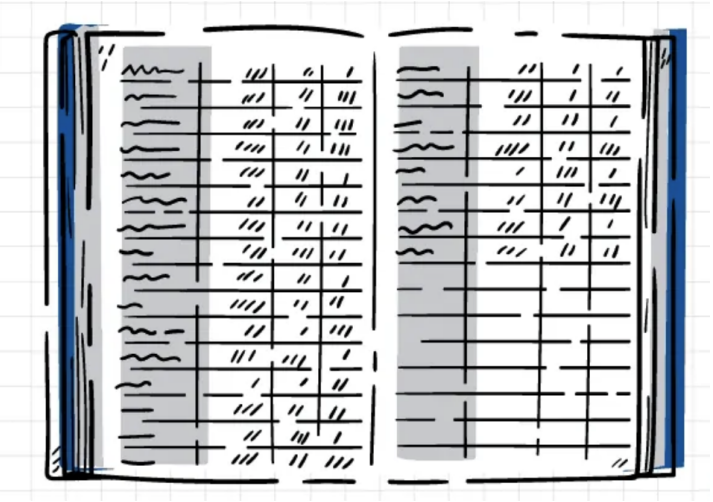
Introduction to blockchain technology and smart-contract programming

Sylvain Conchon
University Paris-Saclay

Blockchain Ledger

The database in a blockchain is similar to a general ledger in traditional accounting, which is a record-keeping system used to track financial transactions and balances

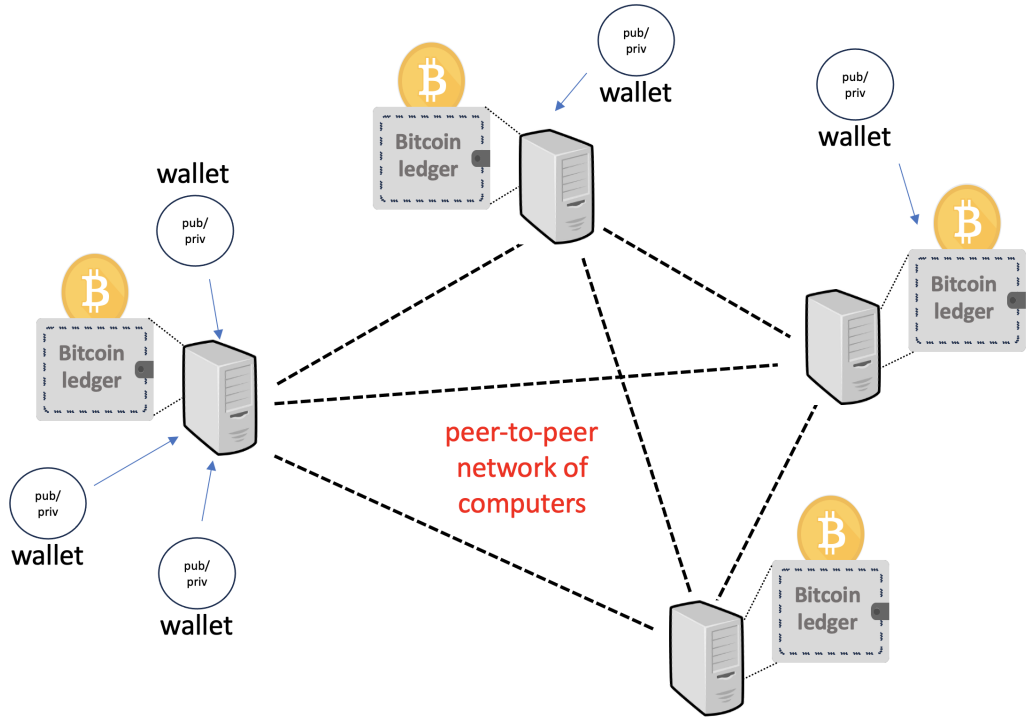


However, the ledger in a blockchain is distinguished by its decentralization, immutability, and cryptographic security mechanisms, making it resistant to tampering and fraud

Distributed Ledger

Blockchain ledgers are

- shared
- synchronized
- accessible in P2P
- Visible to everyone
- immutable



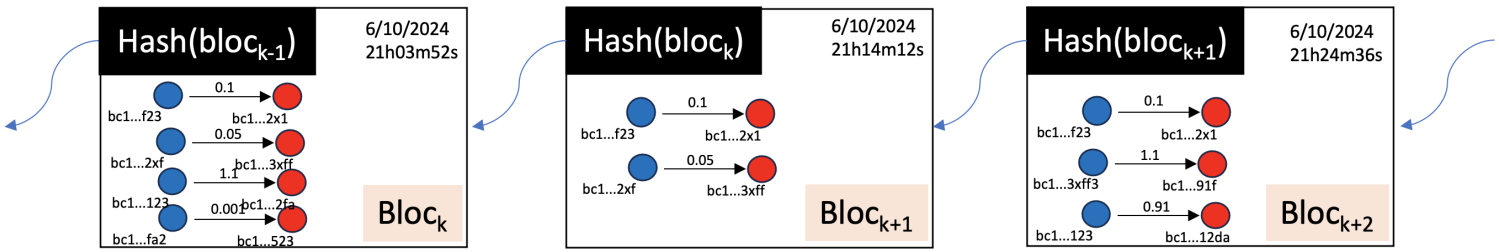
Computers that maintain a complete copy of the ledger are known as full nodes

- Resilience : decentralization makes the system more robust against failures, attacks, and data loss

Chain of Blocks

The ledger of a blockchain is used to records transactions

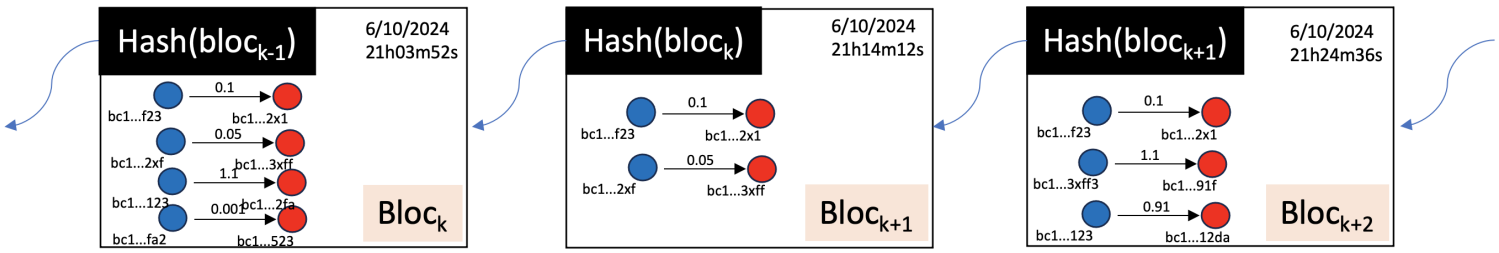
Transactions are grouped into "blocks". A block is similar to a page in a general ledger.



Chain of Blocks

The ledger of a blockchain is used to records transactions

Transactions are grouped into "blocks". A block is similar to a page in a general ledger.

