

# Banks vs Blockchains

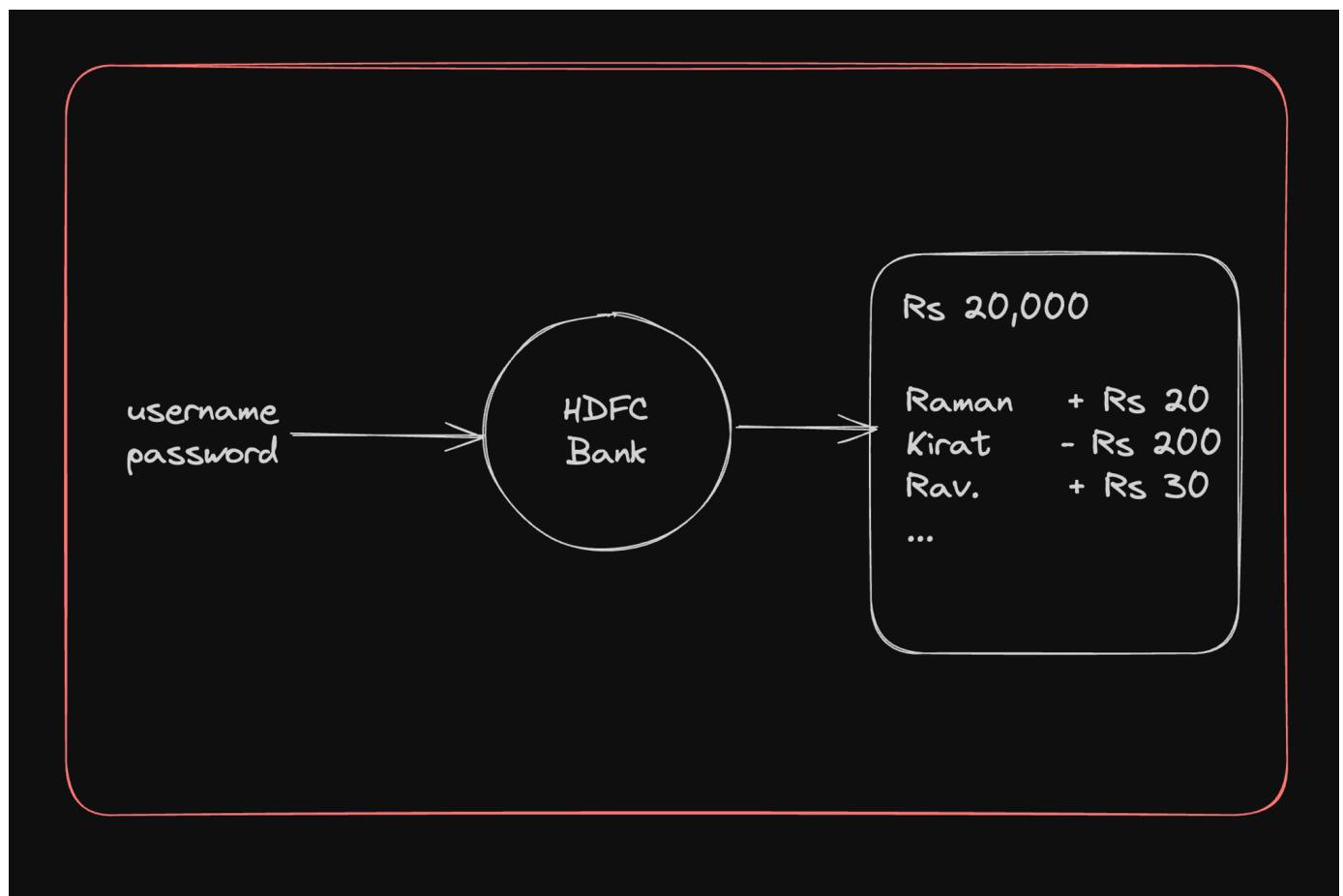
## Goal of today's class -

1. Create a simple web based wallet.
2. Look at the codebase of some wallets to see how they generate private keys

## How banks do Auth

In traditional banks, you have a `username` and `password` that are enough for you to

