System Design and Implementation

This system is developed using Solidity v0.8 to write smart contracts and employs the Truffle framework for compilation, deployment, and testing. It fulfills various requirements of the DutchMarket Dutch auction smart contract, including but not limited to adding accounts, storing ETH and tokens, adding sell orders, modifying sell orders, removing sell orders, adding buy orders, real buyers revealing buy orders, removing buy orders, order matching, extracting ETH and tokens, switching modes, and other functions.

System Architecture

We use Truffle for contract development, mainly consisting of two smart contracts: DutchMarket.sol and Token.sol. DutchMarket is the main contract, containing the entire system's functionality and design. Token, on the other hand, is an ERC20 token contract that inherits from "@openzeppelin/contracts/token/ERC20/ERC20.sol". By default, 100 million tokens are minted for testing purposes.

Installation Steps

```
# Install nodejs, download link: https://nodejs.org/en/download/
# Install truffle yarn
npm install -g truffle yarn
# Install dependencies
yarn
# Run tests
truffle test
```

Smart Contract Analysis

The contract includes three structs: Account, Order, and Bid, which are used to implement user management, sell order management, and buy order management, respectively. The details of each structure in this contract are as follows:

Account

```
// Account structure
struct Account {
    // Custodial ETH balance
    uint256 balance;
    // Custodial token balance
    mapping(address => uint256) tokenBalances;
}
```

- All users are managed through mapping(address => Account) private accounts.
- When users make deposit or withdrawal operations, the corresponding records are updated in this struct.
- `uint256 balance` stores the user's ETH balance.
- `mapping(address => uint256) tokenBalances` stores the balances of multiple ERC20 tokens of different types for this user.
- ETH Deposit/Withdrawal

- Deposit: Users call the depositETH() method to transfer ETH to the contract, and the contract will transfer the ETH into the user's account.
- Withdrawal: Users call the withdrawETH() method to withdraw ETH from the contract, and the contract will transfer the ETH to the user's account.

```
// ETH deposit
function depositETH() public payable {
    // Validate mode
    require(currentMode == Mode.DepositWithdraw, "Not in DepositWithdraw mode");
    // Check deposit amount must be greater than 0
    require(msg.value > 0, "Amount must be greater than 0");
    // Increase custodial ETH balance
    accounts[msg.sender].balance += msg.value;
    // Emit ETH deposit event
    emit Deposit(address(this), msg.sender, msg.value);
}

// ETH withdrawal
function withdrawETH(uint256 amount) public {
    // Validate mode
    require(currentMode == Mode.DepositWithdraw, "Not in DepositWithdraw mode");
    // Check withdrawal amount must be greater than 0
    require(amount > 0, "Amount must be greater than 0");
    // Check custodial ETH balance must be >= withdrawal amount
    require(accounts[msg.sender].balance >= amount, "Insufficient balance");
    // Deduct from custodial ETH balance
    accounts[msg.sender].balance -= amount;
    // Add to user's wallet balance
    payable(msg.sender).transfer(amount);
    // Emit ETH withdrawal event
    emit Withdraw(address(this), msg.sender, amount);
}
```

- Token Deposit/Withdrawal
  - Deposit: Users call the depositToken() method to transfer tokens to the contract, and the contract will transfer the tokens into the user's account.
  - Withdrawal: Users call the withdrawToken() method to withdraw tokens from the contract, and the contract will transfer the tokens to the user's account.

```
// Token deposit
function depositToken(address tokenAddress, uint256 amount) public {
    // Validate mode
    require(currentMode == Mode.DepositWithdraw, "Not in DepositWithdraw mode");
    // Ensure a valid address
    require(tokenAddress != address(0) && tokenAddress != address(this), "Invalid Token Address");
    // Check deposit amount
    require(amount > 0, "Amount must be greater than 0");
    // Transfer token to contract
    require(IERC20(tokenAddress).transferFrom(msg.sender, address(this), amount), "Transfer failed");
    // Increase account token balance
    accounts[msg.sender].tokenBalances[tokenAddress] += amount;
    // Emit token deposit event
    emit Deposit(tokenAddress, msg.sender, amount);
}

// Token withdrawal
function withdrawToken(address tokenAddress, uint256 amount) public {
    // Validate mode
    require(currentMode == Mode.DepositWithdraw, "Not in DepositWithdraw mode");
    // Pass by reference
    Account storage account = accounts[msg.sender];
    // Ensure a valid address
    require(tokenAddress != address(0) && tokenAddress != address(this), "Invalid Token Address");
    // Check withdrawal amount must be greater than 0
    require(amount > 0, "Amount must be greater than 0");
    // Check account balance must be >= withdrawal amount
    require(account.tokenBalances[tokenAddress] >= amount, "Insufficient token balance");
    // Deduct from account balance
    account.tokenBalances[tokenAddress] -= amount;
    // Increase user's wallet token balance
    require(IERC20(tokenAddress).transfer(msg.sender, amount), "Transfer failed");
    // Emit withdrawal event
    emit Withdraw(tokenAddress, msg.sender, amount);
}
```

