Chapter 3 Notes:

2. Proof of Burn

- **What It Is**: Proof of Burn is a way to show a commitment to a project by "burning" (permanently removing) some bitcoins or other cryptocurrency.
- **How It Works**:
    - To burn coins, you send them to a special address where they can never be spent or retrieved. This means those coins are essentially taken out of circulation forever.
    - By burning coins, people can prove they have invested value in the system, sometimes earning privileges or other coins in return.
- **Purpose**: Proof of Burn is used in some blockchain projects as a way to participate without requiring costly hardware, as in mining. It shows dedication by sacrificing resources.

Greeb Address: A green address is a trusted Bitcoin address that allows for faster transactions.

How It Works:

- Transactions from a green address are usually backed by a trusted third party, which means the recipient can trust that the funds are valid without waiting for confirmation from the network.

Micropayment escrows: are used to handle small payments in a secure way.

How It Works:

- In an escrow, funds are held by a third party until specific conditions are met (e.g., delivery of goods or services).
- Micropayment escrows are set up specifically for small transactions, making it cost-effective and secure for both the buyer and the seller.

5. Limitations and Improvements

- **What It Is**: This is about understanding the current limitations of Bitcoin and exploring ways it can be improved.
- **Common Limitations**:

    - **Scalability**: Bitcoin can process only a limited number of transactions per second, which can lead to delays and high fees during busy periods.
    - **Energy Consumption**: Mining Bitcoin consumes a large amount of electricity, which is a concern for environmental sustainability.
    - **Privacy**: Although Bitcoin transactions are pseudonymous, they can still be traced, which may not be ideal for everyone's privacy needs.
- **Possible Improvements**:
    - **Layer 2 Solutions**: Technologies like the Lightning Network can help Bitcoin scale by handling smaller transactions outside the main blockchain, reducing congestion.
    - **Eco-Friendly Mining**: Exploring alternative, energy-efficient methods for mining, such as Proof of Stake.
    - **Better Privacy Features**: Adding enhancements like confidential transactions to improve privacy.

1. Creating and Configuring a New Blockchain

- **multichain-util create [chain-name]**: Initializes a new blockchain configuration with the specified chain name. This creates a configuration file where you can set parameters like mining difficulty, block size, permissions, and more.

2. Starting and Connecting to a Blockchain

- **multichaind [chain-name]**: Starts the blockchain node for a specified chain. This command must be run on the node where the chain was created.
- **multichaind [chain-name]@[ip-address]:[port]**: Connects a new node to an existing blockchain network by specifying the chain name, IP address, and port of an active node.

3. Checking Blockchain Status and Information

- **getinfo**: Provides general information about the blockchain and node status, including protocol version, blocks, connections, difficulty, and memory pool.
- **getblockchainparams**: Shows all configurable parameters of the blockchain, such as mining parameters, transaction fees, and consensus settings.

4. Permission Management

- **grant [address] [permissions]**: Grants specific permissions to an address (e.g., connect, send, receive, mine, admin). Example: grant 1A2b3c4D5E6f connect,send,receive.
- **revoke [address] [permissions]**: Revokes permissions from an address. Example: revoke 1A2b3c4D5E6f connect.
- **listpermissions [permissions]**: Lists addresses and permissions on the blockchain. You can specify a specific permission type or leave it blank to see all.

5. Managing Assets

- **issue [address] [asset-name] [quantity] [unit]**: Issues a new asset on the blockchain to a specified address, defining total supply and smallest divisible unit.
- **listassets**: Lists all the assets created on the blockchain, showing details like asset name, quantity, and divisibility.
- **sendasset [to-address] [asset-name] [quantity]**: Sends a specified quantity of an asset to a target address.

6. Stream Management

- **create stream [stream-name] true**: Creates a new stream on the blockchain, which can store data entries.
- **publish [stream-name] [key] [data-hex]**: Publishes data to a stream with a specified key and hex-encoded data.
- **subscribe [stream-name]**: Subscribes to a stream to start receiving updates when new data entries are added.
- **liststreamitems [stream-name]**: Lists all items in a specified stream, showing keys and associated data.