1. Look at your funds
2. Transfer funds
3. Look at your existing transactions



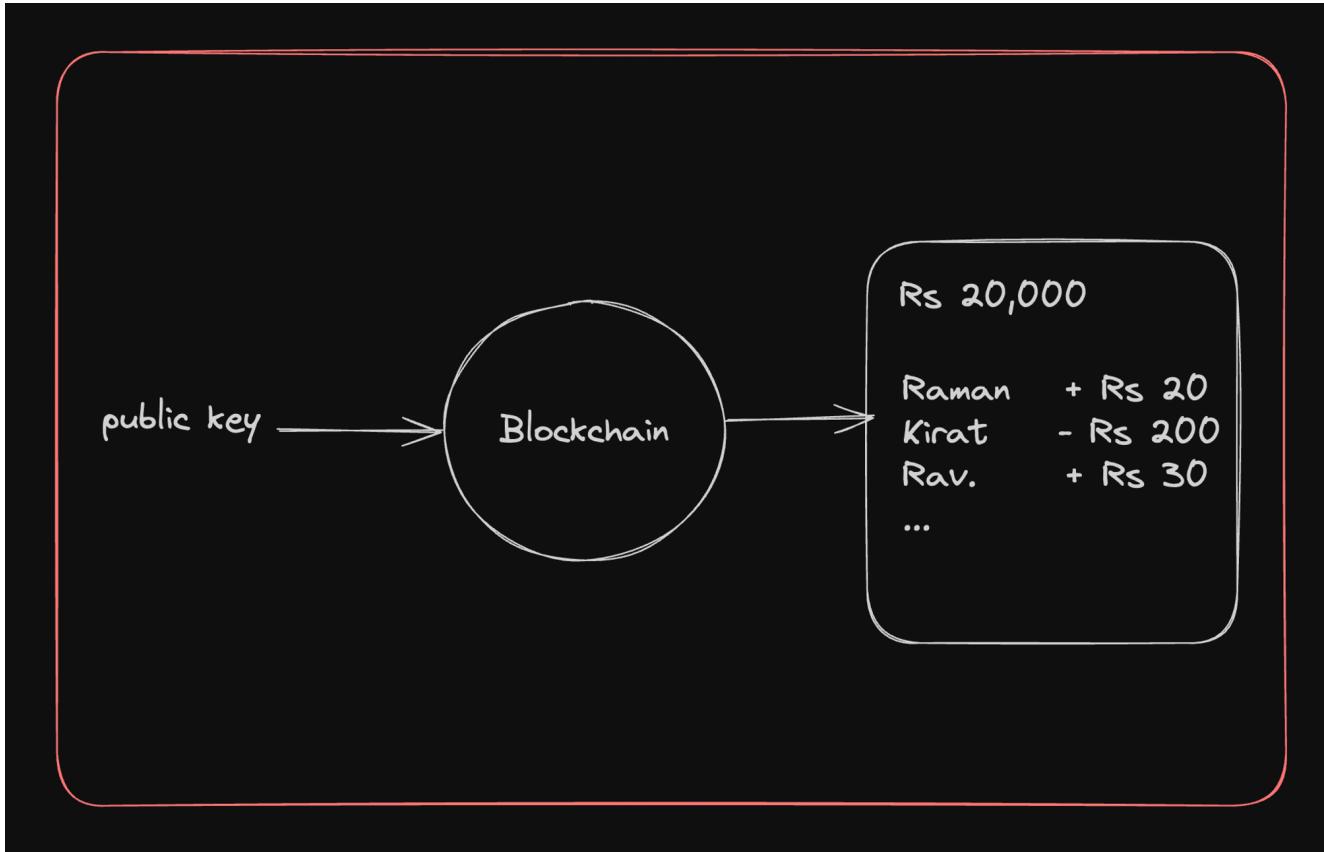
## How Blockchains do auth

If you ever want to create an **account** on a blockchain, you need to generate a **public-private keypair**.

## Public private Keypair

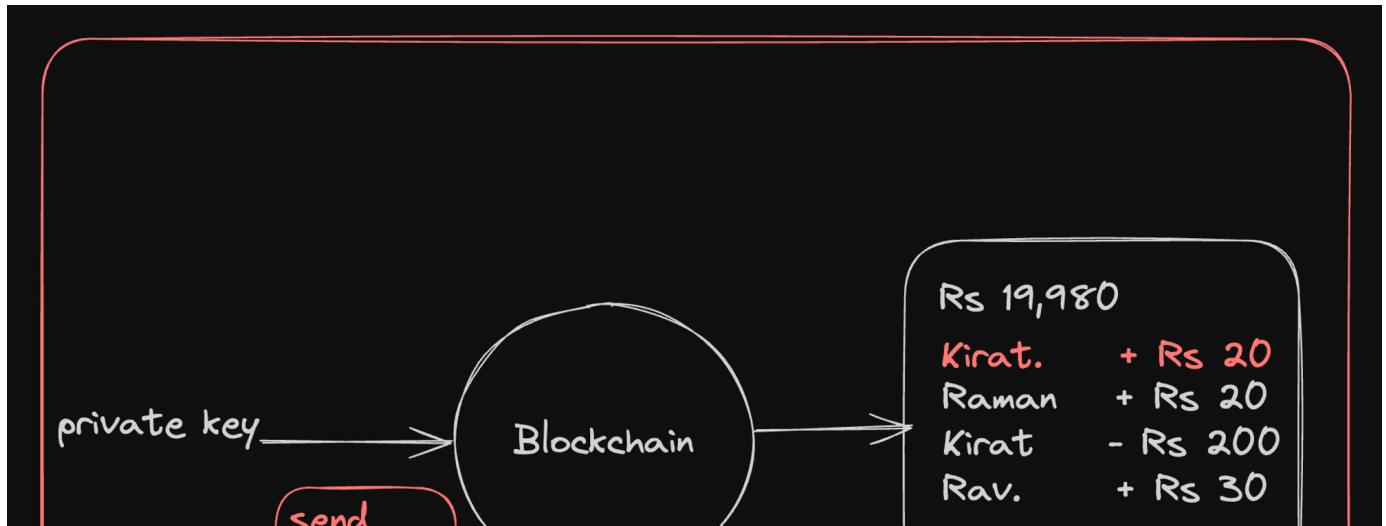
A public-private key pair is a set of two keys used in **asymmetric cryptography**. These two keys have the following characteristics:

**Public Key:** The public key is a string that can be shared openly.



For example - <https://etherscan.io/address/0xD9a657ACB3960DB92AaaA32942019bD3c473FCCB>

**Private key:** The private key is a secret string that must be kept confidential.



# Bits and bytes

## What is a Bit?

A bit is the smallest unit of data in a computer and can have one of two values: 0 or 1.

Think of a bit like a light switch that can be either off (0) or on (1).



## What is a byte?

A byte is a group of **8** bits. It's the standard unit of data used to represent a single character in memory. Since each bit can be either 0 or 1, a byte can have  $2^8$  (256) possible values, ranging from 0 to 255

## Assignment

What is the **11001010** converted to a **decimals** ?

## ▼ Answer

- $2^7: 1 \times 2^7 = 1 \times 128 = 128$
- $2^6: 1 \times 2^6 = 1 \times 64 = 64$
- $2^5: 0 \times 2^5 = 0 \times 32 = 0$
- $2^4: 0 \times 2^4 = 0 \times 16 = 0$
- $2^3: 1 \times 2^3 = 1 \times 8 = 8$
- $2^2: 0 \times 2^2 = 0 \times 4 = 0$
- $2^1: 1 \times 2^1 = 1 \times 2 = 2$
- $2^0: 0 \times 2^0 = 0 \times 1 = 0$   
= 202

## Representing bits and bytes in JS

- Bit

```
const x = 0;                                Copy
console.log(x);
```

- Byte

```
const x = 202                                Copy
console.log(x);
```

- Array of bytes

```
const bytes = [202, 244, 1, 23]                Copy
console.log(bytes);
```

## UInt8Array

A better way to represent an array of bytes is to use a `UInt8Array` in JS

```
let bytes = new Uint8Array([0, 255, 127, 128]);    Copy
console.log(bytes)
```

Why use `UIInt8Array` over `native arrays` ?

- They use less space. Every number takes 64 bits (8 bytes). But every value in a `UIInt8Array` takes 1 byte.
- `UInt8Array` Enforces constraints - It makes sure every element doesn't exceed 255.

## Assignment -

What do you think happens to the first element here? Does it throw an error?

```
let uint8Arr = new Uint8Array([0, 255, 127, 128]);      Copy
uint8Arr[1] = 300;
```

# Encodings

Bytes are cool but highly unreadable. Imagine telling someone

```
Hey, my name is 00101011101010101020      Copy
```

It's easier to `encode` data so it is more `human readable` . Some common encodings include -

1. Ascii
2. Hex
3. Base64
4. Base58

## Ascii

`1 character = 7 bits`

Every byte corresponds to a `text` on the `computer` .

Here is a complete list - [https://www.w3schools.com/charsets/ref\\_html\\_ascii.asp#:~:text=The%20ASCII%20Character%20Set&text=ASCII%20is%20a%207-bit](https://www.w3schools.com/charsets/ref_html_ascii.asp#:~:text=The%20ASCII%20Character%20Set&text=ASCII%20is%20a%207-bit) are all based on ASCII.