Bid

```
// Buy order structure
struct Bid {
    // Token contract, defaults to address(0) before disclosure
    address tokenAddress;
    // Buy price, defaults to 0 before disclosure
    uint256 price;
    // Quantity, defaults to 0 before disclosure
    uint256 quantity;
    // Buy order number
    uint256 bidNumber;
    // Agent address (B)
    address buyer;
    // Real buyer address (A), defaults to bytes32(0) before disclosure
    address bidder;
    // Blinded bid hash
    // Calculated as: keccak256(abi.encode(tokenAddress, price, quantity, secret))
    bytes32 blindedBid;
    // Agent B's signature
    bytes signature;
    // Whether the blinded bid has been disclosed
    bool revealed;
    // Whether the order has been matched
    bool matched;
}
```

- In the design of this system, the buy orders are initiated by agent B. Agent B can choose themselves as the real buyer or select someone else as the real buyer. However, choosing themselves as the real buyer doesn't make sense for the entire system since the purpose of the system is to hide the identity of the real buyer.
- Agent B can create a buy order by calling the addOffer() method, which only requires a blind bid hash and a signature. Both the blind bid hash and the signature are generated off-chain (**calculated using the ethers or web3 library in JavaScript**). The blind bid hash is calculated based on the **order information + real buyer A's address**, while the signature is obtained by signing the blind bid hash with agent B's private key. This part is demonstrated in `test/4_add_reveal_remove_bid.js` and 5_order_maching.js, with some of the code shown below:

```
// Get the hash
// Parameters: token address, price, quantity, and actual buyer
function getHash(tokenAddress, price, quantity, trueBuyer) {
    // Calculate the bytes32 value for the actual buyer
    const trueBuyerBytes32 = ethers.utils.padZeros(ethers.utils.arrayify(trueBuyer), 32);
    const encodedData = ethers.utils.defaultAbiCoder.encode(
        ['address', 'uint256', 'uint256', 'bytes32'],
        [tokenAddress, price, quantity, trueBuyerBytes32]
    );
    const hash = ethers.utils.keccak256(encodedData);
    return hash;
}

// Get the signature
// Parameters: hash and agent
async function getSignature(messageHash, buyer) {
    return await web3.eth.sign(messageHash, buyer);
}

// Calculate the hash that includes the order information and the actual buyer A
const messageHash = getHash(tokenInstance.address, price, quantity, accounts[9]);

// Calculate the signature for agent B
const signature = await getSignature(messageHash, accounts[0]);

// Buy order created. Anyone monitoring the blockchain cannot see my data, as I only submitted a hash and a
await dutchMarketInstance.addBid(
    messageHash,
    signature,
    { from: accounts[0] }
);
```

Next, let's look at the addBid receiving method in Solidity. It stores the hash and agent B's signature using bytes32 blindedBid and bytes signature, respectively, and constructs an *empty order* based on them. The specific implementation is as follows:

```
// Create a blind bid
// Called by agent B
// Parameters: blinded bid hash, signature
function addBid(
    bytes32 _blindedBid,
    bytes memory _signature
) public returns (uint256) {
    // Verify mode
    require(currentMode == Mode.BidOpening, "Not in BidOpening mode");
    // Verify the length of the blinded bid hash
    require(bytes32(_blindedBid).length == 32, "The blindedBid length must be 32");
    // Increment global bid number
    lastBidNumber++;
    // Increment bid count
    bidsCount++;
    // Create a blind bid
    bids[lastBidNumber] = Bid(
        // Token address
        address(0),
        // Price
        0,
        // Quantity
        0,
        // Bid number
        lastBidNumber,
        // Agent B
        msg.sender,
        // Real buyer A
        address(0),
        // Blinded bid hash
        _blindedBid,
        // Signature of agent B
        _signature,
        // Revealed or not
        false,
        // Matched or not
        false
    );
    // Return bid number
    return lastBidNumber;
}
```

This way, agent B creates a blind bid buy order. The entire process only involves submitting the blind bid hash and signature. Even if others listen to lastBidNumber and then query the buy order's detailed data through getBid(uint256 bidNumber), they cannot know what real buyer A has actually submitted, perfectly hiding A's identity as well as order price, quantity, token type, and other information.

