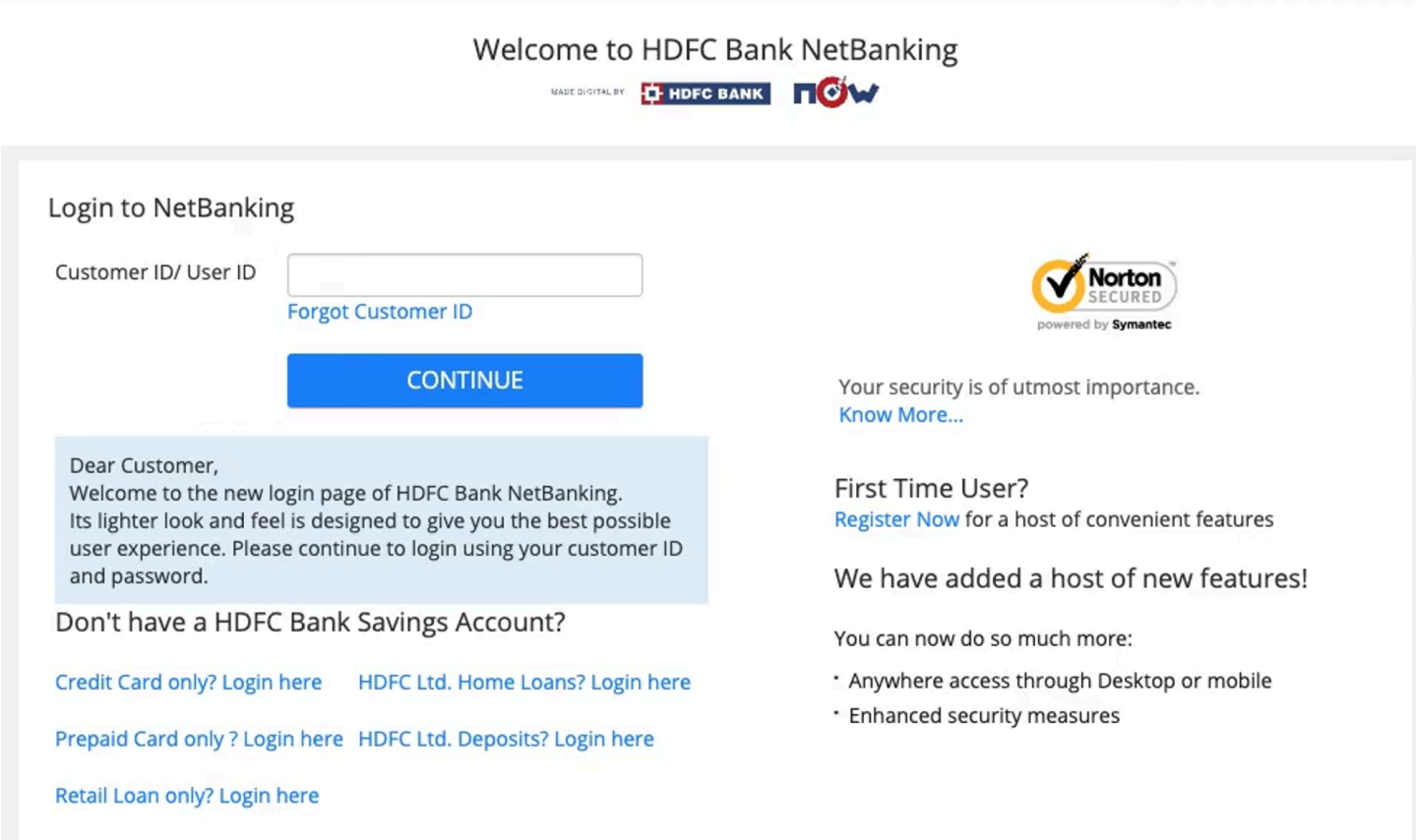
Week 2 - Public Key Cryptography [9 Aug 2024]

> **Timing** - Every Friday 8:00pm to 10:00pm IST
> Class Slides - Public Key Cryptography
> Class Video - Take your development skills from 0 to 100 and join the 100xdevs community
> Assignment - Creating a web based wallet

> **Syllabus** - Notion – The all-in-one workspace for your notes, tasks, wikis, and databases. (100xdevs.com)

How Banks Do Authentication:



- **Username and Password:**
  - Traditional banks rely on a username and password for authentication.