7. Transaction Management

- **getrawtransaction [txid] 1**: Retrieves details of a specific transaction using the transaction ID. Adding 1 provides a full JSON output.
- **send [address] [amount]**: Sends a specified amount of the blockchain's native token to an address.

8. Address Management

- **getaddresses**: Lists all addresses created or controlled by the current node.
- **getnewaddress**: Generates a new address that can be used to receive assets or participate in blockchain transactions.

9. Blockchain and Network Health

- **listnodes**: Lists connected nodes and their current status.
- **listblocks [block-number]**: Retrieves detailed information for a specific block number, useful for block verification and monitoring.
- **getmempoolinfo**: Shows the current status of the memory pool, including pending transaction count and memory usage.

10. Debugging and Logging

- **logging [category] [level]**: Adjusts the logging level for a specific category (e.g., net, rpc, bench).
- **getblock [blockhash]**: Retrieves a block's full details using the block hash, including transactions and timestamp information.

Chapter 3 Questions and Answers:

1. Transaction Validation: Computational Cost & Data Structures

- **Question**: Consider the steps involved in processing Bitcoin transactions. Which of these steps are computationally expensive? If you're an entity validating many transactions (say, a miner) what data structure might you build to help speed up verification?
    - **Answer**:
        - **Computationally Expensive Steps**:
            - **Verifying Signatures**: Each transaction has a digital signature that proves the sender owns the bitcoins being spent. Checking these signatures requires a lot of computing power.

        - **Checking Double-Spending**: The network must verify that bitcoins being spent haven't already been spent elsewhere. This requires scanning past transactions, which can be slow.

    - **Data Structure for Faster Verification**:

        - Miners use a **Merkle Tree** to speed up verification. This data structure helps them quickly check if a transaction is in a block by organizing transactions into a tree format. Miners only need a small part of this tree (called a Merkle proof) to verify a transaction, making it much faster.

---

2. Bitcoin Script

- **Question**: For the following questions, you're free to use non-standard transactions and op codes that are currently disabled. You can use <data> as a shorthand to represent data values pushed onto the stack.

    - **Answer**:

        - **a. Write the Bitcoin ScriptPubKey Script for Anyone Supplying the Square Root of 1764**:

            - OP_DUP OP_MUL 1764 OP_EQUAL

        - **b. Write the ScriptSig to Redeem the Transaction**:

            - 42

        - **c. RSA Factoring Challenge**:

            - **Problem**: Factoring a 1024-bit RSA number is extremely difficult and could take a very long time (even years) to solve with current technology.

            - **Difficulty**: Since only someone who can break the RSA encryption could claim the bitcoins, this method would not be practical because it may remain unclaimed indefinitely.

---

3. Bitcoin Script II

- **Question**: Alice is backpacking and is worried about her devices containing private keys getting stolen. So she would like to store her bitcoins in such a way that they can be redeemed via knowledge of only a password. Accordingly, she stores them in the following ScriptPubKey address:
    - **Answer**:
        - OP_SHA1 <0x084a3501edef6845f2f1e4198ec3a2b81cf5c6bc> OP_EQUALVERIFY
            - **a. Write the ScriptSig to Redeem**:
                - <password>
            - **b. Why This Isn't Secure**:
                - SHA-1 is a weak hashing algorithm and can be broken with enough computational power. Attackers could try various inputs (passwords) to see which one produces the required hash, a process called brute-forcing.
            - **c. Would Using Pay-to-Script-Hash (P2SH) Help?**
                - P2SH would not solve the fundamental issue of weak password security because it still relies on SHA-1. An attacker could brute-force it in the same way; P2SH only changes the way scripts are called and verified.

---

6. Green Addresses

- **Question**: One problem with green addresses is that there is no punishment against double-spending within the Bitcoin system itself. To solve this, you decide to design an altcoin called "GreenCoin" that has built-in support for green addresses. Any attempt at double spending from addresses (or transaction outputs) that have been designated as "green" must incur a financial penalty in a way that can be enforced by miners. Propose a possible design for GreenCoin.
    - **Answer**:
        - **GreenCoin Design**:

        - GreenCoin could include a rule where miners charge a financial penalty if someone tries to double-spend from a green address.
        - **How It Works**: If the same GreenCoin output is spent twice, a penalty could be taken from the sender's wallet, or their future transactions could be temporarily blocked.
        - **Enforcement by Miners**: Miners could include special code in GreenCoin to detect and penalize double-spending attempts automatically.