▼ Bytes to Ascii

```
function bytesToAscii(byteArray) {  
    return byteArray.map(byte => String.fromCharCode(byte)).join('');  
}  
  
// Example usage:  
const bytes = [72, 101, 108, 108, 111]; // Corresponds to "Hello"  
const asciiString = bytesToAscii(bytes);  
console.log(asciiString); // Output: "Hello"
```

[Copy](#)

## ▼ Ascii to bytes

```
function asciiToBytes(asciiString) {  
    const byteArray = [];  
    for (let i = 0; i < asciiString.length; i++) {  
        byteArray.push(asciiString.charCodeAt(i));  
    }  
    return byteArray;  
}  
  
// Example usage:  
const ascii = "Hello";  
const byteArray = asciiToBytes(ascii);  
console.log(byteArray); // Output: [72, 101, 108, 108, 111]
```

[Copy](#)

## ▼ Uint8Array to ascii

```
function bytesToAscii(byteArray) {  
    return new TextDecoder().decode(byteArray);  
}  
  
// Example usage:  
const bytes = new Uint8Array([72, 101, 108, 108, 111]); // Corresponds to "Hello"  
const asciiString = bytesToAscii(bytes);  
console.log(asciiString); // Output: "Hello"
```

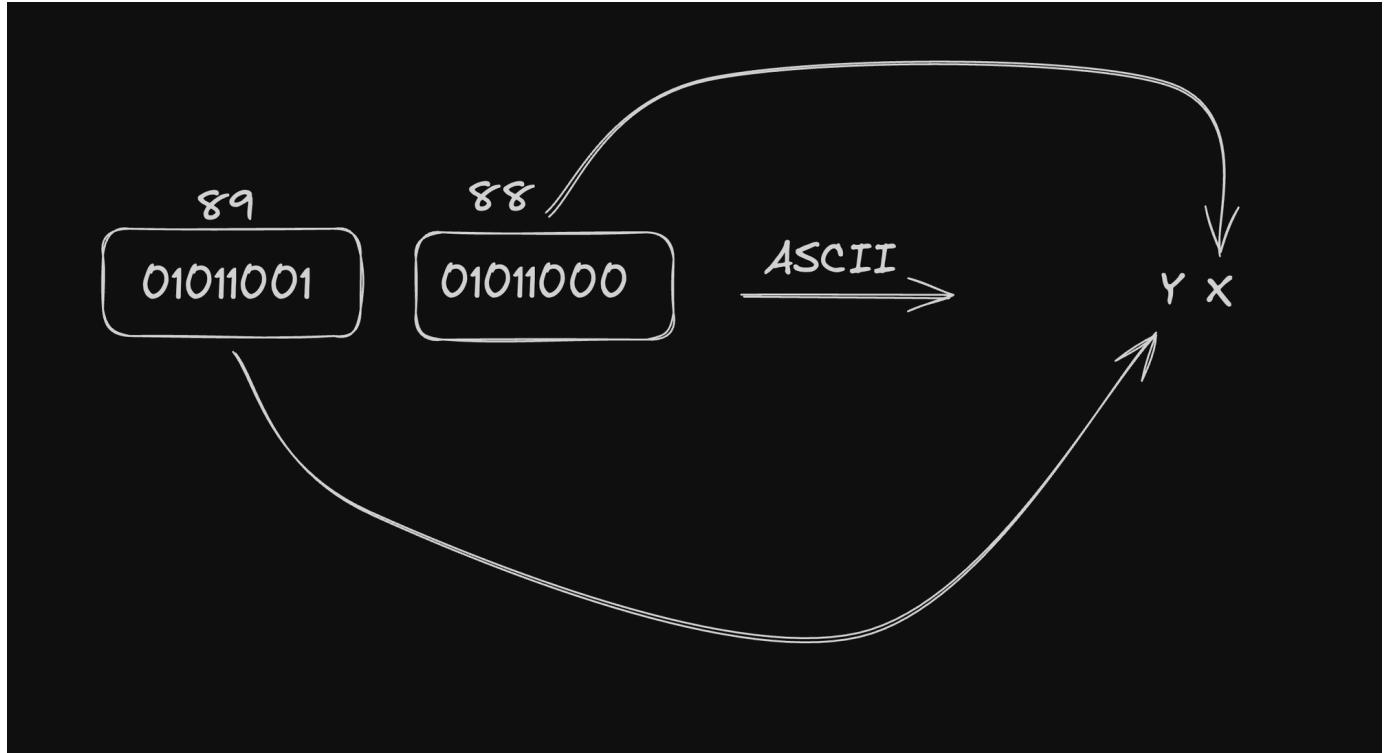
[Copy](#)

## ▼ Ascii to Uint8Array

```
function asciiToBytes(asciiString) {  
    return new Uint8Array([...asciiString].map(char => char.charCodeAt(0)));  
}  
  
// Example usage:
```

[Copy](#)

```
const ascii = "Hello";
const byteArray = asciiToBytes(ascii);
console.log(byteArray); // Output: Uint8Array(5) [72, 101, 108, 108,
```



## Hex

1 character = 4 bits

A single hex character can be any of the 16 possible values: 0-9 and A-F.

### ▼ Array to hex

```
function arrayToHex(byteArray) {
  let hexString = '';
  for (let i = 0; i < byteArray.length; i++) {
    hexString += byteArray[i].toString(16).padStart(2, '0');
  }
  return hexString;
}

// Example usage:
const byteArray = new Uint8Array([72, 101, 108, 108, 111]); // Corre
const hexString = arrayToHex(byteArray);
console.log(hexString); // Output: "48656c6c6f"
```