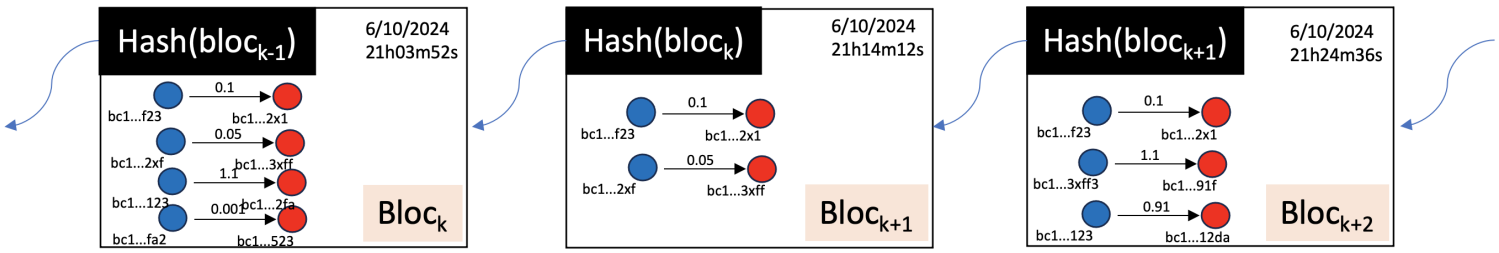


Chain of Blocks: Each block is linked to the previous one through its hash (a kind of unique identifier), forming a continuous chain of blocks—hence the name "blockchain"

Chain of Blocks

The ledger of a blockchain is used to records transactions

Transactions are grouped into "blocks". A block is similar to a page in a general ledger.



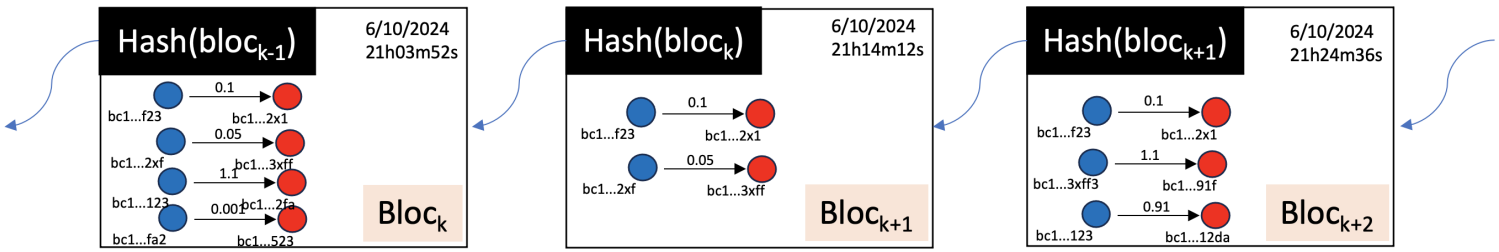
Chain of Blocks: Each block is linked to the previous one through its hash (a kind of unique identifier), forming a continuous chain of blocks—hence the name "blockchain"

Integrity. If a block is altered, its hash changes, invalidating all subsequent blocks, ensuring the integrity of the ledger

Chain of Blocks

The ledger of a blockchain is used to records transactions

Transactions are grouped into "blocks". A block is similar to a page in a general ledger.



Chain of Blocks: Each block is linked to the previous one through its hash (a kind of unique identifier), forming a continuous chain of blocks—hence the name "blockchain"

Integrity. If a block is altered, its hash changes, invalidating all subsequent blocks, ensuring the integrity of the ledger

Immutability. Once a block is added to the blockchain, it becomes nearly impossible to alter or delete, ensuring the transparency and security of the ledger

Bitcoin

Origins

Described in 2008 by a certain Satoshi Nakamoto in the paper Bitcoin: A peer-to-peer electronic cash system

Bitcoin aims to build a distributed payment system, without trusted third parties, for an arbitrary number of participants.

The main difficulty is to design a consensus protocol allowing all participants to agree, without knowing the number of participants.

The problem of distributed consensus with a known number of participants has been studied since the late 1970s, notably through the Byzantine Generals Problem (see later).

Double Spending

Unlike physical currency, an electronic coin can be copied effortlessly, which potentially allows one to spend the same coin multiple times.

This problem does not arise when the currency is managed in a centralized manner by a bank (which is the only entity that keeps the accounts and authorizes or rejects transactions).

In a distributed (or decentralized) system, one solution to address this problem is for the participants to vote in order to agree on the state of the accounts after each transaction.

Sybil Attack

The problem with a voting system involving an arbitrary number of participants is that some may attempt to cheat by creating multiple identities, in order to increase their voting power.

This type of attack is known as a Sybil attack.

Proof-of-Work

Bitcoin's solution to address double spending and Sybil attacks is Proof-of-Work (PoW).

PoW: A block is valid if its hash begins with a certain number of zeros. This condition can only be satisfied by brute force, which makes block production very computationally expensive (and therefore energy-intensive).

In Proof-of-Work, one vote = one CPU.

Thus, multiplying identities in this system amounts to having more computational power, which comes at a high cost.

Longest Chain Rule

What happens if two participants produce a block at the same time?

This problem is related to double spending, since spending the same bitcoin twice only requires producing two blocks with the same parent.

A fork in the blockchain occurs when two blocks have the same parent.

When multiple branches exist in Bitcoin, the rule for adding a new block is to choose the chain with the greatest cumulative proof-of-work (often referred to as the longest chain).

Nakamoto Consensus

This clever combination of Proof-of-Work and the longest chain rule is known as Nakamoto consensus.

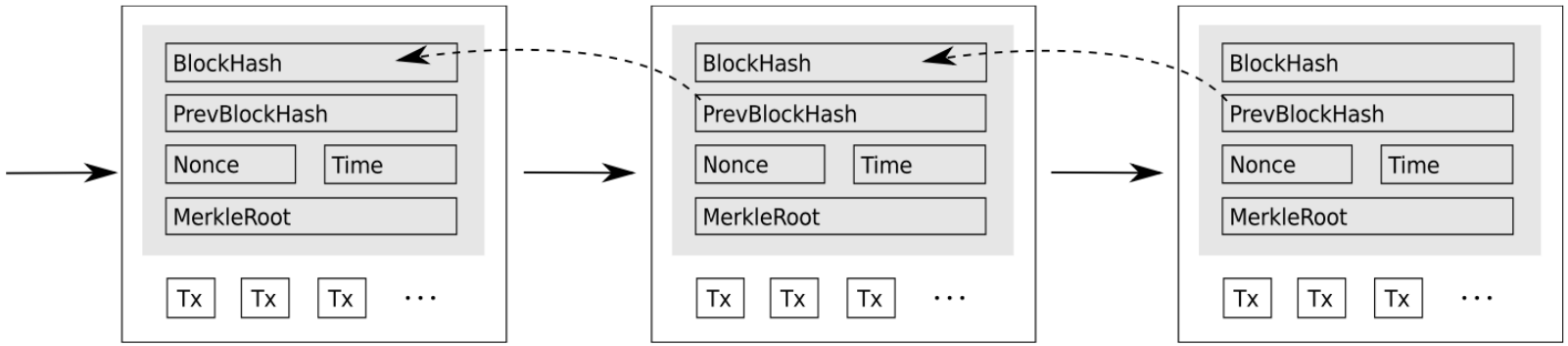
The chain with the greatest accumulated proof-of-work (i.e., the longest chain) is considered valid.

As blocks are added and become older, they become less and less likely to be reverted.

This concept is known as probabilistic finality: a block is considered final with overwhelming probability and is expected to remain permanently in the chain.

Bitcoin: Blockchain

(source:
F. Tschorsch and B. Scheuermann, *Bitcoin and Beyond: A Technical Survey on Decentralized Digital Currencies*)



Bitcoin: Addresses

A Bitcoin address B is generated from a public key P by first hashing it with the SHA-256 function, then hashing the result again with RIPEMD-160.