---

7. SPV Proofs

- **Question**: Suppose Bob the merchant runs a lightweight client and receives the current head of the blockchain from a trusted source.
    - **Answer**:
        - **a. What Bob Needs for Proof of Payment**:
            - To prove that a payment to Bob has been included in the blockchain, customers need to show:
                1. The transaction ID of their payment.
                2. A Merkle proof that shows this transaction is in a specific block.
                3. A block header (or hash) for the block containing the transaction.
            - Bob will need 6 confirmations, meaning he should see 6 more blocks added after the block containing the transaction.
        - **b. Estimating Proof Size**:
            - **Size of Proof**:
                - If each block has 1024 transactions, then the Merkle proof for one transaction would need about 10 hashes (since 210=10242^{10} = 1024210=1024).

        - Each hash is about 32 bytes, so the Merkle proof size is roughly 10×32=32010 \times 32 = 32010×32=320 bytes.
        - Including a block header (80 bytes), the total is about 400 bytes.

---

8. Adding New Features

- **Question**: Assess whether the following new features could be added using a hard fork or a soft fork.
    - **Answer**:
        - **a. Adding a new OP_SHA3 script instruction**:
            - Hard Fork
        - **b. Disabling the OP_SHA1 instruction**:
            - Soft Fork
        - **c. A requirement that each miner include a Merkle root of unspent transaction outputs (UTXOs) in each block**:
            - Hard Fork
        - **d. A requirement that all transactions have their outputs sorted by value in ascending order**:
            - Soft Fork
    - **Explanation**:
        - **Hard Fork**: Major, incompatible update that requires all nodes to upgrade, creating two separate chains if not everyone upgrades.
        - **Soft Fork**: Minor, compatible update that works with old nodes and typically adds restrictions, keeping the blockchain unified if most miners adopt the change.

Chapter 4: Wallet Creation and Decentralization

1. Creating and Decentralizing Wallets

Creating a wallet is the first step in handling cryptocurrencies like Bitcoin. Here's how it works:

How to Create a Wallet:

- **Choose Your Wallet Type**:
    - **Hot Wallet**: This is online and easy to access, like apps on your phone or computer. Great for daily use!
    - **Cold Wallet**: This is offline and more secure, like a USB drive or a piece of paper. Better for long-term storage.
- **Generate Your Wallet**:
    - When you create a wallet, it gives you two keys:
        - **Private Key**: This is like your secret password. Keep it safe because it lets you access your money.
        - **Public Key**: This is like your bank account number. You can share it with others so they can send you money.
- **Backup Your Wallet**:
    - When you set up your wallet, it usually gives you a list of words (called a seed phrase). Write this down and keep it safe! This phrase helps you recover your wallet if you lose access.

Decentralizing Control:

- **Non-Custodial Wallets**: These wallets let you keep your own keys, so you are in control of your money. No one else can access it.
- **Multi-Signature Wallets**: These require more than one key to move funds. For example, if three friends have keys, you might need two of them to agree before spending money. This keeps funds safer.
- **Shared Wallets**: Here, different people have parts of the key. No single person can access the funds alone, which reduces risk.

---

2. Splitting and Sharing Keys

Keeping your private keys safe is crucial. Here are ways to split and share keys:

Key Splitting Techniques:

1. **Shamir's Secret Sharing**:
    - This method splits your private key into several pieces. For example, you might create five pieces but need only three to unlock the key. This way, even if someone loses some pieces, the key can still be reconstructed.
2. **Multi-Signature Wallets**:
    - As mentioned before, these require several people to agree before a transaction can happen. This means you can't lose your money just because one person has their key compromised.
3. **Distributing Trust**:
    - When sharing keys among friends or partners, make sure that no one person has complete control. This way, you can keep your money safer.