Copy

### ▼ Hex to array

```

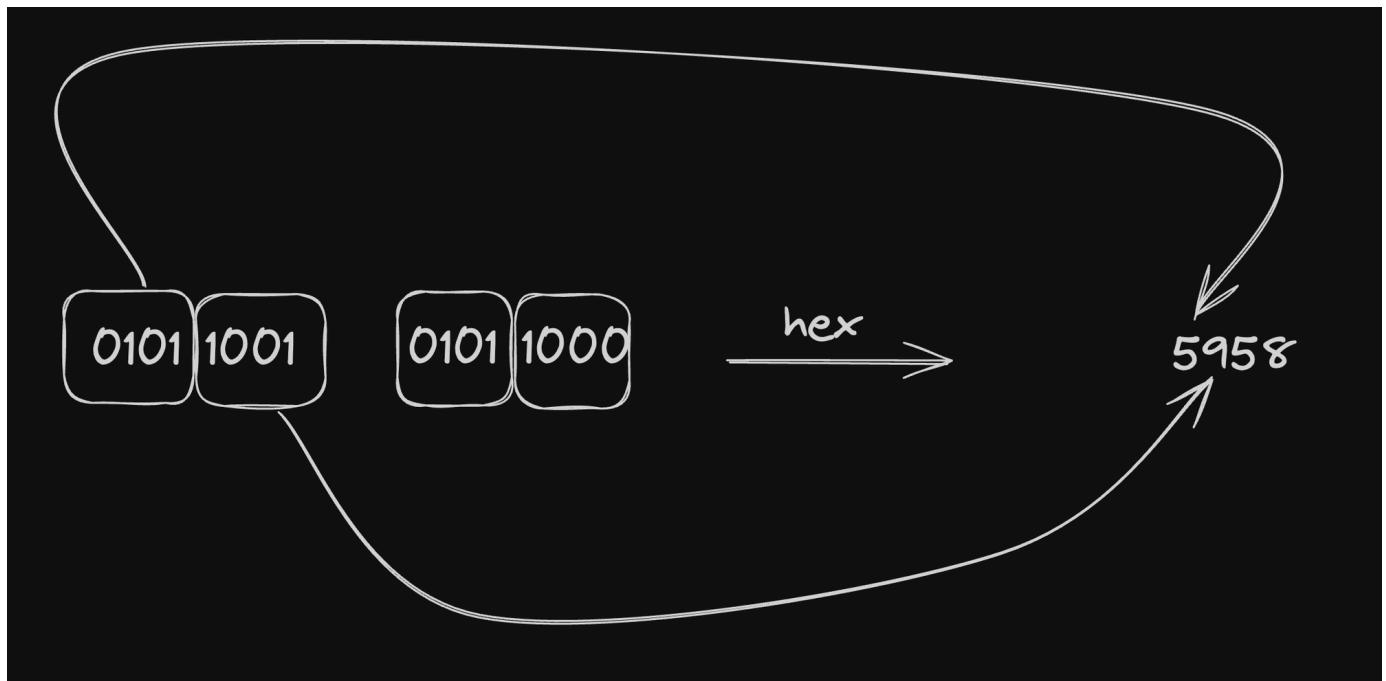
function hexToArray(hexString) {
    const byteArray = new Uint8Array(hexString.length / 2);
    for (let i = 0; i < byteArray.length; i++) {
        byteArray[i] = parseInt(hexString.substr(i * 2, 2), 16);
    }
    return byteArray;
}

// Example usage:
const hex = "48656c6c6f";
const byteArrayFromHex = hexToArray(hex);
console.log(byteArrayFromHex); // Output: Uint8Array(5) [72, 101, 101, 101, 101]

```

Copy

Ref - [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/parselnt](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/parselnt)



## Base64

1 character = 6 bits

Base64 encoding uses 64 different characters ( A-Z , a-z , 0-9 , + , / ), which means each character can represent one of 64 possible values.

<https://www.base64encode.org/>

<https://www.base64decode.org/>

▼ Encode

```
const uint8Array = new Uint8Array([72, 101, 108, 108, 111]);           Copy
const base64Encoded = Buffer.from(uint8Array).toString("base64");
console.log(base64Encoded);
```

## Base58

It is similar to Base64 but uses a different set of characters to avoid visually similar characters and to make the encoded output more user-friendly

Base58 uses 58 different characters:

- Uppercase letters: `A-Z` (excluding `I` and `O`)
- Lowercase letters: `a-z` (excluding `l`)
- Numbers: `1-9` (excluding `0`)

### ▼ Encode

```
const bs58 = require('bs58');                                         Copy

function uint8ArrayToBase58(uint8Array) {
  return bs58.encode(uint8Array);
}

// Example usage:
const byteArray = new Uint8Array([72, 101, 108, 108, 111]); // Corre
const base58String = uint8ArrayToBase58(byteArray);
console.log(base58String); // Output: Base58 encoded string
```

### ▼ Decode

```
const bs58 = require('bs58');                                         Copy

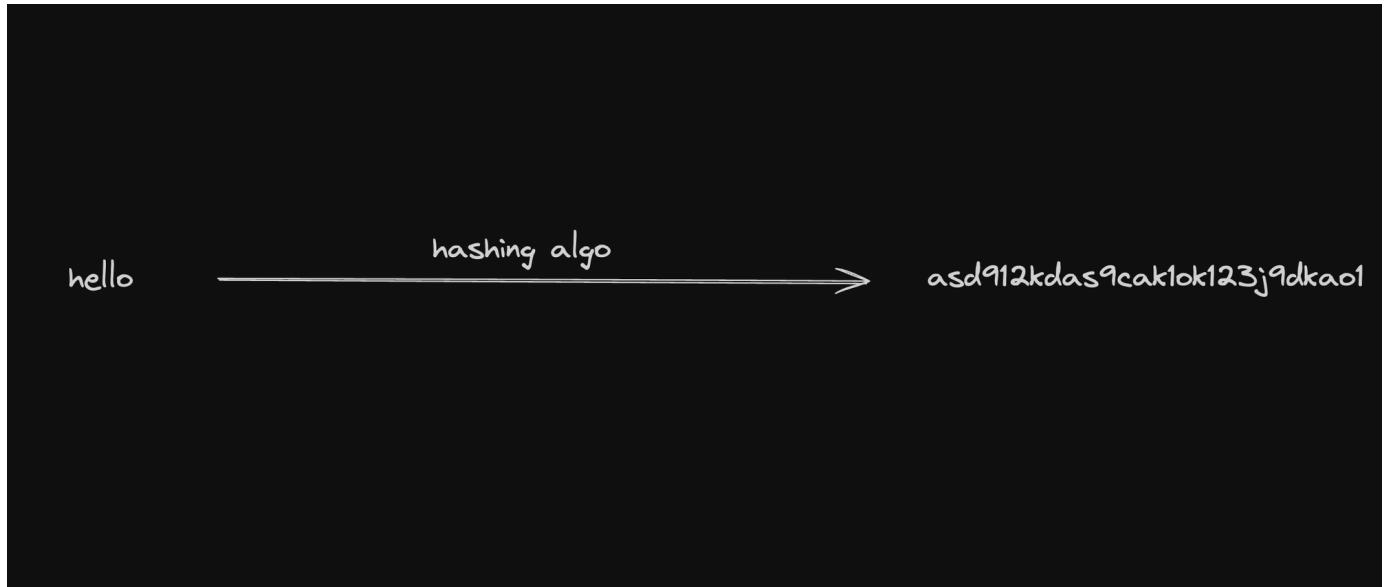
function base58ToUint8Array(base58String) {
  return bs58.decode(base58String);
}

// Example usage:
const base58 = base58String; // Use the previously encoded Base58 st
const byteArrayFromBase58 = base58ToUint8Array(base58);
console.log(byteArrayFromBase58); // Output: Uint8Array(5) [72, 101,
```

# Hashing vs encryption

## Hashing

Hashing is a process of converting data (like a file or a message) into a fixed-size string of characters, which typically appears random.