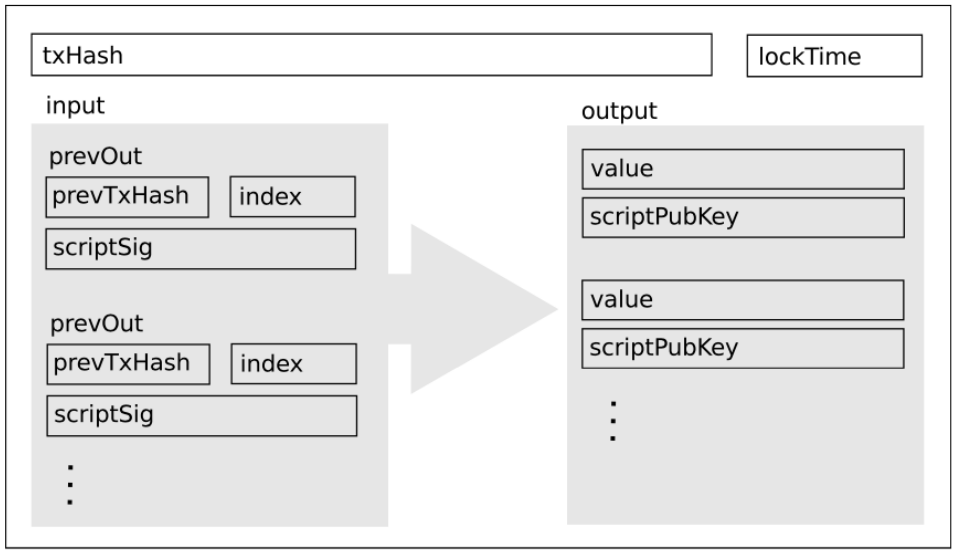
Finally, addresses are represented in Base58, a binary-to-ASCII encoding that uses 58 characters (omitting easily confused characters such as 0, O, l, 1, etc.).

This public key encoding allows both space savings and obfuscation of the keys.

Bitcoin: Transactions

In Bitcoin, a user's balance is represented by the set of transactions they have received from other users (or earned through mining or transaction fees) that have not yet been spent.

To spend funds, one must "consume" unspent transactions (UTXOs: *Unspent Transaction Outputs*) and link them to a new UTXO, which can have multiple outputs.



The Bitcoin `Script` Engine

Each Bitcoin transaction corresponds to a small program written in a language called `Script`.

This language allows controlling a stack-based virtual machine using a set of instructions (over a hundred).

`https://en.bitcoin.it/wiki/Script`

Examples of instructions:

- DUP
- EQUAL
- HASH160
- EQUALVERIFY
- CHECKSIG
- CHECKMULTISIG
- IF, ELSE, ENDIF
- ...

Bitcoin: Scripts (1/2)

There are two types of Bitcoin scripts: locking scripts (Lock or scriptPubKey) and unlocking scripts (Unlock or scriptSig). Bitcoin transactions primarily use four families of scripts.

P2PK

Lock: `<pk>CHECKSIG`

Unlock: `<sig>`

P2PKH :

Lock: `DUP HASH160 <Hpk> EQUALVERIFY CHECKSIG`

Unlock: `<sig> <pk>`

P2MS :

Lock: `<n> <pkh>...<pkh> <m> CHECKMULTISIG`

Unlock: `<sig>...<sig>`

P2SH :

Lock: `HASH160 <hashscript> EQUAL`

Unlock: `<sig>...<sig> <serialized_script>`

Bitcoin: Scripts (2/2)

Here is a brief explanation of the main script types:

P2PK (Pay-to-Public-Key): Simple script where funds are locked to a public key. To spend, the spender provides a valid signature.

P2PKH (Pay-to-Public-Key-Hash): The most common type. Funds are locked to the hash of a public key, providing both space savings and obfuscation. Spending requires providing the public key and a valid signature.

P2MS (Pay-to-Multi-Sig): Funds are locked to multiple public keys with an m-of-n spending condition. Requires m valid signatures to spend.

P2SH (Pay-to-Script-Hash): Funds are locked to the hash of a redeem script. The spender must provide the signatures and the serialized script matching the hash. This allows complex scripts without exposing them on-chain until spent.

Bitcoin : script example

Stack	Script
	`sigBob pubKeyBob OP_DUP OP_HASH160 pubKeyBobHash OP_EQUALVERIFY OP_CHECKSIG`
`sigBob pubKeyBob`	`OP_DUP OP_HASH160 pubKeyBobHash OP_EQUALVERIFY OP_CHECKSIG`
`sigBob pubKeyBob pubKeyBob`	`OP_HASH160 pubKeyBobHash OP_EQUALVERIFY OP_CHECKSIG`
`sigBob pubKeyBob pubKeyBobHash`	`pubKeyBobHash OP_EQUALVERIFY OP_CHECKSIG`
`sigBob pubKeyBob pubKeyBobHash pubKeyBobHash`	`OP_EQUALVERIFY OP_CHECKSIG`
`sigBob pubKeyBob`	`OP_CHECKSIG`
`true`	

\begin{Practice}

Bitcoin scripts

Exercise : Identify a P2PKH on the Blockchain

Understand how a P2PKH script works in a real transaction.

1. Choose a recent transaction on a block explorer.

2. Identify one input and one output of the transaction.

3. Determine the type of the output script (scriptPubKey) and check if it is a P2PKH.

Note the following:

- Destination address
- Amount sent
- Public key hash used in the script

Exercise: Analyze a P2SH Transaction

Understand how a P2SH script works and how multiple signatures are used.

1. On a block explorer, find a transaction that uses P2SH (often a multi-signature script).

2. Examine the corresponding output (scriptPubKey) and note the hash of the redeem script.

3. Look at the input (scriptSig) that spends this output.

Identify: How many signatures were provided If the script requires an m-of-n condition (e.g., 2-of-3 signatures)

\end{Practice}

Bitcoin Light Clients

Light Clients

A blockchain consists of two main types of nodes: miners and wallets.

Wallets are light clients that do not operate as full nodes in the P2P network: they do not attempt to mine blocks or download the entire blockchain.

They simply track the transactions they are involved in by querying a blockchain server.

The underlying technology behind these wallets is Simplified Payment Verification (SPV), which allows users to verify that a transaction is included in the blockchain without downloading the entire chain.

Wallets

Light clients on a blockchain like Bitcoin are called wallets.

Unlike full nodes in the P2P network, *wallets* are used by blockchain users to create transactions and track them.

To perform these operations, a *wallet* connects to a network node to obtain the minimal information needed to verify that a transaction is confirmed, without ever downloading the entire blockchain.

This feature is called Simplified Payment Verification (SPV).

Problem Statement

How can we verify that a transaction is included in block *i* without downloading all the transactions in that block, but by using only the block header?

Transaction Hash List

One solution is to compute the digital hash of the list of transactions in a block and store it in the block header.

Transaction Hash List

One solution is to compute the digital hash of the list of transactions in a block and store it in the block header.

Given the list $[H_0, H_1, \ldots, H_k]$ of transaction hashes, one could, for example, compute the hash of the list as the hash of the sum $H_0 + \ldots + H_k$.

Transaction Hash List

One solution is to compute the digital hash of the list of transactions in a block and store it in the block header.

Given the list $[H_0, H_1, \ldots, H_k]$ of transaction hashes, one could, for example, compute the hash of the list as the hash of the sum $H_0 + \ldots + H_k$.

To verify that a particular transaction v is included in block b_i, one can:

Transaction Hash List

One solution is to compute the digital hash of the list of transactions in a block and store it in the block header.

Given the list $[H_0, H_1, \ldots, H_k]$ of transaction hashes, one could, for example, compute the hash of the list as the hash of the sum $H_0 + \ldots + H_k$.

To verify that a particular transaction v is included in block b_i, one can:

- Request (as proof) from a network node the list ℓ of hashes of all transactions in b_i

Transaction Hash List

One solution is to compute the digital hash of the list of transactions in a block and store it in the block header.

Given the list $[H_0, H_1, \ldots, H_k]$ of transaction hashes, one could, for example, compute the hash of the list as the hash of the sum $H_0 + \ldots + H_k$.