---

3. Proof of Liability

Proof of l

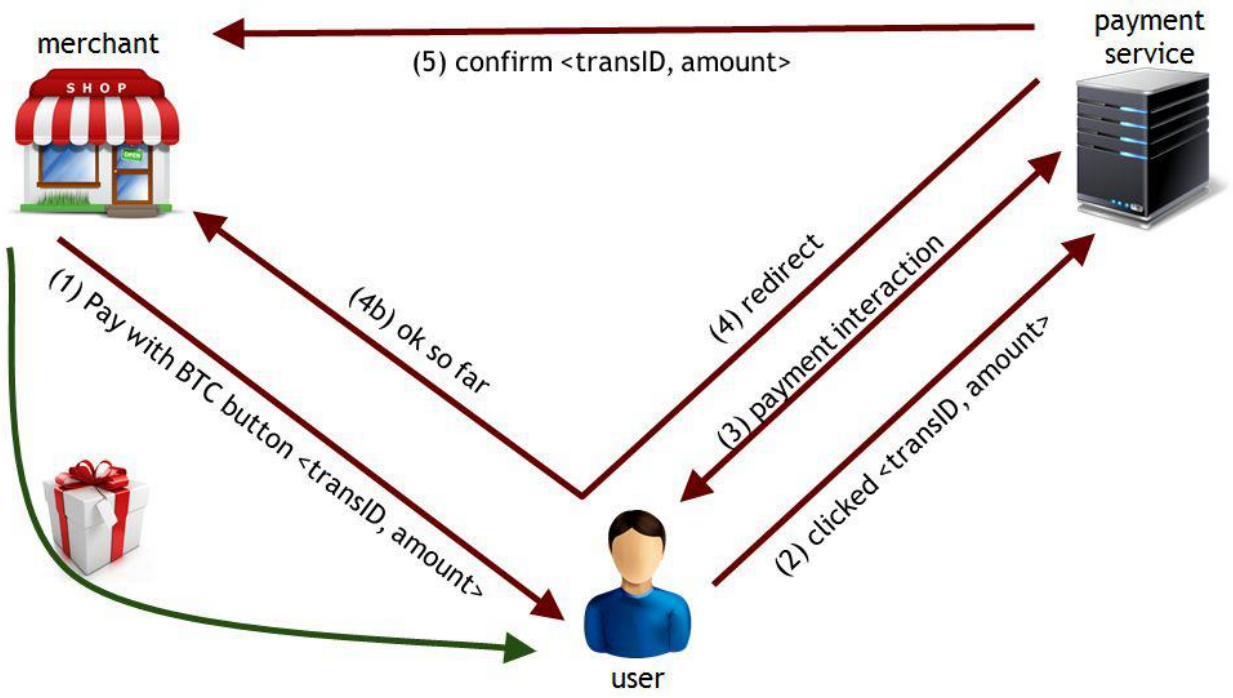
iability is about showing you have enough funds and that they are managed correctly.

Key Points:

1. **Transparency**:
    - In regular banks, they manage your money, and you trust them. In decentralized systems, everyone should be able to check that their money is safe without needing to trust a bank.
2.
3. **Smart Contracts**:

- These are like automated agreements that execute themselves when certain conditions are met. For instance, if two people agree to trade, the smart contract will hold the money until both fulfill their parts.

4. **Regular Audits**:
    - Checking the smart contracts and transactions regularly ensures everything is working as it should. This helps keep everything transparent and accountable.



Payment process involving a user, merchant, and payment service.

2. Proof of Liabilities

Scenario: TransparentExchange is a cryptocurrency exchange that wants to prove to its customers that it has enough funds to cover all deposits. They use a Merkle Tree to keep track of each customer's balance and total deposits.

Question 2.1: Why can't the exchange include fake customers with negative values to lower the total?

Answer: The exchange cannot include fake customers with negative values because it would make their total deposits look artificially lower. If they did this, real customers could check their

own balances and realize the exchange was lying about having enough funds. If customers notice discrepancies, it will expose the exchange's dishonesty and harm its reputation.