- These credentials allow you to:
  - View your funds.
  - Transfer funds.
  - Review your transaction history.

How Blockchains Do Authentication:

Public-Private Keypair:

Blockchain accounts are secured through a **public-private keypair**.

- A public-private keypair consists of two keys used in asymmetric cryptography.

Public Key:

- The public key is a string that can be shared openly with anyone.
- It acts like your "account number" on the blockchain.
- Example: Ethereum Address on Etherscan

Private Key:

- The private key is a secret string that must be kept confidential.
- It is used to sign transactions and prove ownership of the associated public key.
- Never share your private key with anyone.

Bits and Bytes

▼ Why Learning this?

Private keys are stored in a certain format, whenever you use them or generate them.

They are stored/generated in bits and bytes.

Bits and Bytes understanding, and various encoding-decoding helps when dealing and understanding private keys.

What is a Bit?

- A bit is the smallest unit of data in a computer.
- It can have one of two values: `0` or `1`.
- All programs and code you write are eventually converted to `0's` and `1's`.
- **Analogy:**
  - Think of a bit like a light switch that can either be off (`0`) or on (`1`).
- **Bit Representation in JavaScript:**

```
const x = 0;
console.log(x); // Outputs: 0
```

  - Here, `x` represents a single bit with a value of `0`.

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.
- **Possible Values:**
  - Since each bit can be either `0` or `1`, a byte can have 2^8 (256) possible values, ranging from `0` to `255`.
  - Example: The binary sequence `11001010` represents a specific value in decimal (we'll cover this in the assignment below).
- **Byte Representation:**

```
const x = 202;
console.log(x); // Outputs: 202
```

  - Here, `x` is a byte, representing the decimal value `202`, which is equivalent to `11001010` in binary.
- **Array of Bytes:**

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

- This is an array containing multiple bytes.

Using `UInt8Array` in JavaScript:

- **Definition:**
  - `UInt8Array` is a typed array in JavaScript that represents an array of 8-bit unsigned integers (bytes).
- **Advantages:**
  - **Memory Efficiency:** Uses less space; each value takes only 1 byte.
  - **Constraints:** Ensures that values don't exceed `255`, which is the maximum value a byte can hold.
- **Example:**

```
let bytes = new Uint8Array([0, 255, 127, 128]);
console.log(bytes); // Outputs: Uint8Array(4) [ 0, 255, 127, 128 ]
```

  - This code creates a `UInt8Array` with four bytes, ensuring that each value stays within the valid byte range.
- **Example:**

```
const binaryRepresentation = new TextEncoder().encode("h");
console.log(binaryRepresentation); // Uint8Array(1)[104]
```

Why Use `UInt8Array` Over Native Arrays?

- **Memory Efficiency:**
  - Native arrays in JavaScript store numbers using 64 bits (8 bytes) per number, regardless of the actual size of the number.
  - `UInt8Array` stores each number using only 1 byte, which is sufficient for values between `0` and `255`.
- **Constraints:**
  - `UInt8Array` enforces that each element doesn't exceed `255`, preventing potential overflow errors.

▼ **Assignment:**

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

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

▼ **Answer:**

- When you try to assign `300` to the second element, `UInt8Array` will not throw an error but will instead wrap the value around, storing `300 % 256 = 44`. The `UInt8Array` enforces the 8-bit limit, so the value will be truncated to fit within the range `0-255`.

▼ **Assignment:**

- **Question:** What is `11001010` converted to in decimal?
- **Answer:** `202`

> 💡 Binary to Decimal Converter (rapidtables.com)

Encodings

- When working with computers, data is often represented in a format that is not human-readable, such as binary or bytes.
- Encoding is the process of converting this data into a more readable format.
- Some common encodings include ASCII, Hex, Base64, and Base58.
- These encodings help us represent binary data in a more understandable way.

1. ASCII (American Standard Code for Information Interchange)

- **1 character = 7 bits**
- ASCII is one of the oldest encodings used to represent text in computers. Each character in ASCII corresponds to a specific number (ranging from 0 to 127), which is represented in binary.
- For example, the letter 'A' is represented by the number 65 in ASCII, which is `01000001` in binary.