Common hashing algorithms - SHA-256, MD5

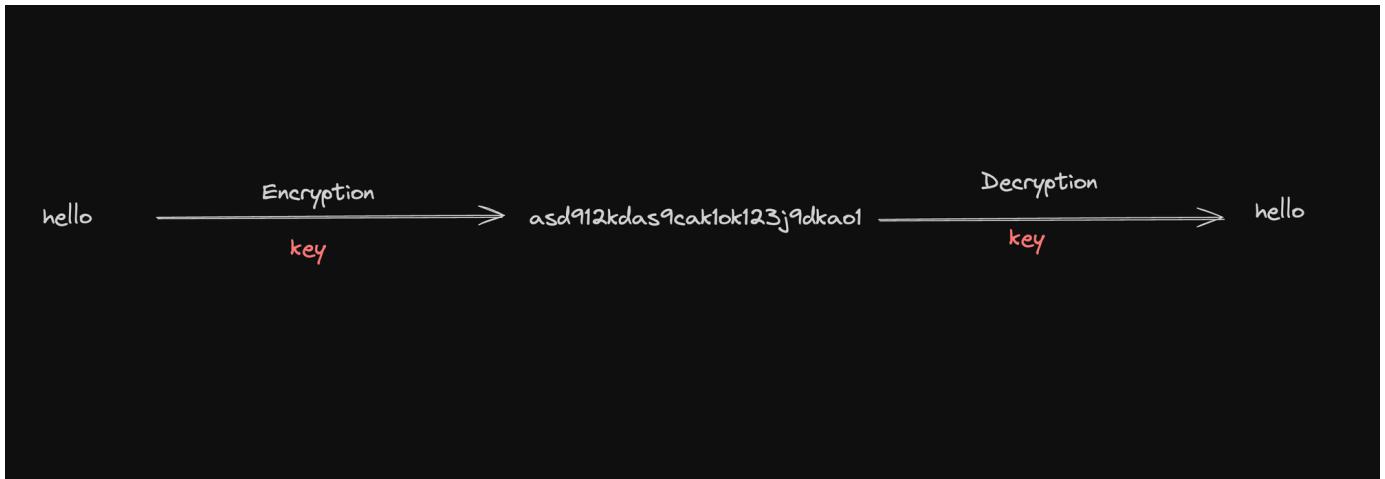
## Encryption

Encryption is the process of converting plaintext data into an unreadable format, called ciphertext, using a specific algorithm and a key. The data can be decrypted back to its original form only with the appropriate key.

### Key Characteristics:

- **Reversible:** With the correct key, the ciphertext can be decrypted back to plaintext.
- **Key-dependent:** The security of encryption relies on the secrecy of the key.
- **Two main types:**
  - **Symmetric encryption:** The same key is used for both encryption and decryption.
  - **Asymmetric encryption:** Different keys are used for encryption (public key) and decryption (private key).

## Symmetric encryption



## ▼ Code

```
const crypto = require('crypto');

// Generate a random encryption key
const key = crypto.randomBytes(32); // 32 bytes = 256 bits
const iv = crypto.randomBytes(16); // Initialization vector (IV)

// Function to encrypt text
function encrypt(text) {
    const cipher = crypto.createCipheriv('aes-256-cbc', key, iv);
    let encrypted = cipher.update(text, 'utf8', 'hex');
    encrypted += cipher.final('hex');
    return encrypted;
}

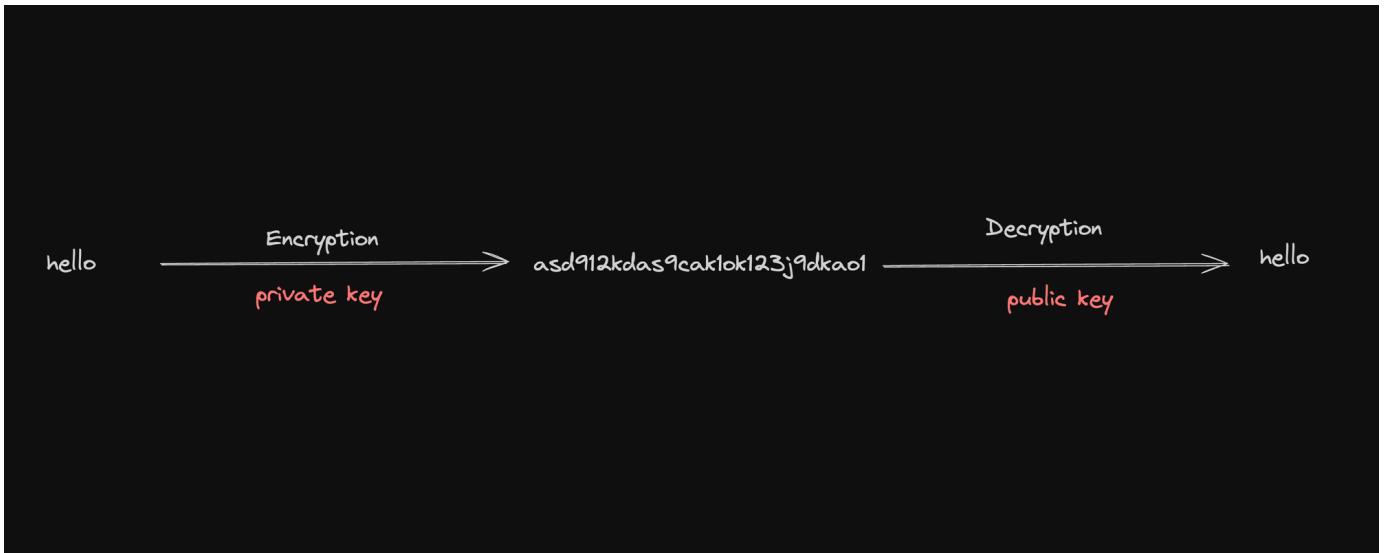
// Function to decrypt text
function decrypt(encryptedText) {
    const decipher = crypto.createDecipheriv('aes-256-cbc', key, iv)
    let decrypted = decipher.update(encryptedText, 'hex', 'utf8');
    decrypted += decipher.final('utf8');
    return decrypted;
}

// Example usage
const textToEncrypt = 'Hello, World!';
const encryptedText = encrypt(textToEncrypt);
const decryptedText = decrypt(encryptedText);

console.log('Original Text:', textToEncrypt);
console.log('Encrypted Text:', encryptedText);
console.log('Decrypted Text:', decryptedText);
```