Question 2.2: Show an attack on this scheme that would allow the exchange to claim a total less than the actual sum of deposits.

Answer: An attack could occur if the exchange:

1. **Excludes Real Customers**: They might leave out real customers from their Merkle Tree records.
2. **Creates Fake Entries**: They could create fake customer entries with negative balances to further reduce the claimed total.

By doing this, the exchange could falsely claim they have lower total deposits than they do, misleading customers about their financial health.

Question 2.3: Fix this scheme so that it is not vulnerable to the attack you identified.

Answer: To secure the scheme:

1. **Customer Verification**: Allow customers to verify that their balances are included in the Merkle Tree.
2. **Independent Audits**: Employ a third-party auditor to periodically check the accuracy of the exchange's records.
3. **Secure Data Storage**: Ensure customer data is stored securely and cannot be easily manipulated.

These measures would help prevent the exchange from falsely claiming lower totals.

Question 2.4: Ideally, the proof that the exchange provides to a customer shouldn’t leak information about other customers. Does this scheme have this property? If not, how can you fix it?

Answer: No, this scheme can leak information because when customers receive proof of their balance, they might inadvertently learn about other customers' balances.

To fix it:

1. **Use Zero-Knowledge Proofs**: Implement cryptographic techniques that allow customers to prove their balance without revealing others' data.
2. **Separate Merkle Trees**: Create different Merkle Trees for groups of customers to maintain privacy.

These solutions would enhance the privacy of customer information.

---

4. Multi-signature Wallet

Scenario: BitCorp has noticed a security breach where an unauthorized person (Mallory) has accessed one of their servers containing Bitcoin private keys. BitCorp uses a 2-of-3 multi-signature wallet, which means they need two out of three keys to make transactions.

Question 4.1: How do they re-secure their wallet and effectively revoke the information that Mallory has learned?

Answer: To re-secure their wallet:

1. **Create a New Wallet**: BitCorp should create a new multi-signature wallet with a fresh set of keys.
2. **Transfer Funds**: They need to move all their Bitcoin from the old wallet to the new one to ensure that the compromised key is no longer valid.
3. **Inform Customers**: Notify their customers of the new wallet address so that they can update their records.

By doing this, BitCorp can protect their funds from unauthorized access.

Question 4.2: If BitCorp uses a 2-out-of-2 multi-signature wallet instead of a 2-out-of-3 wallet, what steps can they take in advance so that they can recover even if one of their servers gets broken into?

Answer: To prepare for such a situation:

1. **Backup Keys**: BitCorp should keep backups of both keys in different secure locations to prevent loss if one server is compromised.

2. **Emergency Recovery Plan**: Have a strategy in place to quickly generate new key pairs if they need to recover from a breach.
3. **Regular Key Changes**: Change the keys periodically to limit the time any stolen key could be used.

These steps would help ensure the wallet can be secured even if one key is compromised.

---

7. BitcoinLotto

Scenario: The nation of Bitcoinia wants to modernize its lottery system by using Bitcoin. They print scratch-off lottery tickets that link to a Bitcoin address containing the jackpot.

Question 7.1: What might happen if the winner finds the ticket on Monday and immediately claims the jackpot? Can you modify your design to ensure this won't be an issue?

Answer: If a winner claims their jackpot immediately after finding a ticket, it could cause issues, such as:

- **Timing Conflicts**: Other people may have bought tickets after the winner, thinking they had a chance to win as well.

Solution: To avoid this, introduce a **waiting period** where the winner must wait until the end of the week before claiming the prize. This allows time for all tickets sold during that week to be processed.

Question 7.2: Some tickets inevitably get lost or destroyed. How can we roll forward any unclaimed jackpot from Week n to the winner in Week n+1 without letting the lottery administrators embezzle funds? Also, ensure the Week n winner can't just wait until Week n+1 to double their winnings.

Answer: To manage unclaimed jackpots:

1. **Track Unclaimed Prizes**: Maintain a public record of unclaimed jackpots that everyone can see.
2. **Roll Forward Mechanism**: At the end of each week, any unclaimed jackpot from Week n should automatically add to the jackpot for Week n+1.