To verify that a particular transaction v is included in block b_i, one can:

- Request (as proof) from a network node the list ℓ of hashes of all transactions in b_i
- Check that the hash of v is in ℓ

Transaction Hash List

One solution is to compute the digital hash of the list of transactions in a block and store it in the block header.

Given the list $[H_0, H_1, \ldots, H_k]$ of transaction hashes, one could, for example, compute the hash of the list as the hash of the sum $H_0 + \ldots + H_k$.

To verify that a particular transaction v is included in block b_i, one can:

- Request (as proof) from a network node the list ℓ of hashes of all transactions in b_i
- Check that the hash of v is in ℓ
- Recompute the hash of ℓ and verify that it matches the hash of the transactions stored in the header of b_i

Merkle Trees

To minimize the size of ℓ, we organize the list of transaction hashes as a tree.

This tree structure is called a Merkle tree. It was invented in 1979 by Ralph Merkle. Initially filed as a patent, Merkle later published it in 1987 in the following paper:

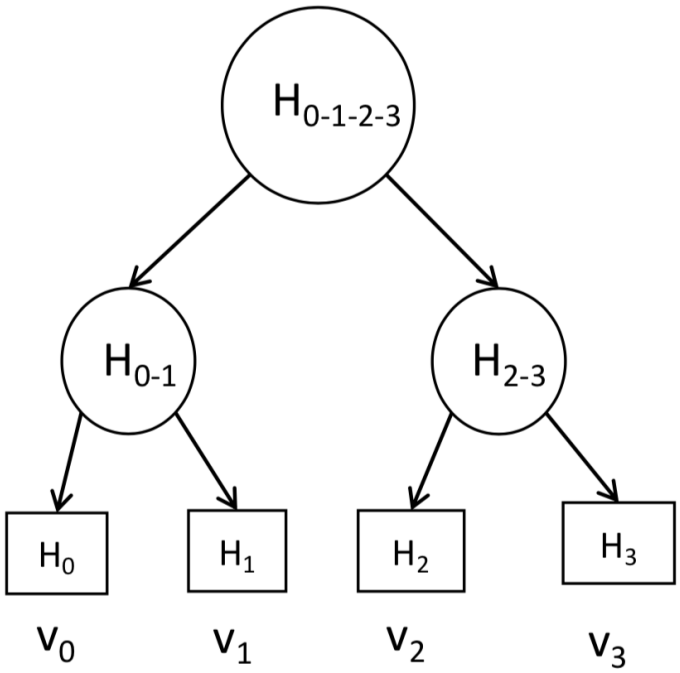
R. Merkle. A digital signature based on a conventional encryption function.

`https://people.eecs.berkeley.edu/~raluca/cs261-f15/readings/merkle.pdf`

Merkle Tree: Principle

A Merkle tree is a binary tree structure in which:

- Each leaf contains the hash of a transaction
- Each node contains the hash of the concatenation of its two child hashes.



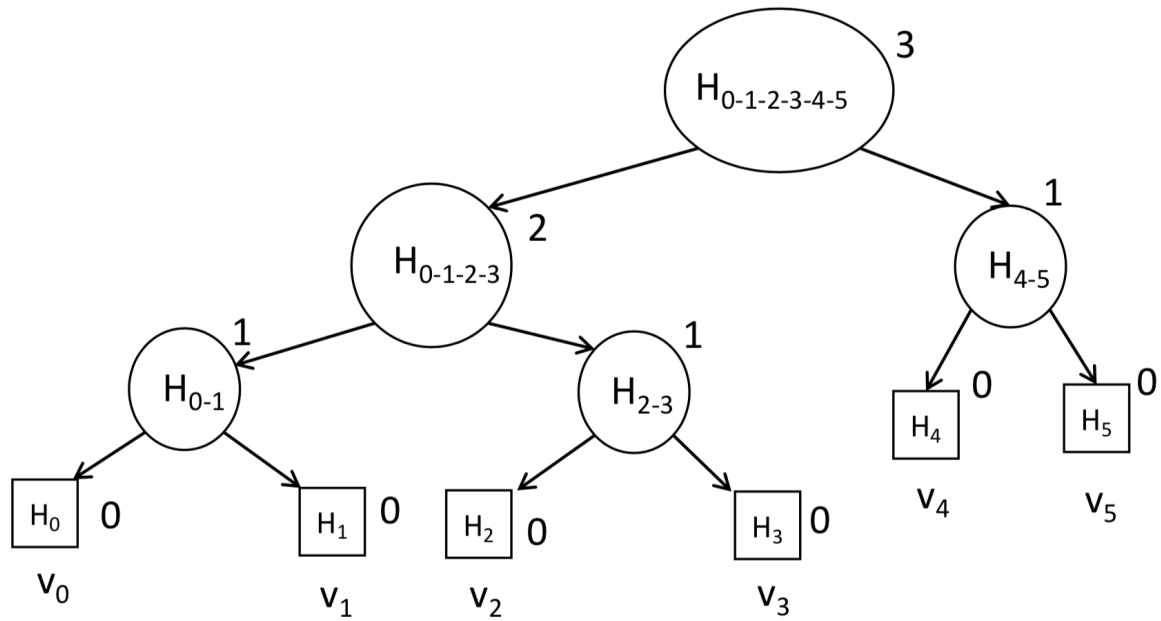
- Leaves $H_0, \ldots, H_3$ contain the hashes of transactions $v_0, \ldots, v_3$
- Internal nodes H_{i-j} are equal to $\mathrm{hash}(H_i + H_j)$
- Finally, the root $H_{0-1-2-3}$ is equal to $\mathrm{hash}(H_{0-1} + H_{2-3})$

Merkle Trees: Property

Merkle trees have the property that left subtrees are always complete, meaning that all levels of these subtrees are fully filled.

Therefore, a node at level n will necessarily have 2^{n-1} leaves in its left subtree.

Example:



Merkle Root

It is only the hash stored at the root of the Merkle tree that is recorded in the block header of a blockchain.

Only the nodes of the blockchain's P2P network store the transactions, which are kept in each block in the form of Merkle trees.

Merkle Proofs: Principle

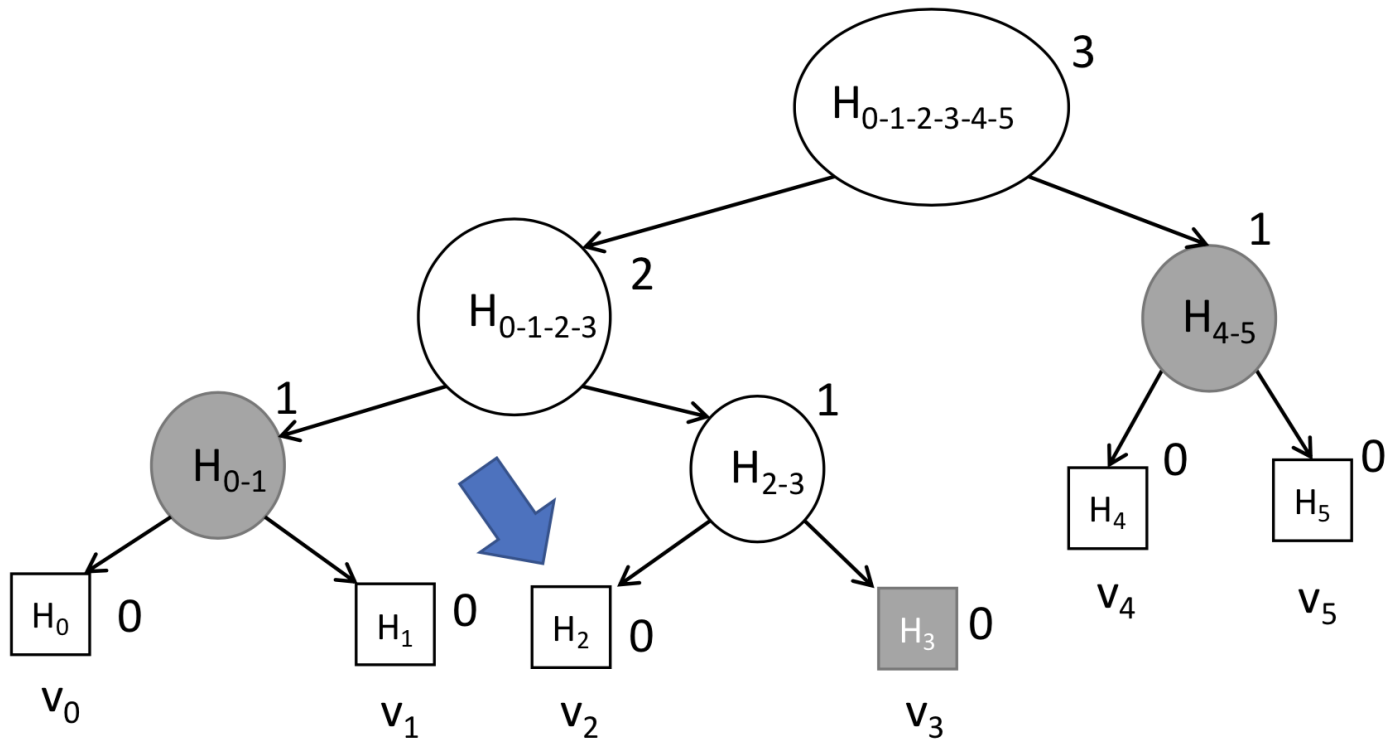
To verify that a transaction v is included in a block $<$ (for which we only have the header), one only needs to provide the list of hashes contained in the sibling nodes along the path from the root of the Merkle tree to the leaf node holding the hash of v.