▼ **Converting 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"
```

▼ Converting 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]
```

▼ Using `UInt8Array` for 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"
```

▼ ASCII to `UInt8Array`

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

// Example usage:
const ascii = "Hello";
const byteArray = asciiToBytes(ascii);
console.log(byteArray); // Output: Uint8Array(5) [72, 101, 108, 108, 111]
```

> 💡 ASCII table - Table of ASCII codes, characters and symbols (ascii-code.com)
> HTML ASCII Reference (w3schools.com)

2. Hexadecimal (Hex)

- **1 character = 4 bits**
- Hexadecimal is a base-16 encoding system that uses 16 characters: `0-9` and `A-F`. It is commonly used in programming and digital systems to represent binary data in a more compact and readable format.
- Each hex digit represents four bits (a nibble), and two hex digits represent one byte.

▼ **Converting 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]); // Corresponds to "Hello"
const hexString = arrayToHex(byteArray);
console.log(hexString); // Output: "48656c6c6f"
```

▼ **Converting 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, 108, 108, 111]
```

> 💡 Hex (Base16) encoder & decoder, a simple online tool 🧰 (hexator.com)

3. Base64

- **1 character = 6 bits**
- Base64 is an encoding scheme that represents binary data in an ASCII string format. It uses 64 different characters (`A-Z` , `a-z` , `0-9` , `+` , `/`). It is commonly used in data transfer, encoding images, and storing complex data as text.

▼ **Encoding to Base64**

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

> 💡 Base64 Encode/Decode
> Base64 Decode/Encode

4. Base58

- Base58 is similar to Base64 but uses a different set of characters to avoid visually similar characters (e.g., `0` and `O` , `l` and `1`) and to make the encoded output more user-friendly.
- It is often used in Bitcoin and other cryptocurrencies for encoding addresses and other data.

▼ **Encoding to Base58**

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

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

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

▼ **Decoding from Base58**

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

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

// Example usage:
const base58 = base58String; // Use the previously encoded Base58 string
const byteArrayFromBase58 = base58ToUint8Array(base58);
console.log(byteArrayFromBase58); // Output: Uint8Array(5) [72, 101, 108, 108, 111]
```

Hashing vs Encryption

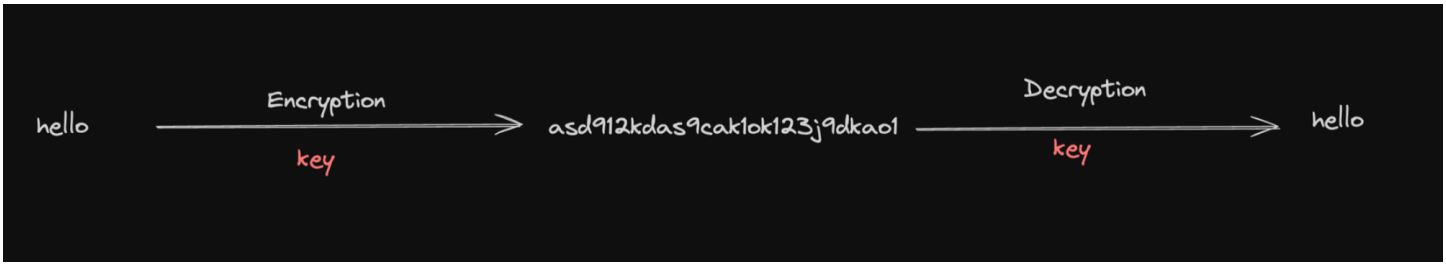
Hashing

- Hashing converts data into a fixed-size string of characters, known as a hash.
- **Key points**:
  - **Deterministic**: The same input will always produce the same hash.
  - **Fixed Size**: Regardless of the input size, the output hash will always be the same length.
  - **One-Way Function**: Hashes cannot be reversed to retrieve the original input data.
  - **Collision Resistance**: It is computationally difficult to find two different inputs that produce the same hash.
- **Common Hashing Algorithms**:
  - **SHA-256**: Widely used in blockchain technology, ensuring data integrity.
  - **MD5**: Once popular for checksums, now considered insecure due to vulnerabilities.

Encryption

- Encryption converts plaintext into ciphertext using an algorithm and a key.
- **Key points**:
  - **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.

Types of Encryptions:



1. Symmetric Encryption:

- **Definition**: The same key is used for both encryption and decryption.
- **Common Algorithms**:
  - **AES (Advanced Encryption Standard)**
  - **DES (Data Encryption Standard)**

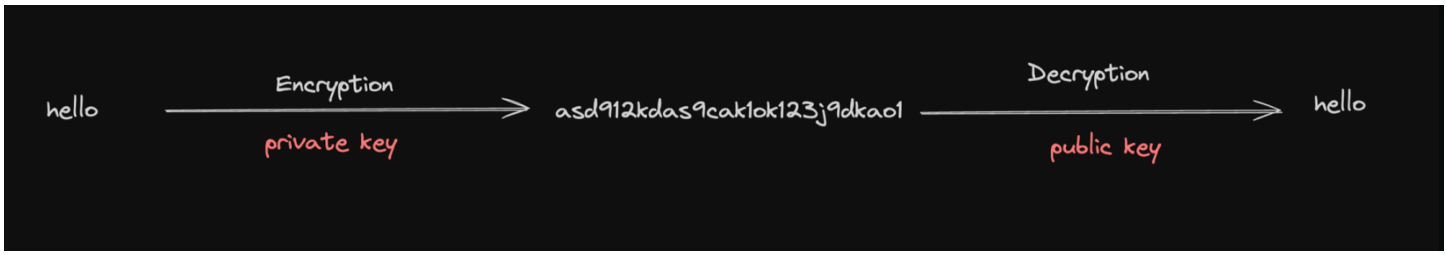
▼ **Example**:

```
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);
```

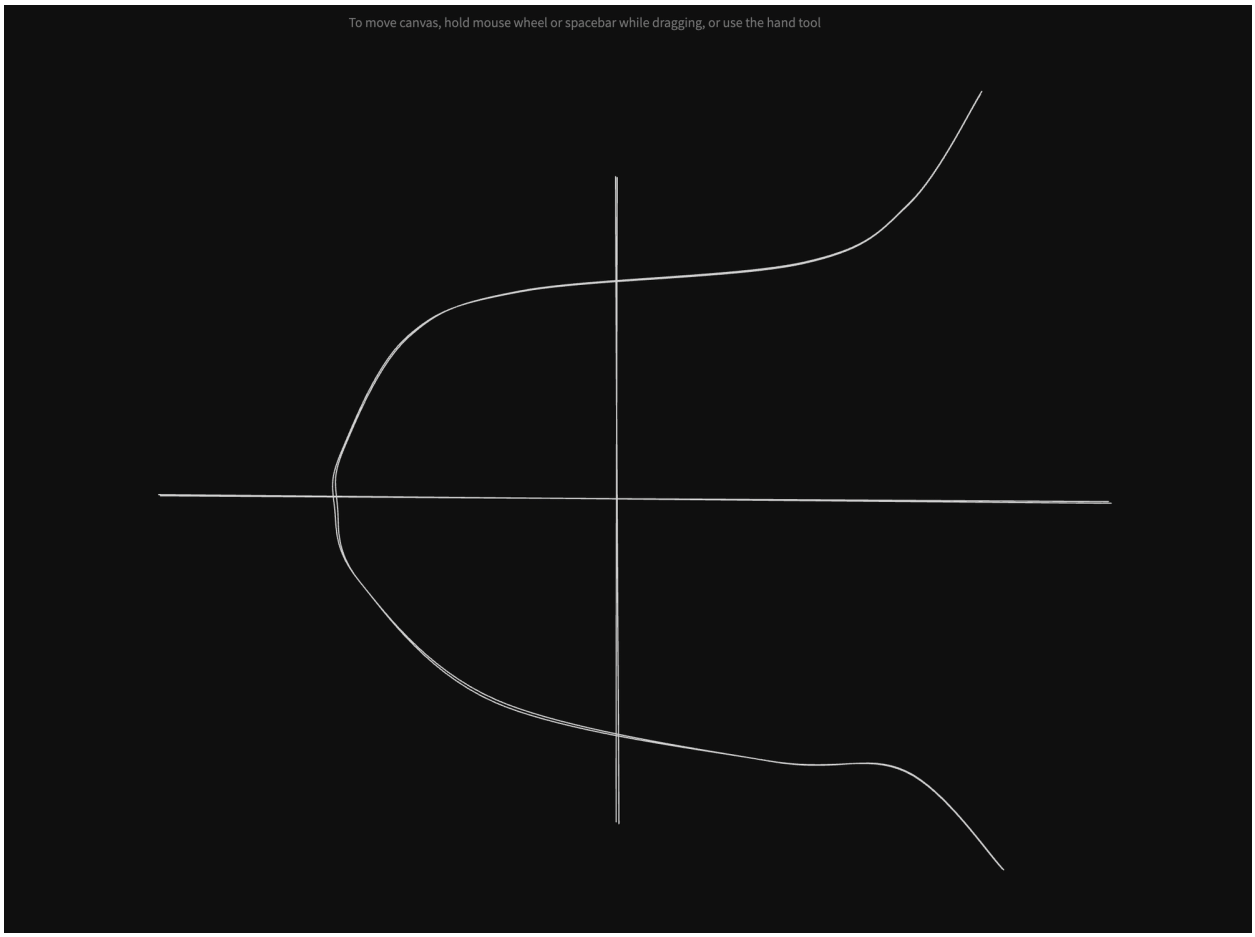
💡 AES Encryption and Decryption Online (devglan.com)

2. Asymmetric Encryption:



Uses a pair of keys – a public key and a private key – for encryption and decryption.

- **Key Pair**:
  - **Public Key**: Can be shared openly and is used to encrypt data.
  - **Private Key**: Must be kept confidential and is used to decrypt data encrypted with the corresponding public key.
- **Common Algorithms**:
  - **RSA (Rivest–Shamir–Adleman)**
  - **ECC (Elliptic Curve Cryptography) - Used by ETH and BTC**
  - **EdDSA (Edwards-curve Digital Signature Algorithm) - Used by SOL**
- **Common Elliptic Curves**:
  - **secp256k1**: Used in Bitcoin (BTC) and Ethereum (ETH).
  - **ed25519**: Used in Solana (SOL).



💡 How Elliptic Curves Work

- **Use Cases of Public-Key Cryptography**:
  - **SSL/TLS Certificates**: Ensuring secure communication over the internet.
  - **SSH Keys**: For secure server access or pushing code to GitHub.
  - **Blockchains and Cryptocurrencies**: Ensuring secure and verifiable transactions.

💡 - A message on the blockchain is signed using private key.
- A miner verifies the transaction using the signature and public key.
- Public/PrivateKeys & Signing - Blockchain Demo: Public / Private Keys & Signing (andersbrownworth.com)

Creating a public/private keypair

EdDSA - Edwards-curve Digital Signature Algorithm - ED25519

▼ Using `@noble/ed25519`

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

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 `@solanaweb3.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);
```