Copy

## Asymmetric encryption

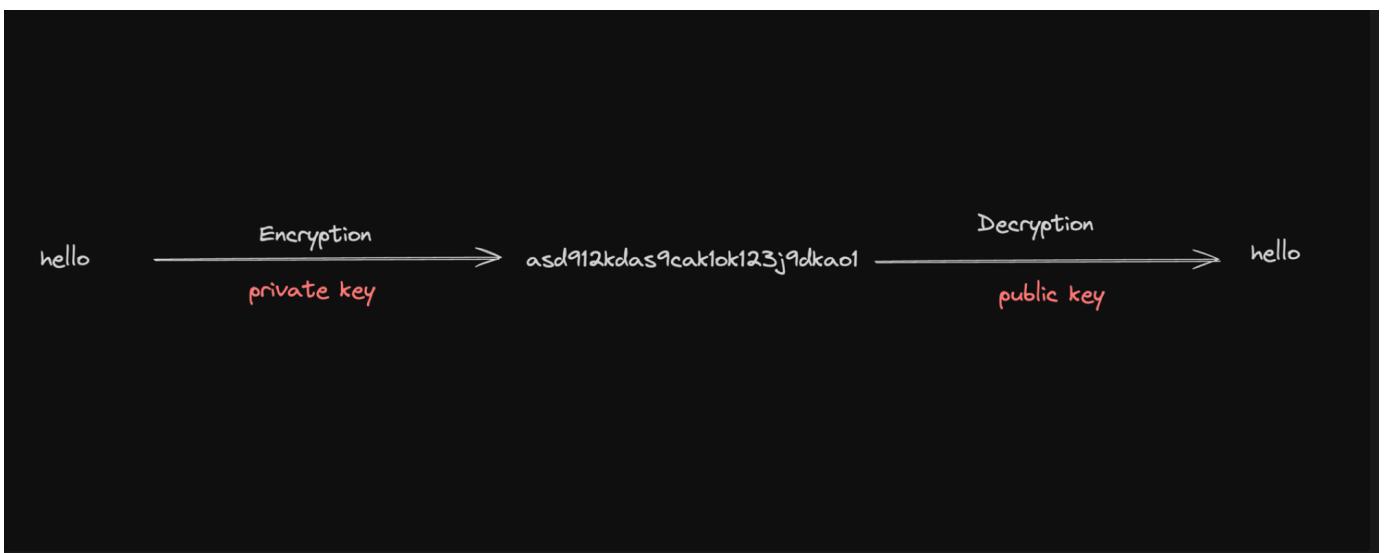


# Asymmetric Encryption

Asymmetric encryption, also known as public-key cryptography, is a type of encryption that uses a pair of keys: a public key and a private key. The keys are mathematically related, but it is computationally infeasible to derive the **private key** from the **public key**.

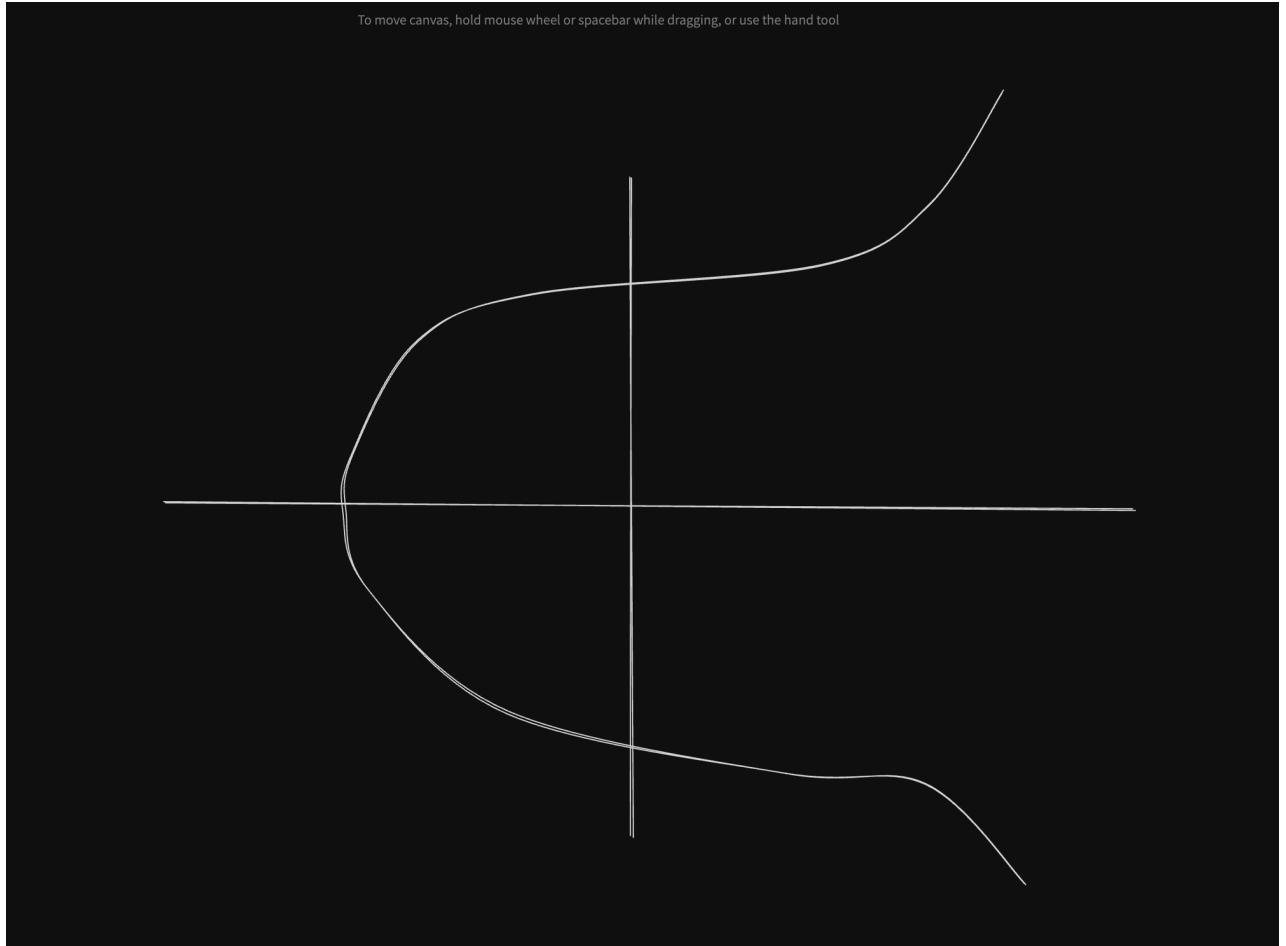
**Public Key:** The public key is a string that can be shared openly

**Private Key:** The private key is a secret cryptographic code that must be kept confidential. It is used to decrypt data encrypted with the corresponding public key or to create digital signatures.



## Common Asymmetric Encryption Algorithms:

1. RSA - Rivest–Shamir–Adleman
2. ECC - Elliptic Curve Cryptography (ECDSA) - ETH and BTC
3. EdDSA - Edwards-curve Digital Signature Algorithm - SOL



How elliptic curves work - <https://www.youtube.com/watch?v=NF1pwjL9-DE&>

## Common elliptic curves

1. secp256k1 - BTC and ETH
2. ed25519 - SOL

Few usecases of public key cryptography -

1. SSL/TLS certificates
2. SSH keys to connect to servers/push to github
3. Blockchains and cryptocurrencies

# Creating a public/private keypair

Let's look at various ways of creating public/private keypairs, signing messages and verifying them