This list, called a Merkle proof, allows one to recompute the root hash of the tree and verify that it matches the value stored in the block header.

Merkle Proofs: Example

The Merkle proof for transaction v_2 in the tree below is the list $[H_{4-5}; H_{0-1}; H_3]$.
Indeed, we have:

$$H_{0-1-2-3-4-5} = \text{hash}(H_{4-5} + \text{hash}(H_{0-1} + \text{hash}(\text{hash}(v_2) + H_3)))$$



\begin{Practice}

Merkle trees

Exercise

Using the Blockstream API:

`https://github.com/Blockstream/esplora/blob/master/API.md`

and a tool like `curl` or a library such as Python's `requests`:

1. Write a program that retrieves the Merkle root of a block and the Merkle proof for the inclusion of a transaction.

2. Verify that the Merkle proof is correct.

Hint 1 : Hashing and byte orders

In practice, the hash values are sequence of bytes, and the addition operation is just the concatenation.

We use `hashlib.sha256` to compute hash and Merkle proofs use double `sha256`

Bitcoin uses Big-Endian to display information

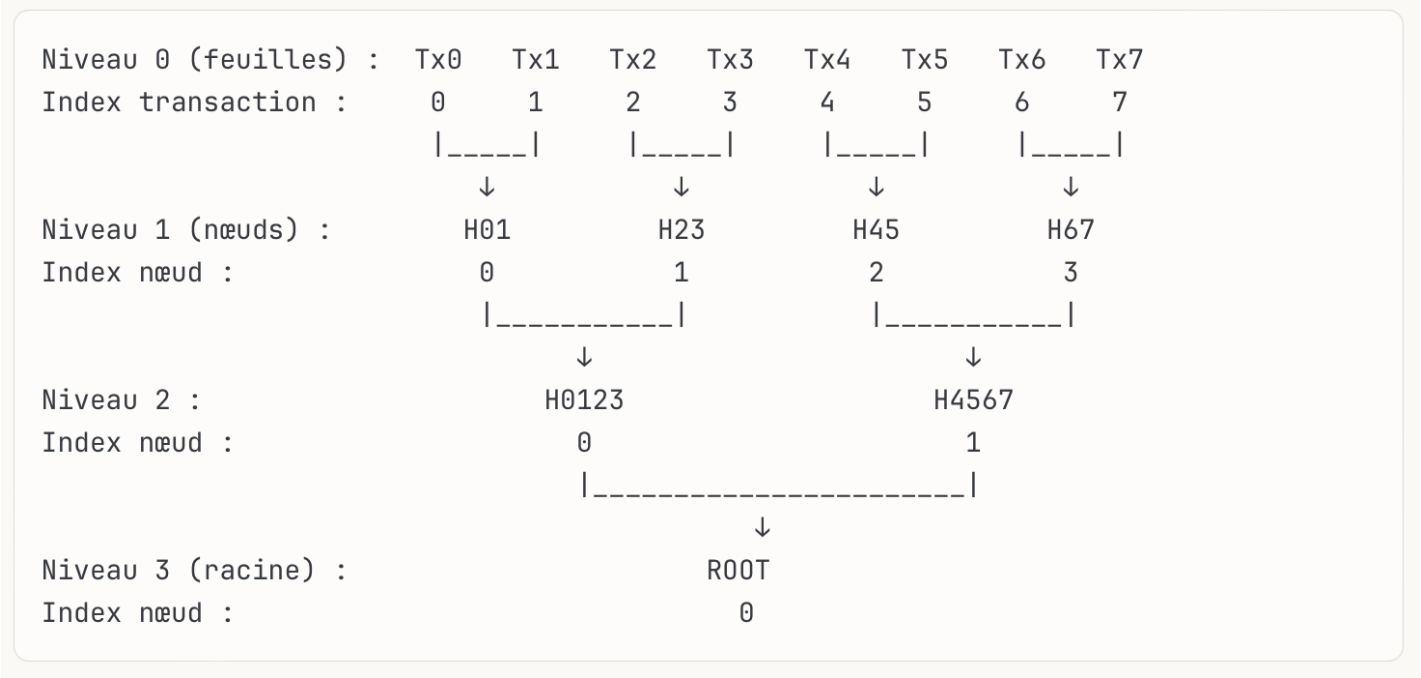
`"a1b2c3d4"` $\rightarrow$ reads left to right

but Little-Endian is used for internal calculations (hashing)

`d4 c3 b2 a1` $\rightarrow$ reversed order in memory

Hint 2: Merkle indexes and division by 2

Since concatenation is not commutative, it is necessary to know whether to concatenate on the left or on the right.



```
\end{Practice}
```

Lightning Network

Scaling Bitcoin

Currently, the main challenge for a blockchain like Bitcoin is scalability.

This is usually measured by the number of transactions per second (TPS) that a blockchain can handle.

Scaling Bitcoin

Currently, the main challenge for a blockchain like Bitcoin is scalability.

This is usually measured by the number of transactions per second (TPS) that a blockchain can handle.

Bitcoin produces 1 block every 10 minutes, and each block can contain up to 1 megabyte (= 1,048,576 bytes) of data. On average, a transaction is 380 bytes in size.

$$
\begin{aligned}
T_B(\text{Block time}) &= 600 \text{ seconds} \\
B(\text{Block size}) &= 1{,}048{,}576 \text{ bytes} \\
A(\text{Average transaction size}) &= 380 \text{ bytes}
\end{aligned}
$$

Scaling Bitcoin

Currently, the main challenge for a blockchain like Bitcoin is scalability.

This is usually measured by the number of transactions per second (TPS) that a blockchain can handle.

Bitcoin produces 1 block every 10 minutes, and each block can contain up to 1 megabyte (= 1,048,576 bytes) of data. On average, a transaction is 380 bytes in size.

$$
\begin{aligned}
T_B(\text{Block time}) &= 600 \text{ seconds} \\
B(\text{Block size}) &= 1{,}048{,}576 \text{ bytes} \\
A(\text{Average transaction size}) &= 380 \text{ bytes}
\end{aligned}
$$

Number of transactions per block (TPB) and the TPS value are:

$$
\begin{aligned}
TPB &= B/A = 1{,}048{,}576/380 \approx 2{,}759 \\
TPS &= TPB/T_B \approx 2{,}759/600 \approx 4.6
\end{aligned}
$$

How to Increase TPS?