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 private key
  // sha256 of 'hello world'
  const msgHash =
    "b94d27b9934d3e08a52e52d7da7dabfac484efe37a5380ee9088f7ace2efcde9";
  const pubKey = secp.getPublicKey(privKey);
  const signature = await secp.signAsync(msgHash, privKey); // Sync methods below
  const isValid = secp.verify(signature, msgHash, pubKey);
```

```
  console.log(isValid);
}

main();
```

▼ Using `ethers`

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

// 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);
```

Hierarchical Deterministic (HD) Wallet

HD wallets generate a tree of key pairs from a single seed, allowing users to manage multiple addresses from one root seed.

Problem:

- Traditionally, maintaining multiple wallets required storing multiple public-private key pairs.
- This is cumbersome and risky, as losing any one of these keys can result in the loss of associated funds.

Solution - BIP-32:

- Bitcoin Improvement Proposal 32 (BIP-32), introduced by Bitcoin Core developer Pieter Wuille in 2012, addresses this problem by standardizing the derivation of private and public keys from a single master seed.
- BIP-32 introduced the concept of hierarchical deterministic (HD) wallets, which use a tree-like structure to manage multiple accounts easily.

How to Create an HD Wallet

Mnemonics

- A mnemonic phrase, or seed phrase, is a human-readable sequence of words used to generate a cryptographic seed.
- BIP-39(Improvement to BIP-32) defines how mnemonic phrases are generated and converted into a seed.

