16. Bitcoin

16. Bitcoin

16.1. Problem Statement

Bitcoin is a digital cryptocurrency, which means it should have all the same properties as physical currency (e.g. the United States dollar). In our simplified model, a functioning currency should have the following properties:

- Each person has a bank account, in which they can store units of currency they own.
- Alice cannot impersonate Bob and perform actions as Bob.
- Any two people can engage in a *transaction*. Alice can send Bob n units of currency. This will cause Alice's bank account balance to decrease by n units, and Bob's bank account to increase by n units.
- If Alice has n units of currency in her account, she cannot spend any more than n units in any transaction.

In traditional physical currency, these properties are enforced by a trusted, centralized party such as a bank. Everyone trusts the bank to keep an accurate list of account holders with their appropriate account balances, and ensure that the identity of each user is correct before proceeding with a transaction. So, if Alice sends n units to Bob, both Alice and Bob trust that the bank will correctly decrease Alice's balance by n and increase Bob's balance by n. Everyone also trusts that the bank will not let Alice spend $n + 1$ units of currency if she only has n units in her account.

The goal of Bitcoin is to replicate these basic properties of a functioning currency system, but without any centralized party. Instead of relying on a trusted entity, Bitcoin uses cryptography to enforce the basic properties of currency.

16.2. Cryptographic Primitives

Bitcoin uses two cryptographic primitives that you have already seen in this class. Let's briefly review their definitions and relevant properties.

A *cryptographic hash* is a function H that maps arbitrary-length input x to a fixed-length output $H(x)$. The hash is collision-resistant, which means it is infeasible to find two different inputs that map to the same output. In math, it is infeasible to find $x \neq y$ such that $H(x) = H(y)$.

A *digital signature* is a cryptographic scheme that guarantees authenticity on a message. Alice generates a public verification key PK and a secret signing key SK. She broadcasts the public key to the world and keeps the secret key to herself. When Alice writes a message, she uses the secret key to generate a signature on her message and attaches the signature to the message. Anyone else can now use the public key to verify that the signature is valid, proving that the message was written by Alice and nobody tampered with it.

With these two cryptographic primitives in mind, we can now start designing Bitcoin.

16.3. Identities

Since there is no centralized party to keep track of everyone's accounts, we will need to assign a unique identity to everyone. We also need to prevent malicious users from pretending to be other users.

Every user of Bitcoin generates a public key and private key. Their identity is the public key. For example, Bob generates PK_B and SK_B and publishes PK_B to the world, so now his identity in Bitcoin is PK_B. When Bob is interacting with Bitcoin, he can prove that he is the user corresponding to PK_B by creating a message and signing it with SK_B. Then anybody can use PK_B to verify his signature and confirm that he is indeed the PK_B user. Because digital signatures are unforgeable, an attacker who doesn't know Bob's secret signing key will be unable to impersonate Bob, because the attacker cannot generate a signature that validates with PK_B.

16.4. Transactions

Without a centralized party to validate transactions, we will need a way to cryptographically verify that Alice actually wants to send n units of currency to Bob. Fortunately, this problem is essentially solved with our identity scheme above. If Alice wants to send n units of currency to Bob, she can create a message "PK_A sends n units of currency to PK_B" and sign it with her secret key. Note how she uses her public key PK_A as her identity and Bob's public key PK_B as his identity. Now anybody can verify the signature with Alice's public key to confirm that the user PK_A did intend to make this transaction. Bitcoin doesn't validate the recipient–if someone wanted to refuse a transaction, they could create another transaction to send the money back.

16.5. Balances

In our transaction scheme so far, nothing is stopping Alice from creating and signing a message "PK_A sends $100n$ units of currency to PK_B," even though she may only have n units of currency to spend. We need some way to keep track of each user's balances.

For now, assume that there is a *trusted ledger*. A ledger is a written record that everybody can view. It is append-only and immutable, which means you can only add new entries to the ledger, and you cannot change existing entries in the ledger. You can think of the ledger like a guest book: when you visit, you can add your own entry, and you can view existing entries, but you cannot (or should not) change other people's old entries. Later we will see how to build a decentralized ledger using cryptography.

Bitcoin does not explicitly record the balance of every user. Instead, every completed transaction (along with its signature) is recorded in the public ledger. Since everyone can view the ledger, anybody can identify an invalid transaction, such as Alice trying to spend more than she has. For example, suppose Bob starts with \$10 and everyone else starts with \$0. (We will discuss where Bob got the \$10 later.) Consider the following ledger:

- PK_B (Bob) sends PK_A (Alice) \$5. Message signed with SK_B.
- PK_B (Bob) sends PK_M (Mallory) \$2. Message signed with SK_B.
- PK_M (Mallory) sends PK_A (Alice) \$1. Message signed with SK_M.
- PK_A (Alice) sends PK_E (Eve) \$9. Message signed with SK_A.