Bitcoin currently allows an average of 4.6 transactions per second (TPS). In comparison, the Visa network can handle peaks of 50,000 TPS and hundreds of millions of transactions per day.

To achieve such throughput on Bitcoin, a block of 11 GB would need to be mined every 10 minutes, which amounts to roughly 600 petabytes per year!

Moreover, in 2021, Bitcoin has about 10,000 nodes, and the average time to propagate a block to all these nodes is estimated at 14 seconds.

How to Increase TPS?

Bitcoin currently allows an average of 4.6 transactions per second (TPS). In comparison, the Visa network can handle peaks of 50,000 TPS and hundreds of millions of transactions per day.

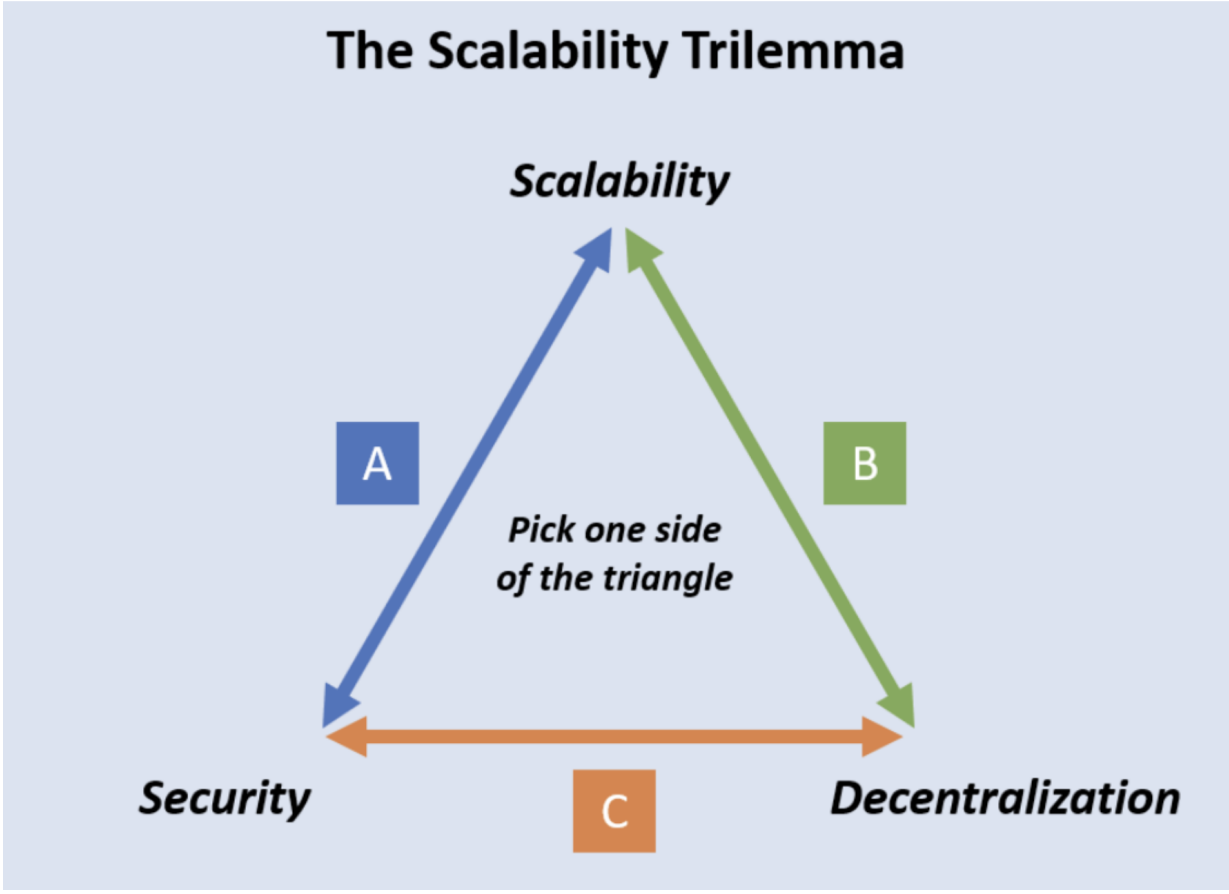
To achieve such throughput on Bitcoin, a block of 11 GB would need to be mined every 10 minutes, which amounts to roughly 600 petabytes per year!

Moreover, in 2021, Bitcoin has about 10,000 nodes, and the average time to propagate a block to all these nodes is estimated at 14 seconds.

Increasing the block size is not a good solution:

- Only a few nodes could handle storing such large amounts of data
- Fewer miners = less security
- Blockchain validation would become impossible

The Blockchain Trilemma



It is challenging for a blockchain to achieve all three properties simultaneously: scalability, decentralization, and security.

Lightning Network

One solution to increase Bitcoin's scalability is to build a secondary layer on top of Bitcoin, where users can make micro-payments that are (almost) instantaneous and low-fee through off-chain communications.

The Bitcoin blockchain is used to secure these payments.

The main secondary layer for Bitcoin is the Lightning Network.

It is a network of micropayment channels.

Micropayment Channels

A micropayment channel allows two “agents” to make payments very quickly, without using the blockchain, except at the opening and closing of the channel.

Micropayment Channels

A micropayment channel allows two “agents” to make payments very quickly, without using the blockchain, except at the opening and closing of the channel.

Suppose Alice and Bob want to regularly exchange money (in both directions).

Micropayment Channels

A micropayment channel allows two “agents” to make payments very quickly, without using the blockchain, except at the opening and closing of the channel.

Suppose Alice and Bob want to regularly exchange money (in both directions).

To do this, Alice and Bob start by opening a shared (multisig) account on Bitcoin, each depositing a certain amount—for example, Alice deposits 1 BTC and Bob deposits 0.5 BTC.

How can the opening of such an account be done securely?

Opening a Channel

Opening this shared account (AB) requires two transactions:

1. $A \xrightarrow{1} B$: 1 BTC from Alice to AB (signed by Alice)
2. $B \xrightarrow{0.5} A$: 0.5 BTC from Bob to AB (signed by Bob)

Opening a Channel

Opening this shared account (AB) requires two transactions:

1. $A \xrightarrow{1} B$: 1 BTC from Alice to AB (signed by Alice)
2. $B \xrightarrow{0.5} A$: 0.5 BTC from Bob to AB (signed by Bob)

But this is not enough, because if Bob does not cooperate, Alice would lose her 1 BTC in the shared account (and vice versa), since she could no longer retrieve her funds.

Opening a Channel

Opening this shared account (AB) requires two transactions:

1. $A \xrightarrow{1} B$: 1 BTC from Alice to AB (signed by Alice)
2. $B \xrightarrow{0.5} A$: 0.5 BTC from Bob to AB (signed by Bob)

But this is not enough, because if Bob does not cooperate, Alice would lose her 1 BTC in the shared account (and vice versa), since she could no longer retrieve her funds.

To prevent this, before broadcasting their transactions, Alice and Bob each construct and sign the following two-output transaction, which represents the state of the payment channel.

$$AB \begin{array}{l} \xrightarrow{1} A \\ \xrightarrow{0.5} B \end{array}$$

Alice and Bob keep this transaction (the commitment) and only broadcast the first two transactions on Bitcoin. This allows the payment channel to be opened safely and securely.

Micropayments

Suppose Alice wants to send 0.1 BTC to Bob using their shared account. After the payment, Alice will have 0.9 BTC and Bob will have 0.6 BTC in the account.