Example Code to Generate a Mnemonic:

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

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

💡 **Reference:**

- BIP-39
- Example in where it is done in Backpack: GitHub Link
- YouTube Reference

Seed Phrase

- The seed is a binary number derived from the mnemonic phrase. This seed is used to generate the master private key.

Example Code to Generate a Seed from a Mnemonic:

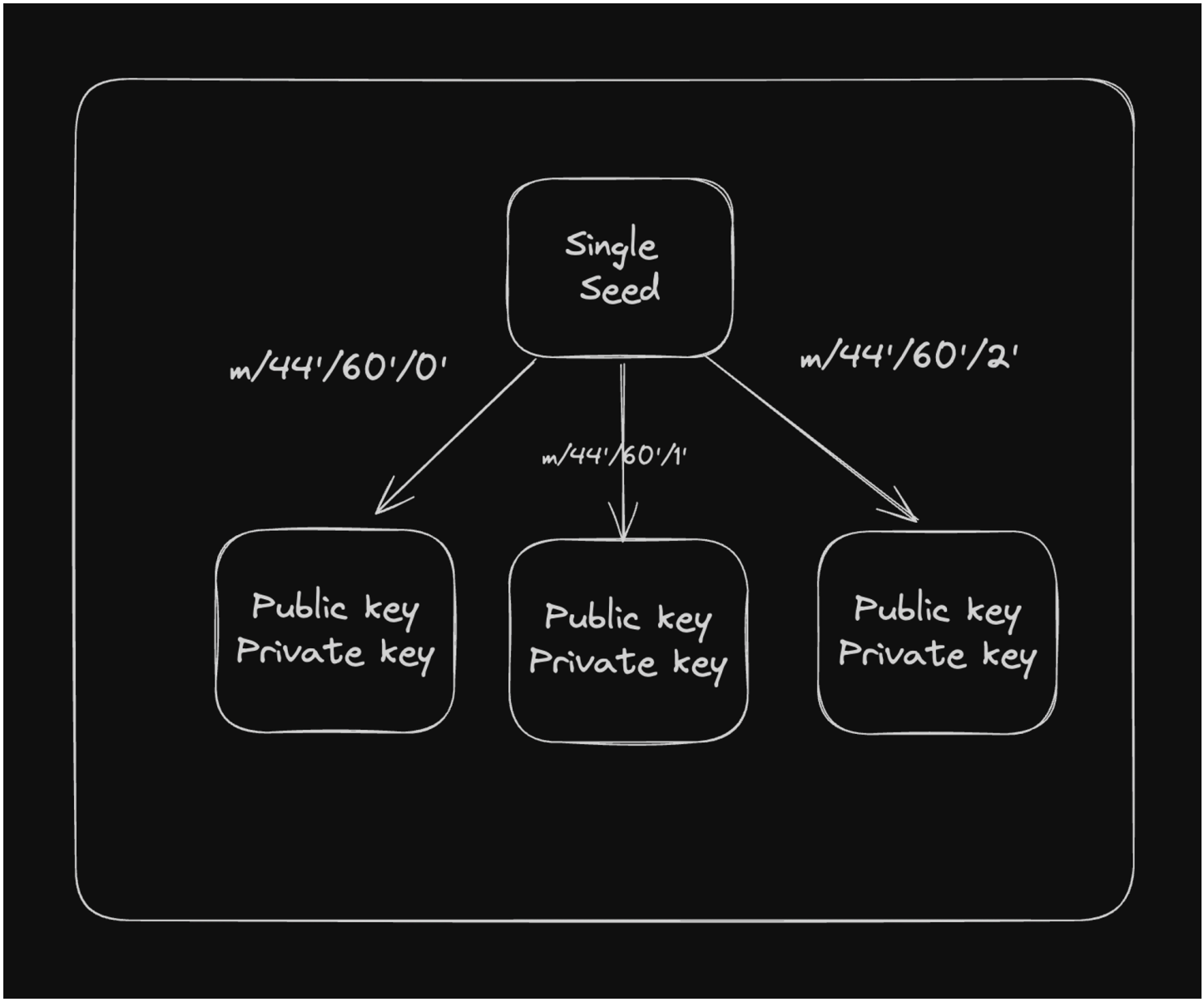
```
import { generateMnemonic, mnemonicToSeedSync } from "bip39";

const mnemonic = generateMnemonic();
```

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

Reference:

- Example in Backpack: GitHub Link

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.