## EdDSA - Edwards-curve Digital Signature Algorithm - ED25519

### ▼ Using [@noble/ed25519](#)

```
import * as ed from "@noble/ed25519";Copy

async function main() {
  // Generate a secure random private key
  const privKey = ed.utils.randomPrivateKey();

  // Convert the message "hello world" to a Uint8Array
  const message = new TextEncoder().encode("hello world");

  // Generate the public key from the private key
  const pubKey = await ed.getPublicKeyAsync(privKey);

  // Sign the message
  const signature = await ed.signAsync(message, privKey);

  // Verify the signature
  const isValid = await ed.verifyAsync(signature, message, pubKey);

  // Output the result
  console.log(isValid); // Should print `true` if the signature is valid
}

main();
```

### ▼ Using [@solana/web3.js](#)

```

import { Keypair } from "@solana/web3.js";
import nacl from "tweetnacl";

// Generate a new keypair
const keypair = Keypair.generate();

// Extract the public and private keys
const publicKey = keypair.publicKey.toString();
const secretKey = keypair.secretKey;

// Display the keys
console.log("Public Key:", publicKey);
console.log("Private Key (Secret Key):", secretKey);

// Convert the message "hello world" to a Uint8Array
const message = new TextEncoder().encode("hello world");

const signature = nacl.sign.detached(message, secretKey);
const result = nacl.sign.detached.verify(
    message,
    signature,
    keypair.publicKey.toBytes(),
);

console.log(result);

```

Copy

## ECDSA (Elliptic Curve Digital Signature Algorithm) - secp256k1

### ▼ Using `@noble/secp256k1`

```

import * as secp from "@noble/secp256k1";

async function main() {
    const privKey = secp.utils.randomPrivateKey(); // Secure random pr
    // sha256 of 'hello world'
    const msgHash =
        "b94d27b9934d3e08a52e52d7da7dabfac484efe37a5380ee9088f7ace2efcde
    const pubKey = secp.getPublicKey(privKey);
    const signature = await secp.signAsync(msgHash, privKey); // Sync
    const isValid = secp.verify(signature, msgHash, pubKey);
    console.log(isValid);
}

```

Copy

```
main();
```

## ▼ Using ethers

```
import { ethers } from "ethers";Copy

// Generate a random wallet
const wallet = ethers.Wallet.createRandom();

// Extract the public and private keys
const publicKey = wallet.address;
const privateKey = wallet.privateKey;

console.log("Public Key (Address):", publicKey);
console.log("Private Key:", privateKey);

// Message to sign
const message = "hello world";

// Sign the message using the wallet's private key
const signature = await wallet.signMessage(message);
console.log("Signature:", signature);

// Verify the signature
const recoveredAddress = ethers.verifyMessage(message, signature);

console.log("Recovered Address:", recoveredAddress);
console.log("Signature is valid:", recoveredAddress === publicKey);
```

# How to transactions work on the blockchain?

Ref - <https://andersbrownworth.com/blockchain/>

## User side

1. User first creates a **public/private** keypair
2. They create a **transaction** that they want to do (send Rs 50 to Alice). The transaction includes all necessary details like the recipient's address, the amount and some blockchain specific parameters (for eg - latestBlockHash in case of solana)
3. They hash the **transaction**
4. They **sign** the transaction using their private key
5. They send the **raw transaction**, **signature** and their **public key** to a node on the blockchain.

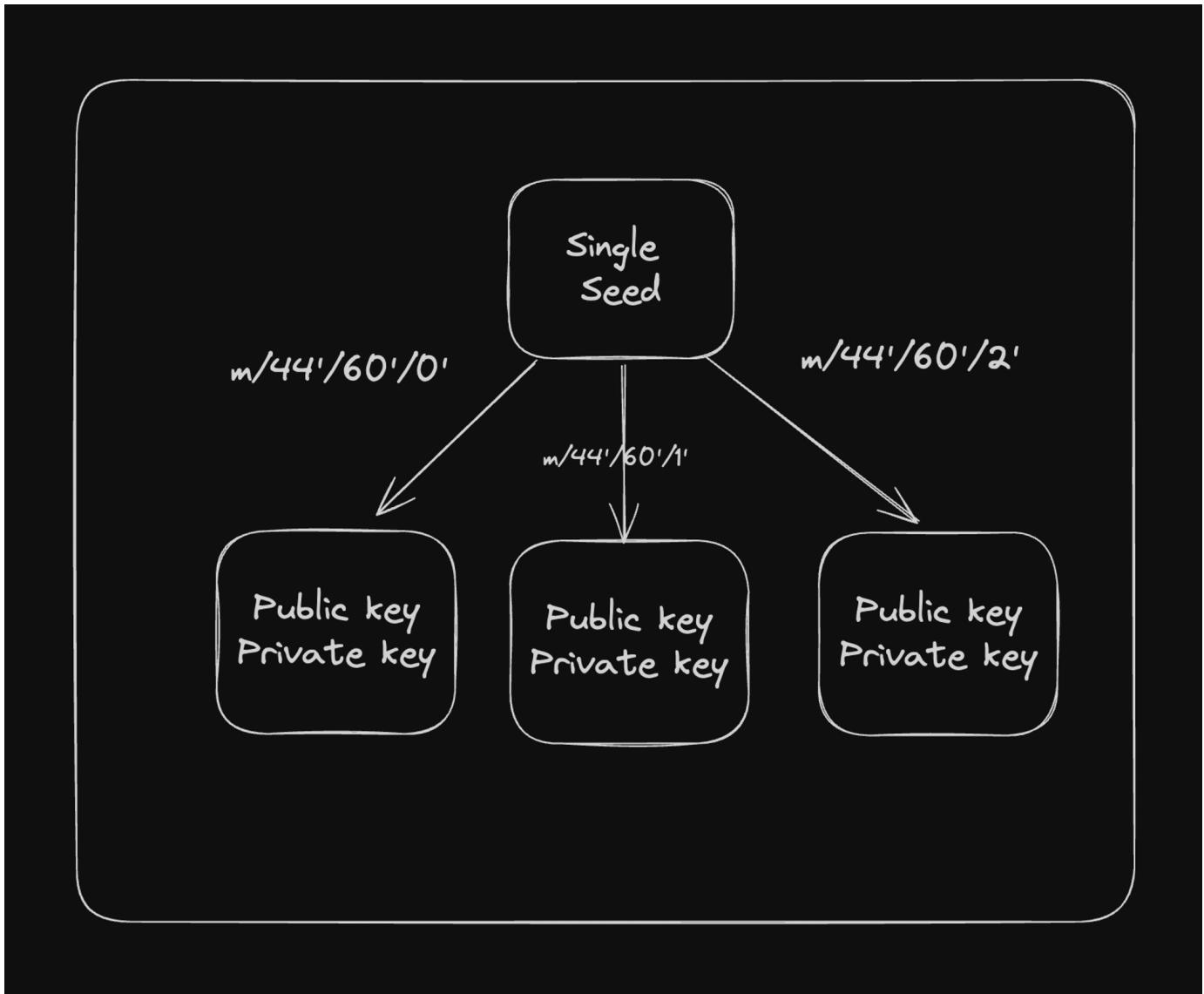
## Miner

1. Hashes the original message to generate a **hash**
2. Verifies the **signature** using the users **public key** and the **hash** generated in step 1
3. Transaction validation - The miner/validator checks additional aspects of the transaction, such as ensuring the user has sufficient funds
4. If everything checks out, adds the transaction to the block

# Hierarchical Deterministic (HD) Wallet

Hierarchical Deterministic (HD) wallets are a type of wallet that can generate a tree of key pairs from a single seed. This allows for the generation of multiple addresses from a single root seed,

providing both security and convenience.