3. **Prevent Double Claims**: Set rules to ensure that if a winner tries to claim their prize, they can't also claim the rolled-forward jackpot. For instance, a ticket must be validated for the current week before claiming any funds.

These steps help keep the lottery fair and transparent, preventing fraud while allowing unclaimed prizes to benefit future draws.

Certainly! Let's break down **Bitcoin mining** into three main categories: **CPU Mining**, **GPU Mining**, and **ASIC Mining**. I'll explain each one in detail, including how they work, their advantages and disadvantages, and provide simple examples.

Chapter 5: Bitcoin Miners

Bitcoin mining is the process of validating transactions on the Bitcoin network and adding them to the blockchain. Miners compete to solve complex mathematical problems, and the first one to solve the problem gets to add a new block to the blockchain and is rewarded with newly minted bitcoins.

1. CPU Mining

What is CPU Mining?

- **CPU** stands for **Central Processing Unit**, which is the main part of a computer that performs most of the processing.
- In the early days of Bitcoin (around 2009), miners used regular computers with CPUs to mine Bitcoin.

How it Works:

- Miners use software to connect to the Bitcoin network and try to solve cryptographic puzzles.
- Each time a miner successfully solves a puzzle, they validate transactions and add a new block to the blockchain.

Advantages:

- Easy to set up since most computers already have a CPU.
- Doesn't require special hardware.

Disadvantages:

- Very slow and inefficient compared to other methods.
- High electricity costs relative to the small amount of Bitcoin mined.

Example:

- Imagine you have an old laptop with a CPU. You install Bitcoin mining software, and while it runs, it tries to solve the puzzles for Bitcoin mining. However, because CPUs are slow, you might earn very little Bitcoin over a long period, especially since the competition is fierce.

---

2. GPU Mining

What is GPU Mining?

- **GPU** stands for **Graphics Processing Unit**, which is typically used in gaming and graphic design.
- Miners began using GPUs for Bitcoin mining around 2010 because they are much faster than CPUs.

How it Works:

- Similar to CPU mining, miners use software to connect to the network and solve puzzles.
- GPUs can handle many calculations simultaneously, making them more efficient for mining tasks.

Advantages:

- Significantly faster than CPU mining.
- More energy-efficient for the amount of Bitcoin mined.

Disadvantages:

- Still not as efficient as specialized hardware (ASICs).
- Initial investment in GPU hardware can be high.

Example:

- Suppose you have a gaming computer with a powerful GPU. You download mining software, and the GPU begins solving puzzles much faster than a CPU could. As a result, you can mine more Bitcoin in less time, but you're still competing with others who have similar or better setups.

---

3. ASIC Mining

What is ASIC Mining?

- **ASIC** stands for **Application-Specific Integrated Circuit**. These are specialized devices built specifically for Bitcoin mining.
- ASIC miners are designed to perform only one task: mining Bitcoin, and they do it very efficiently.

How it Works:

- ASIC miners connect to the Bitcoin network and work on solving puzzles.
- Because they are optimized for this specific task, they can solve puzzles much faster than both CPUs and GPUs.

Advantages:

- Extremely efficient and powerful for mining.
- Lower power consumption per unit of Bitcoin mined compared to CPUs and GPUs.

Disadvantages:

- Higher upfront cost for purchasing ASIC hardware.
- Limited use; they can't be repurposed for other tasks.

Example:

- Imagine you invest in an Antminer S19, one of the most popular ASIC miners. This machine is designed only for Bitcoin mining and can generate a significant amount of Bitcoin each day. While the initial cost is high, the efficiency and speed make it a profitable choice in the long run.

Summary

- **CPU Mining**: Uses standard computer CPUs; slow and inefficient for Bitcoin mining today.
- **GPU Mining**: Utilizes powerful graphics cards for faster mining; better than CPUs but less efficient than ASICs.
- **ASIC Mining**: Specialized hardware designed specifically for mining Bitcoin; the most efficient option available.