Instead of broadcasting this transaction on Bitcoin, Alice and Bob can *simply* discard the previous commitment transaction representing the channel state and replace it with a new transaction:

$$\mathrm{AB} \begin{array}{l} \xrightarrow{0.9} \mathrm{A} \\ \xrightarrow{0.6} \mathrm{B} \end{array}$$

Like the previous one, this transaction does not need to be sent to Bitcoin, it is only kept securely by Alice and Bob.

The mere fact that Alice and Bob hold this transaction allows them to acknowledge the transfer.

Timelock and Revocation Key

One important problem remains:

How can we ensure that the commitment $AB \xrightarrow{1} A$, $\xrightarrow{0.5} B$ is no longer usable?

Timelock and Revocation Key

One important problem remains:

How can we ensure that the commitment AB $\begin{matrix} \xrightarrow{1} A \\ \xrightarrow{0.5} B \end{matrix}$ is no longer usable?

To solve this, we define scripts that allow transactions on these payment channels to be locked temporarily and protected with revocation keys.

Timelock and Revocation Key

One important problem remains:

How can we ensure that the commitment AB $\begin{matrix} \xrightarrow{1} A \\ \xrightarrow{0.5} B \end{matrix}$ is no longer usable?

To solve this, we define scripts that allow transactions on these payment channels to be locked temporarily and protected with revocation keys.

A timelock script prevents a transaction (or its output) from being spent for a specified number of blocks.

A script with a revocation key prevents a transaction (or its output) from being spent if the signer of the transaction does not possess the revocation key.

Secure Commitment Transactions

The commitments constructed by Alice and Bob are defined as contracts with Timelock and a revocation key. Alice's commitment has the following form:

$$
\begin{array}{rll}
 & \xrightarrow{1} \mathsf{A} & \text{timelock N} \\
\mathsf{AB} & \xrightarrow{1} \mathsf{B} & \text{revocation key} = (\mathsf{HS}_\mathsf{A}, \mathsf{HS}_\mathsf{B}) \\
 & \xrightarrow{0.5} \mathsf{B} &
\end{array}
$$

where HS_A (resp. HS_B) is the hash of a secret held only by A (resp. B). If this transaction is broadcast on Bitcoin, then

Secure Commitment Transactions

The commitments constructed by Alice and Bob are defined as contracts with Timelock and a revocation key. Alice's commitment has the following form:

$$\mathsf{AB} \begin{array}{ll} \xrightarrow{1} \mathsf{A} & \text{timelock N} \\ \xrightarrow{1} \mathsf{B} & \text{revocation key} = (\mathsf{HS_A}, \mathsf{HS_B}) \\ \xrightarrow{0.5} \mathsf{B} & \end{array}$$

where $\mathsf{HS_A}$ (resp. $\mathsf{HS_B}$) is the hash of a secret held only by A (resp. B). If this transaction is broadcast on Bitcoin, then

- The effect of the output $\mathsf{AB} \xrightarrow{1} \mathsf{A}$ is delayed by N blocks. Thus, A can only retrieve her funds after N blocks.
- The effect of the output $\mathsf{AB} \xrightarrow{1} \mathsf{B}$ is possible only if the signer can provide two secrets $\mathsf{S_A}$ and $\mathsf{S_B}$ such that $\mathit{hash}(\mathsf{S_A}) = \mathsf{HS_A}$ and $\mathit{hash}(\mathsf{S_B}) = \mathsf{HS_B}$. In this way, B can be credited with 1 BTC only if he knows A's secret.
- B can immediately receive 0.5 BTC.

Protocol on a Channel

Before creating a new commitment, Alice and Bob each generate a new secret S'_A and S'_B.

Protocol on a Channel

Before creating a new commitment, Alice and Bob each generate a new secret S'_A and S'_B.

They also exchange the hash values HS'_A and HS'_B of these secrets, along with the secrets S_A and S_B from the previous commitment.

Protocol on a Channel

Before creating a new commitment, Alice and Bob each generate a new secret S'_A and S'_B.

They also exchange the hash values HS'_A and HS'_B of these secrets, along with the secrets S_A and S_B from the previous commitment.

In this way, if A tries to cheat:

- B can immediately claim the funds from the previous state;
- B has enough time to take the remaining balance of the shared account, since he knows A's secret and the previous state is delayed by the timelock, preventing A from "stealing" funds.

Closing a Channel

To close a channel cooperatively, Alice and Bob must construct a final transaction, the closing transaction, in the following form:

$$\mathrm{AB} \begin{array}{l} \xrightarrow{x} \mathrm{A} \\ \xrightarrow{y} \mathrm{B} \end{array}$$

where x_A and x_B represent the balances of Alice and Bob in the shared account, respectively.

Alice and Bob must also agree on the transaction fees they are willing to pay to retrieve their funds within a reasonable time (the higher the fees, the more likely the transaction will be confirmed).

This allows for a fast and low-cost closure (compared to a forced closure using the last commitment transaction).

LN: A Network of Channels

It is impossible for a user to open a channel with every other user (too costly); the number of channels would be far too large.

How can Alice and Bob then exchange money if they do not share a direct channel?

LN: A Network of Channels

It is impossible for a user to open a channel with every other user (too costly); the number of channels would be far too large.

How can Alice and Bob then exchange money if they do not share a direct channel?

The Lightning Network is a graph of payment channels, with each channel connecting two users.

The nodes in this graph represent users, and the edges are the channels.

A user can open channels with multiple users; therefore, a node can have multiple neighbors.

LN: Example

To understand how this graph works, suppose that Alice and Charlie share a channel, and Bob and Charlie also share a channel.

$$\mathsf{A}(x_{\mathsf{A}}) \overset{x}{\longleftrightarrow} (x_{\mathsf{C}})\mathsf{C}(y_{\mathsf{C}}) \overset{y}{\longleftrightarrow} (y_{\mathsf{B}})\mathsf{B}$$

where x is the name of the channel between Alice and Charlie, and x_{A} (resp. x_{C}) represents the funds that Alice (resp. Charlie) has on this channel.

LN: Payment

Suppose Alice wants to send an amount d to Bob. To do this, she must first send d to Charlie, who then sends d to Bob. If this works, the resulting situation is:

$$\mathsf{A}(x_\mathsf{A} - d) \overset{x}{\longleftrightarrow} (x_\mathsf{C} + d)\mathsf{C}(y_\mathsf{C} - d) \overset{y}{\longleftrightarrow} (y_\mathsf{B} + d)\mathsf{B}$$

LN: Payment

Suppose Alice wants to send an amount d to Bob. To do this, she must first send d to Charlie, who then sends d to Bob. If this works, the resulting situation is:

$$\mathrm{A}(x_\mathrm{A} - d) \overset{x}{\longleftrightarrow} (x_\mathrm{C} + d)\mathrm{C}(y_\mathrm{C} - d) \overset{y}{\longleftrightarrow} (y_\mathrm{B} + d)\mathrm{B}$$

Notice that this operation is neutral for Charlie since his total funds remain unchanged.

LN: Payment

Suppose Alice wants to send an amount d to Bob. To do this, she must first send d to Charlie, who then sends d to Bob. If this works, the resulting situation is:

$$\mathsf{A}(x_\mathsf{A} - d) \overset{x}{\longleftrightarrow} (x_\mathsf{C} + d)\mathsf{C}(y_\mathsf{C} - d) \overset{y}{\longleftrightarrow} (y_\mathsf{B} + d)\mathsf{B}$$

Notice that this operation is neutral for Charlie since his total funds remain unchanged.
However, this transfer can fail if $y_\mathsf{C} < d$. In that case, Alice must:

- look for a different path in the graph, hoping that all nodes along the path have enough funds to route the payment
- or split the payment into multiple smaller transfers along different paths.