Can you spot the invalid transaction? Although we don't have the balances of each user, the transaction ledger gives us enough information to deduce every user's balance at any given time. In this example, after the first three transactions, Bob has \$3, Mallory has \$1, and Alice has \$6. In the fourth transaction, Alice is trying to spend \$9 when she only has \$6, so we know it must be an invalid transaction. Because the ledger is trusted, it will reject this invalid transaction.

Thus, the idea is to have each block have a list of the transactions that show where the money being used in this transaction came from, which also means that blocks have to be sorted in order of creation. Now, our ledger looks as follows (again assuming that Bob starts with $10B$ and everyone else starts with $0B$:

- $TX_1 = PK_B$ (Bob) sends PK_A (Alice) $5B$, and the money came from the initial budget. TX_1 signed with SK_B
- $TX_2 = PK_A$ (Alice) sends PK_E (Eve) $5B$, and the money came from TX_1. TX_2 signed with SK_A

So, to check a transaction, we follow three steps:

1. Check that the signature on the transaction is verified using the PK of the sender
2. Check that the sender in this transaction was the receiver in some previous transaction
3. Check that the sender in this transaction has not spent the money in some previous transaction (aka they have enough money left over)

If we were checking TX_2, we first check that TX_2 was actually signed by Alice. Then, we check that Alice received some money in the past by checking the previous transactions. In TX_2, we see that Alice received the money from TX_1, and checking TX_1 verifies that Alice was the receiver. Next, we check that Alice has not spent the money earlier, so we scan the history of the blockchain and we don't see anywhere where the money from TX_1 was used. Finally, we check that Alice has 5 B by again checking TX_1 and seeing that she did receive 5 B from Bob. At this point, we have verified that TX_2 is a valid transaction, and we thus append it to the blockchain ledger.

At this point, we have created a functioning currency:

- Each person has a unique account, uniquely identified by public key.
- Users cannot impersonate other users, because each user can be validated by a secret signing key that only that user knows.
- Users can engage in a transaction by having the sender add their transaction to the ledger, with a signature on the transaction.
- Users cannot spend more than their current balance, because the trusted ledger is append-only, and everyone is able to calculate balances from the ledger.

The only remaining design element is creating a decentralized append-only ledger, which we will discuss next.

16.6. Hash chains

Recall that we need a public ledger that is append-only and immutable: everyone can add entries to the ledger, but nobody can modify or delete existing entries.

To build this ledger, we will start with a *hash chain*. Suppose we have five messages, $m_1, m_2, \ldots, m_5$ that we want to append to the ledger. The resulting hash chain would look like this:

Block 1	Block 2	Block 3	Block 4	Block 5
m_1	$m_2, H(\text{Block 1})$	$m_3, H(\text{Block 2})$	$m_4, H(\text{Block 3})$	$m_5, H(\text{Block 4})$

Note that each block contains the hash of the previous block, which in turn contains the hash of the previous block, etc. In other words, each time we append a new message in a new block, the hash of the previous block contains a digest of all the entries in the hash chain so far.

Another way to see this is to write out the hashes. For example:

$$\begin{aligned}\text{Block 4} &= m_4, H(\text{Block 3}) \\ &= m_4, H(m_3, H(\text{Block 2})) \\ &= m_4, H(m_3, H(m_2, H(\text{Block 1}))) \\ &= m_4, H(m_3, H(m_2, H(m_1)))\end{aligned}$$

Note that Block 4 contains a digest of all the messages so far, namely m_1, m_2, m_3, m_4.

16.7. Properties of Hash Chains

Assume that Alice is given the $H(\text{Block } i)$ from a trusted source, but she downloads blocks 1 through i from an untrusted source. Only using the $H(\text{Block } i)$, Alice can verify that the blocks she downloaded from the untrusted source are not compromised by recomputing the hashes of each block, checking that they match the hash in the next block, and so on, until the last block, which she checks against the hash she received from the trusted source. Let's walk through an example:

Say Alice received the $H(\text{Block } 4)$ from somewhere she trusts and then fetches the entire blockchain from a compromised server (so she downloads blocks 1 through 4). Can an attacker give Alice an incorrect chain, say with block 2 being incorrect, without her detecting it? No! Since we use cryptographic hashes, which are collision resistant, two different blocks cannot hash to the same value. Say that block 2 is incorrect and Alice instead received block $2'$, then $H(\text{Block } 2') \neq H(\text{Block } 2)$. Since block 3 includes the hash of block $2'$, block 3 will also be incorrect, so the third block that Alice received is block $3' \neq$ block 3. So, $H(\text{Block } 3') \neq H(\text{Block } 3)$. Then, since block 4 includes the hash of block $3'$, block 4 will also be incorrect, so the fourth block that Alice received is block $4' \neq$ block 4. So, $H(\text{Block } 4') \neq H(\text{Block } 4)$. Since Alice received H(block 4) from a trusted source, and it does not match up with $H(\text{Block}4')$, Alice is able to detect misbehavior. On the other hand, if the $H(\text{Block } 4')$ did match $H(\text{Block } 4)$, then the blockchain that Alice downloaded is correct, and we have no misbehavior.

So, perhaps the most important property in a hash chain is that if you get the hash of the latest block from a trusted source, then you can verify that all of the previous history is correct.

16.8. Consensus in Bitcoin

In Bitcoin, every participant in the network stores the entire blockchain (and thus all of its history) since we don't utilize a centralized server. When someone wants to create a new transaction, they broadcast that transaction to everyone, and each user on the network has to check the transaction. If the transaction is correct, they will append it to their local blockchain.

The issue is that some users might be malicious, meaning that they might not append certain transactions or might not check certain transactions correctly or might replay certain transactions or might allow invalid transactions. Bitcoin, however, assumes that the majority of users are honest.

Perhaps one of the biggest issues is forks, which are essentially different versions of the blockchain that exist at the same time. For example, say that Mallory bought a house from Bob for 500 B, and this transaction is appended to the ledger. Mallory can then try "go back in time" and start the blockchain from just before this transaction was added to it, and can start appending new transaction entries from there. If Mallory can get other users to accept this new forked chain, she can get her 500 B back!

This means that we need a way for all users to agree on the content of the blockchain: *consensus via proof of work*.

16.9. Consensus via Proof of Work

In Bitcoin, while every user locally stores the entire blockchain, not every user can add a block. This special privilege is reserved for certain users, known as *miners*, who can only add a block if they have a valid proof of work. A miner validates transactions before solving a *proof of work*, which, if completed before any other miner, allows the miner to append the block to the blockchain. The proof of work is a computational puzzle that takes the hash of the current block concatenated with a random number. This random number can be incremented so that the hash changes, until the proof of work is solved. The proof of work is considered solved when the resulting hash starts with N zero bits, where the value of N (e.g. 33) is determined by the Bitcoin algorithm.

Miners then broadcast blocks with their proof of work. All honest miners listen for such blocks, check the blocks for correctness, and *accept the longest correct chain*. If a miner appends a block with some incorrect transaction, the block is ignored. The key idea for consensus is that everyone will always prefer the longest correct chain. Thus, if multiple miners append blocks at the same time, consensus is gained by the longest

correct chain, and the rest of the "versions" are discarded. When two different miners at the same time solve a proof of work and append two different blocks, thus forking the network, the next miner that appends onto one of these chains invalidates the other chain.

Say for example that an honest miner M_1 stores the current local blockchain $b_1 \rightarrow b_2 \rightarrow b_3$, and hears about transaction T. M_1 checks T, then tries to mine (solve for the proof of work) for a new block b_4 to now include transaction T. However, if miner M_2 mines b_4 first, M_2 will broadcast $b_1 \rightarrow b_2 \rightarrow b_3 \rightarrow b_4$. M_1 checks b_4, accepts it, gives up mining block 4, then starts to mine for block 5. M_1 now has the blockchain $b_1 \rightarrow b_2 \rightarrow b_3 \rightarrow b_4$ stored locally and has started to mine b_5. However, if M_1 hears miner 3 broadcasts $b_1 \rightarrow b_2 \rightarrow b_3 \rightarrow b'_4 \rightarrow b'_5$, M_1 will discard the shorter blockchain ($b_1 \rightarrow b_2 \rightarrow b_3 \rightarrow b_4$) in favor of the longer one ($b_1 \rightarrow b_2 \rightarrow b_3 \rightarrow b'_4 \rightarrow b'_5$). By always accepting the longest blockchain, all the miners are ensured to have the same blockchain view.

Remember that Bitcoin assumes that more than half of the users are honest, meaning that more than half of the computing power is in the hands of honest miners, thus ensuring that honest miners will always have an advantage to mine the longest chain. Going back to the example about forks that prompted this discussion, if proof of consensus is implemented, Mallory cannot fork the blockchain since she does not have $>50\%$ of the computing power in the world. Since the longest chain is always taken as the accepted, Mallory's forked chain will be shorter unless she can mine new entries faster than the aggregate mining power of everyone else in the world.

Go forth and mine!