in the calculation of the root hash h in fig. on left
- We only need to provide the Merkle branch consisting of the hashes:
  - h_{00} and h_1
- The hashes h_{01}, h_0 and h can be calculated from t_1 and the Merkle branch
- If the root hash h appears in the hashMerkleRoot field of a block header, then we can be certain that this block contains the transaction t_1



Bob can compute the $H(t_1)$, that is h_{01}, and Bob has been provided h_{00}. So, he can concatenate h_{00} and h_{01} to get h_0. So, Bob can calculate h_0, and Bob has been provided h_1. So, he can concatenate h_0 and h_1 and get h. So, that tells Bob that this h can be generated in this manner using t_1 as one of the inputs. So, this way Bob is convinced that t_1 is indeed involved in the computation of this hash h.

So, the hashes h_{01}, which is here, then h_0, which is this one, and h can be calculated from t_1 and the Merkle branch consisting of h_{00} and h_1, as we just discussed. So, if the root hash h appears in the hashMerkleRoot field of a block header, which Bob has, then we can be certain that this block contains the transaction t_1. This is an example of efficient membership proof of transactions within a block. So, in our example, the full node presented an efficient membership proof of the transaction t_1 within a particular block, whose block header was present with Bob, but Bob did not have the full block with him.

In general, the Merkle branch required to prove the existence of a transaction in a block containing n transactions has size $O(\log_2 n)$.

- In general, the Merkle branch required to prove the existence of a transaction in a block containing n transactions has size:
  - $O(\log_2 n)$
- In contrast, if $H(t_0||t_1||\ldots||t_{n-1})$ were included in block header instead of the hashMerkleRoot field, then what data would be required to prove the existence of a transaction in the block?
  - all the transactions $t_0, \ldots, t_{n-1}$ would have to be specified
- Hence required size would be:
  - $O(n)$

This arises from the fact that if we have binary tree with n leaves, in that case the height of the tree is order log(n). From this fact, you can easily verify that the Merkle branch required to prove the existence of a transaction in a block has size O(log(n)). So, that's left as a simple exercise. In contrast, if the other scheme that we discussed, that is just concatenating all the transactions and computing the hash value, if $H(t_0 \parallel t_1 \parallel \cdots \parallel t_{n-1})$, that is the concatenation of all these, were included in the block header instead of the hashMerkleRoot field, then what data would be required to prove the existence of a transaction in the block? Now, $H(t_0 \parallel t_1 \parallel \cdots \parallel t_{n-1})$ is included in the block header, and an SPV node, Bob, has only t_1. So, what other information should we provide to Bob to convince him that t_1 is included in the data that is used to compute this hash?

So, we have to present all the transactions t_0, t_2 up to t_{n-1}. So, all these transactions would have to be specified. So, then Bob can put them together along with t_1 and verify that the hash is indeed the value that is there in the block header. So, hence, the required size would be the size of the list of all transactions, and that is O(n) since there are n transactions. So, log n is much smaller than n; hence, when a Merkle tree is used, the amount of information that needs to be provided to an SPV node is much lower than in the case where the hash of the concatenation of all the transactions were stored in the block header.

So, we have discussed this property that a Merkle tree enables efficient membership proofs of transactions within a block. So, that's the reason for using a Merkle tree as opposed to just concatenating all the transactions and putting the hash in the block header. So, in summary, we have discussed various features of Bitcoin. In particular, we focused on the

block header and discussed the various fields that are present in the block header. Some of them are integer fields.

Others are cryptographic hash values. We discussed hashPrevBlock and hashMerkleRoot. We discussed the use of a Merkle tree for membership proofs. We will continue our discussion of Bitcoin in the next lecture. Thank you.