- A derivation path is typically expressed in a format like `m / purpose' / coin_type' / account' / change / address_index` .

- `m` : Refers to the master node, or the root of the HD wallet.
- `purpose` : A constant that defines the purpose of the wallet (e.g., `44'` for BIP44, which is a standard for HD wallets).
- `coin_type` : Indicates the type of cryptocurrency (e.g., `0'` for Bitcoin, `60'` for Ethereum, `501'` for solana).
- `account` : Specifies the account number (e.g., `0'` for the first account).
- `change` : This is either `0` or `1` , where `0` typically represents external addresses (receiving addresses), and `1` represents internal addresses (change addresses).
- `address_index` : A sequential index to generate multiple addresses under the same account and change path.

Example Code for Deriving Paths and Generating Keys:

```
import nacl from "tweetnacl";
import { generateMnemonic, mnemonicToSeedSync } from "bip39";
import { derivePath } from "ed25519-hd-key";
import { Keypair } from "@solana/web3.js";

const mnemonic = generateMnemonic();
const seed = mnemonicToSeedSync(mnemonic);
for (let i = 0; i < 4; i++) {
  const path = `m/44'/501'/${i}'/0'`; // Derivation path for Solana
  const derivedSeed = derivePath(path, seed.toString("hex")).key;
  const secret = nacl.sign.keyPair.fromSeed(derivedSeed).secretKey;
  console.log(Keypair.fromSecretKey(secret).publicKey.toBase58());
}
```

Reference:

- Solana-specific Implementation:
  - GitHub Link 1
  - GitHub Link 2

Additional

💡 Can you guess the 12-word recovery phrase [Explanation with Calculations] 💡

Understanding the 12-Word Recovery Phrase

1. **Mnemonic Phrases (BIP39 Standard):**
    - The 12-word recovery phrase is based on the BIP39 standard, which is commonly used to generate and restore wallets.
    - These phrases are used to generate the wallet's private key. The words are chosen from a specific list of 2,048 words (known as the BIP39 wordlist).
2. **Combinatorial Explosion:**
    - A 12-word recovery phrase can be any combination of 12 words from this list.
    - The number of possible combinations of 12 words from a list of 2,048 words is astronomical.

Computation of Combinations

To compute the total number of possible 12-word combinations:

$$\text{Total Combinations} = 2048^{12}$$

Let's calculate that:

$$2048^{12} \approx 2^{132} \approx 5.444517870735016 \times 10^{39}$$

This is approximately 5.4×10^{39} possible combinations.

Probability of Guessing Correctly

The probability of correctly guessing a 12-word recovery phrase is the inverse of the number of combinations:

$$Probability = \frac{1}{2048^{12}} \approx 1.8 \times 10^{-40}$$

This probability is incredibly small, making it nearly impossible to guess the correct recovery phrase by chance.

Computational Effort and Time

Let’s assume you could check a huge number of phrases per second:

- **Hypothetical Scenario:**
    - Suppose you could check 1 billion (10^9) Phrases per second. This is an unrealistically high number but will help illustrate the difficulty.
    - Number of seconds in a year: $31,536,000 seconds/year$
    - Number of checks per year: $10^9 \times 31,536,000 \approx 3.1536 \times 10^{16}$

Even at this rate, it would take:

$$\frac{5.4\times10^{39}}{3.1536\times10^{16}} \approx 1.71 \times 10^{23} \text{ years}$$

This is longer than the current age of the universe by many orders of magnitude.

Practical Considerations

- **Random Generation Is Impractical:** Generating a random 12-word phrase and finding a matching wallet by brute force is practically impossible due to the enormous number of possible combinations.
- **Cryptographic Security:** Modern cryptocurrencies are designed with security in mind, making brute force attacks infeasible.

Conclusion

It is theoretically possible to find a 12-word recovery phrase by luck or by generating random phrases, but the probability of success is so low that it is effectively impossible.

Even with the most powerful computational resources, the time required would exceed the age of the universe by an unimaginable factor.

Cryptocurrencies rely on this extremely low probability to ensure the security of wallet keys, making it virtually impossible to guess or brute-force someone's private key or recovery phrase.