Chapter 6:

Mixing

```
pragma solidity ^0.8.0;


contract MiniMixer {
    mapping(address => uint256) public deposits; // Track deposits by user
    address[] public participants; // Store participants' addresses
    bool public mixingStarted; // Indicates if mixing has started

    // Event declarations
    event Deposit(address indexed sender, uint256 amount);
    event Withdraw(address indexed recipient, uint256 amount);
    event MixingStarted();

    // Function to deposit Ether into the mixer
    function deposit() external payable {
```

```
    require(msg.value > 0, "Must send some Ether");

    // Add user to participants list if not already present
    if (deposits[msg.sender] == 0) {
        participants.push(msg.sender);
    }

    // Increase the deposit balance for the sender
    deposits[msg.sender] += msg.value;
    emit Deposit(msg.sender, msg.value);
}

// Function to start mixing
function startMixing() external {
    require(participants.length > 1, "Need at least 2 participants");
    mixingStarted = true;
    emit MixingStarted();
}

// Function to withdraw mixed Ether
function withdraw(uint256 amount) external {
    require(mixingStarted, "Mixing not started yet");
    require(deposits[msg.sender] >= amount, "Insufficient balance");
```

```
        // Deduct the amount from the participant's balance
        deposits[msg.sender] -= amount;

        // Transfer the amount to the user
        (bool success, ) = msg.sender.call{value: amount}("");
        require(success, "Withdrawal failed");

        emit Withdraw(msg.sender, amount);
    }

    // Fallback function to receive Ether directly
    receive() external payable {}
}
```

BlindSignature

```
pragma solidity ^0.8.0;

contract BlindSignature {
    address public signer; // The address of the signer who will approve blind messages
    mapping(bytes32 => bool) public isSignatureApproved; // Mapping to track approved blind messages

    // Event declarations
    event SignatureRequested(address indexed requester, bytes32 blindedMessage);
```

```
event SignatureApproved(bytes32 blindedMessage);

// Constructor to initialize the signer address
constructor(address _signer) {
    signer = _signer;
}

// Function for users to request a blind signature
function requestSignature(bytes32 blindedMessage) external {
    // Emit an event to log the signature request
    emit SignatureRequested(msg.sender, blindedMessage);
}

// Function for the signer to approve the blind signature
function approveSignature(bytes32 blindedMessage) external {
    require(msg.sender == signer, "Only the designated signer can approve");
    isSignatureApproved[blindedMessage] = true;

    // Emit an event to indicate approval
    emit SignatureApproved(blindedMessage);
}

// Function to check if a blinded message has been approved
function checkApproval(bytes32 blindedMessage) external view returns (bool) {
```

```
        return isSignatureApproved[blindedMessage];
    }

    // Function to hash a message using Keccak256
    function hashMessage(string calldata message) external pure returns (bytes32) {
        return keccak256(abi.encodePacked(message));
    }
}
```

Minting:

```
pragma solidity ^0.8.0;

contract SimpleZerocoin {
    mapping(address => uint256) public balances; // User balances of Zerocoins
    mapping(bytes32 => bool) public nullifiers; // Track spent coins to prevent double-spending

    // Events for Zerocoin operations
    event ZerocoinMinted(address indexed user, bytes32 coin);
    event ZerocoinSpent(address indexed user, bytes32 coin);

    // Function to mint Zerocoins
    function mint(bytes32 coin) external {
```

```
        require(!nullifiers[coin], "Coin already spent");
        // Increase the user's Zerocoin balance
        balances[msg.sender] += 1;
        emit ZerocoinMinted(msg.sender, coin);
    }

    // Function to spend Zerocoins
    function spend(bytes32 coin) external {
        require(balances[msg.sender] > 0, "No Zerocoins to spend");
        require(!nullifiers[coin], "Coin already spent");
        // Decrease user's Zerocoin balance and mark the coin as spent
        balances[msg.sender] -= 1;
        nullifiers[coin] = true;
        emit ZerocoinSpent(msg.sender, coin);
    }

    // Function to check if a specific coin has been spent
    function isSpent(bytes32 coin) external view returns (bool) {
        return nullifiers[coin];
    }
    function hashMessage(string calldata message) external pure returns (bytes32) {
        return keccak256(abi.encodePacked(message));
    }
}
```