LN: Payment

Suppose Alice wants to send an amount d to Bob. To do this, she must first send d to Charlie, who then sends d to Bob. If this works, the resulting situation is:

$$\mathsf{A}(x_\mathsf{A} - d) \overset{x}{\longleftrightarrow} (x_\mathsf{C} + d)\mathsf{C}(y_\mathsf{C} - d) \overset{y}{\longleftrightarrow} (y_\mathsf{B} + d)\mathsf{B}$$

Notice that this operation is neutral for Charlie since his total funds remain unchanged.
However, this transfer can fail if $y_\mathsf{C} < d$. In that case, Alice must:

- look for a different path in the graph, hoping that all nodes along the path have enough funds to route the payment
- or split the payment into multiple smaller transfers along different paths.

Problem: only the initial capacities of channels are known (from reading the blockchain), but not their current capacities.

LN: Transfer Fees

The Lightning Network also includes transfer fees. These fees compensate the intermediate nodes that route the funds.

For the same channel, fees can differ depending on the direction of the transfer.

In the previous example, Charlie can charge a fee ϵ to forward the funds to Bob through his channel y. In this case, Alice must send $d + \epsilon$ to Charlie to cover the fee.

LN: Transfer Fees

The Lightning Network also includes transfer fees. These fees compensate the intermediate nodes that route the funds.

For the same channel, fees can differ depending on the direction of the transfer.

In the previous example, Charlie can charge a fee ϵ to forward the funds to Bob through his channel y. In this case, Alice must send $d + \epsilon$ to Charlie to cover the fee.

After the transfer, the state becomes:

$$\mathsf{A}(x_{\mathsf{A}} - (d + \epsilon)) \overset{x}{\longleftrightarrow} (x_{\mathsf{C}} + d + \epsilon)\mathsf{C}(y_{\mathsf{C}} - d) \overset{y}{\longleftrightarrow} (y_{\mathsf{B}} + d)\mathsf{B}$$

LN: Route Selection

Transfer fees depend on the path taken (since fees differ across nodes).

Unlike a traditional data network where each node (or router) decides which neighbor to forward a packet to, in the Lightning Network the sending nodes (like Alice) choose the entire path for their payment, for example based on:

- the fees they are willing to pay for the payment
- the capacities of the channels along the path to maximize the chance of success
- etc.

The LN application must maintain an up-to-date network map so that each user can select their paths.

HTLC

In Alice-to-Bob payments, what happens if Charlie does not forward the payment to Bob?

To secure transfers, special payment scripts called HTLCs (Hashed Time-Locked Contracts) are used.

HTLC

In Alice-to-Bob payments, what happens if Charlie does not forward the payment to Bob?

To secure transfers, special payment scripts called HTLCs (Hashed Time-Locked Contracts) are used.

The rules of these payments are:

- Conditional: a payment is settled only if the recipient can provide a certain secret
- Time-bound: a payment expires after a certain time if the secret is not revealed

Payment Protocol with HTLC

1. To securely send d BTC to Bob via Charlie, Alice first asks Bob to provide $H_s = hash(s)$, the hash of a secret s known only to Bob.

Payment Protocol with HTLC

1. To securely send d BTC to Bob via Charlie, Alice first asks Bob to provide $H_s = hash(s)$, the hash of a secret s known only to Bob.
2. Alice then sends a contract $HTLC(d, H_s, N)$ to Charlie. This contract allows Charlie to claim d BTC only if he can reveal the secret s. If Charlie does not reveal s before block N, the funds remain locked.

Payment Protocol with HTLC

1. To securely send d BTC to Bob via Charlie, Alice first asks Bob to provide $\mathrm{H}_s = \mathit{hash}(s)$, the hash of a secret s known only to Bob.
2. Alice then sends a contract $\mathrm{HTLC}(d, \mathrm{H}_s, \mathrm{N})$ to Charlie. This contract allows Charlie to claim d BTC only if he can reveal the secret s. If Charlie does not reveal s before block N, the funds remain locked.
3. Charlie sends $\mathrm{HTLC}(d, \mathrm{H}_s, \mathrm{N} - i)$ to Bob (with $1 \leq i$).

Payment Protocol with HTLC

1. To securely send d BTC to Bob via Charlie, Alice first asks Bob to provide $\mathrm{H}_s = hash(s)$, the hash of a secret s known only to Bob.
2. Alice then sends a contract $\mathrm{HTLC}(d, \mathrm{H}_s, \mathrm{N})$ to Charlie. This contract allows Charlie to claim d BTC only if he can reveal the secret s. If Charlie does not reveal s before block N, the funds remain locked.
3. Charlie sends $\mathrm{HTLC}(d, \mathrm{H}_s, \mathrm{N} - i)$ to Bob (with $1 \leq i$).

4. Bob unlocks the contract by proving he knows s; Upon seeing the secret, Charlie can then unlock his contract, finalizing the payment.

Payment Protocol with HTLC

1. To securely send d BTC to Bob via Charlie, Alice first asks Bob to provide $H_s = hash(s)$, the hash of a secret s known only to Bob.
2. Alice then sends a contract $HTLC(d, H_s, N)$ to Charlie. This contract allows Charlie to claim d BTC only if he can reveal the secret s. If Charlie does not reveal s before block N, the funds remain locked.
3. Charlie sends $HTLC(d, H_s, N - i)$ to Bob (with $1 \leq i$).

4. If Bob does not unlock the contract, the block $N - i$ will eventually be mined and the contract expires. Since Charlie does not learn the secret s, his HTLC also expires. The payment is canceled.

Payment Protocol with HTLC

1. To securely send d BTC to Bob via Charlie, Alice first asks Bob to provide $H_s = hash(s)$, the hash of a secret s known only to Bob.
2. Alice then sends a contract $HTLC(d, H_s, N)$ to Charlie. This contract allows Charlie to claim d BTC only if he can reveal the secret s. If Charlie does not reveal s before block N, the funds remain locked.
3. Charlie sends $HTLC(d, H_s, N - i)$ to Bob (with $1 \leq i$).

4. If Bob does not unlock the contract, the block $N - i$ will eventually be mined and the contract expires. Since Charlie does not learn the secret s, his HTLC also expires. The payment is canceled.

Why does the second HTLC have an expiration time of $N - i$?

HTLC and Commitment Transactions (1/2)

HTLCs are represented by commitment transactions.
Suppose the channel between Alice and Charlie is:

$$A(1) \longleftrightarrow (2)C$$

and Alice wants to send $\mathrm{HTLC}(0.1, H_s, N)$ to Charlie.
An HTLC is represented by the following commitment transactions:

Alice	Charlie
$\overset{0.9}{\Rightarrow}$ A(TL + Revoc)	$\overset{0.9}{\rightarrow}$ A
AC $\overset{2}{\rightarrow}$ C	AC $\overset{2}{\Rightarrow}$ C(TL + Revoc)
$\overset{0.1}{\rightarrow}$ $\mathrm{HTLC}_{\mathrm{OUT}}$	$\overset{0.1}{\rightarrow}$ $\mathrm{HTLC}_{\mathrm{IN}}$

The HTLC outputs (OUT and IN) are secured by H_s (for Charlie) and the timelock N (for Alice).

HTLC and Commitment Transactions (2/2)

If the secret is revealed before the timelock N expires, Alice and Charlie generate the commitment transactions corresponding to the state **after** the HTLC is accepted.

Alice	Charlie
$AC \overset{0.9}{\Rightarrow} A(TL + Revoc)$ $\overset{2.1}{\rightarrow} C$	$AC \overset{0.9}{\rightarrow} A$ $\overset{2.1}{\Rightarrow} C(TL + Revoc)$

Advantages and Disadvantages

What are the advantages and disadvantages of the Lightning Network?