## Problem

You have to maintain/store multiple `public private` keys if you want to have multiple wallets.

## Solution - BIP-32

Bitcoin Improvement Proposal 32 (BIP-32) provided the solution to this problem in 2012. It was proposed by Pieter Wuille, a Bitcoin Core developer, to simplify the recovery process of crypto wallets. BIP-32 introduced a hierarchical tree-like structure for wallets that allowed you to manage multiple accounts much more easily than was previously possible. It's essentially a standardized way to derive private and public keys from a master seed.

# How to create a wallet

## Mnemonics

A mnemonic phrase (or seed phrase) is a human-readable string of words used to generate a cryptographic seed. BIP-39 (Bitcoin Improvement Proposal 39) defines how mnemonic phrases are generated and converted into a seed.

Ref - <https://github.com/bitcoin/bips/blob/master/bip-0039/english.txt>

Where this is done in Backpack - <https://github.com/coral-xyz/backpack/blob/master/packages/app-extension/src/components/common/Account/MnemonicInput.tsx#L143>

### ▼ Code

```
import { generateMnemonic } from 'bip39';

// Generate a 12-word mnemonic
const mnemonic = generateMnemonic();
console.log('Generated Mnemonic:', mnemonic);
```

Copy

# Secret Recovery Phrase

Save these words in a safe place.

[Read the warnings again](#)

1 elevator

2 bus

3 jump

4 toilet

5 flag

6 dove

7 blouse

8 already

9 robot

10 dentist

11 enlist

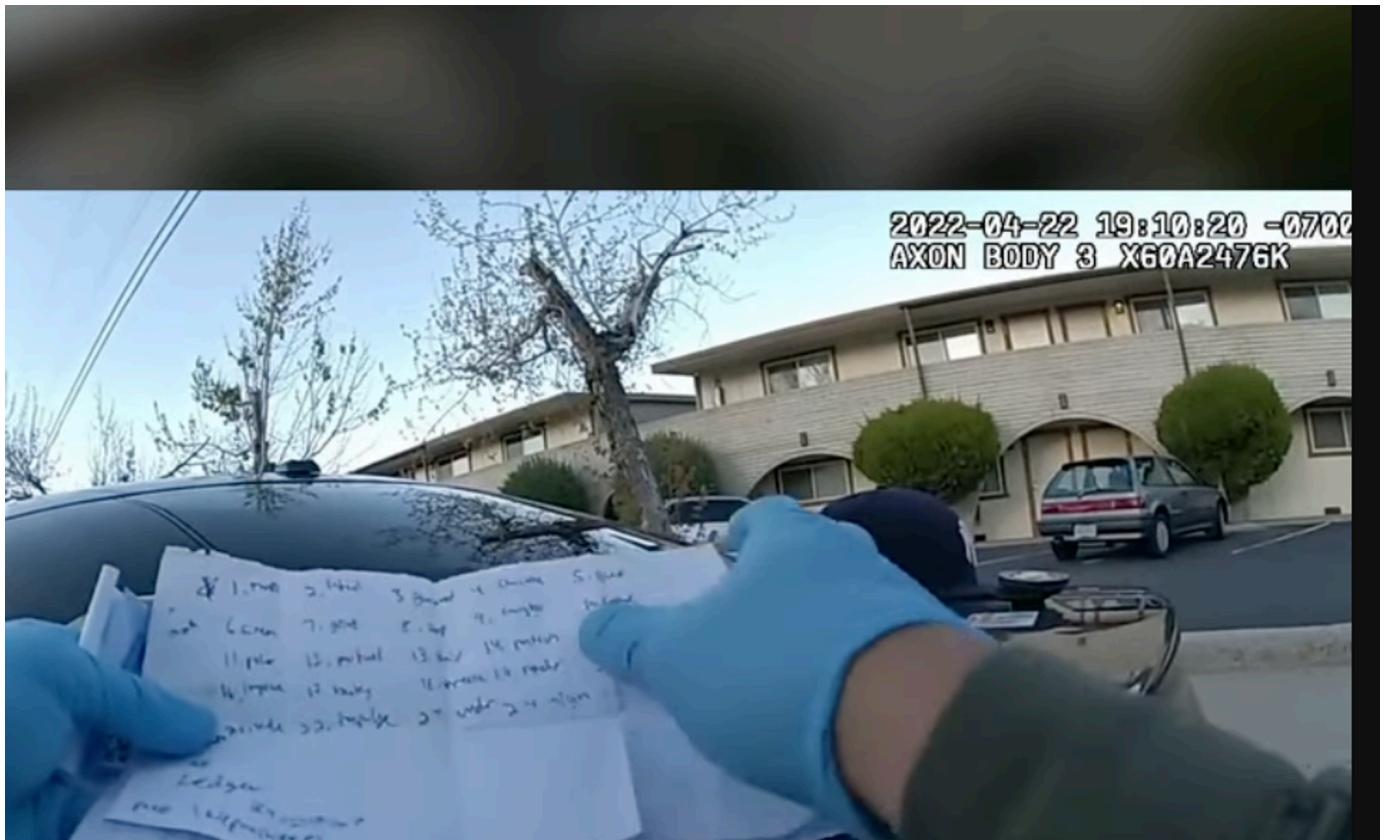
12 inflict

Click anywhere on this card to copy

I saved my secret recovery phrase

Next





Ref - <https://www.youtube.com/shorts/ojBlcnPOk6k>

## Seed phrase

The seed is a binary number derived from the mnemonic phrase.

```
import { generateMnemonic, mnemonicToSeedSync } from "bip39";      Copy

const mnemonic = generateMnemonic();
console.log("Generated Mnemonic:", mnemonic);
const seed = mnemonicToSeedSync(mnemonic);
```

Ref - <https://github.com/coral-xyz/backpack/blob/master/packages/secure-background/src/services/svm/keyring.ts#L131>

## Derivation paths



- Derivation paths specify a systematic way to derive various keys from the master seed.
- They allow users to recreate the same set of addresses and private keys from the seed across different wallets, ensuring interoperability and consistency. (for example if you ever want to

# Assignments

## Adding support for ETH

Given we just derived a few public keys in SOL, can you try doing the same for ETH?

Again for reference, this is how backpack does it - <https://github.com/coral-xyz/backpack/blob/master/packages/secure-background/src/services/evm/util.ts#L3>

## Creating a web based wallet

Create a simple web based wallet where someone can come and create a mnemonic, add multiple wallets and see the public key associated with each wallet