- The real buyer A, hidden in the dark, has been keeping an eye on a certain seller's token. When the seller keeps lowering the price to A's expected price, A decides to reveal their buy order and then waits for the matching mode to arrive to be matched and traded.
  A calls the BidReveal() method to reveal their blind bid, which requires the buy order number, token address, price, quantity, and blind bid hash as inputs. Of course, if agent B chooses themselves as the real buyer, they can also directly call this method to reveal the blind bid.
  Once the blind bid is revealed, A's identity and order price, quantity, and other information will be exposed. Therefore, there is only a reason to reveal the blind bid when the seller lowers the price to A's psychological

price level. Once revealed, the blind bid loses its meaning. This mechanism ensures the security of the blind bid to some extent.

Calling the BidReveal method will verify whether the blind bid hash is consistent with the previously submitted one. If consistent, it will expose the real buyer A's address and order information, and the blind bid's revealed status will be set to true. This way, others can query A's buy order's complete information through the getBid(uint256 bidNumber) method, rather than just an empty order as before.

The entire verification process is based on cryptographic security, proving that: **It is indeed A who intended to use B's signed transaction to submit the blind auction**

The specific implementation is as follows:

The specific implementation is as follows:

```
// Reveal a bid
// Can only be called by the real buyer A, unless A and B are the same person. Otherwise, no one else can r
// The seller will call this function to reveal the bid only when the price drops to the price that the rea
// Unrevealed bids will be ignored during the order matching period
// Parameters: bid number, token address, price, quantity, blinded bid hash
function BidReveal(uint256 bidNumber, address tokenAddress, uint256 price, uint256 quantity, bytes32 _blinc
    // Verify mode
    require(currentMode == Mode.BidOpening, "Not in BidOpening mode");
    // Verify bid number
    require(bids[bidNumber].bidNumber != 0, "Bid not found");
    // Authentication, proving that A did intend to sign and submit the blinded bid in the first place
    require(verifyBid(bidNumber, tokenAddress, price, quantity, _blindedBid), "Invalid bid");
    // Ensure that the token address is a valid address
    require(tokenAddress != address(0) && tokenAddress != address(this), "Invalid Token Address");
    // Purchase price must be greater than 0
    require(price > 0, "Price must be greater than 0");
    // Purchase quantity must be greater than 0
    require(quantity > 0, "Quantity must be greater than 0");
    // Update bid data
    bids[bidNumber].tokenAddress = tokenAddress;
    bids[bidNumber].price = price;
    bids[bidNumber].quantity = quantity;
    bids[bidNumber].revealed = true;
    // Reveal the identity of the real buyer A
    bids[bidNumber].bidder = msg.sender;
    // Emit bid reveal event
    emit BidRevealed(bidNumber, tokenAddress, price, quantity, msg.sender);
}
```

This process achieves identity verification and order updating. Let's delve into the verifyBid() function to

```solidity
// Blind bid verification
// Verify agent B + verify buyer A
function verifyBid(
    uint256 bidNumber,
    address tokenAddress,
    uint256 price,
    uint256 quantity,
    bytes32 _blindedBid
) public view returns (bool) {
    // Verify if the signer is agent B
    if (getBuyer(_blindedBid, bids[bidNumber].signature) != bids[bidNumber].buyer) {
        return false;
    }
    // Verify if the blinded bid hash matches the input parameters
    if (keccak256(abi.encode(tokenAddress, price, quantity, msg.sender)) != bids[bidNumber].blindedBid) {
        return false;
    }
    return true;
}
```

Continuing to delve into the getBuyer() function:

```solidity
// Get the address of agent B
// Parameters: blinded bid hash, signature
// _blindedBid = keccak256(abi.encode(tokenAddress, price, quantity, realBuyerA))
// _signature = Signature of agent B for _blindedBid
function getBuyer(
    bytes32 _blindedBid,
    bytes memory _signature
) public pure returns (address) {
    // Value of r in the ECDSA signature
    bytes32 r;
    // Value of s in the ECDSA signature
    bytes32 s;
    // Value of v in the ECDSA signature
    uint8 v;
    assembly {
        // Read the value of r from the signature data
        r := mload(add(_signature, 32))
        // Read the value of s from the signature data
        s := mload(add(_signature, 64))
        // Read the value of v from the signature data, and restrict it to the range of 0~255
        v := and(mload(add(_signature, 65)), 255)
    }
    // Determine the chain the signature belongs to based on the value of v
    if (v < 27) {
        v += 27;
    }
    // Add prefix
    bytes32 prefixedBlindedBid = keccak256(abi.encodePacked("\x19Ethereum Signed Message:\n32", _blindedBid)
    // Verify signature using ECDSA and extract the address of the signer from the signature
    address buyer = ecrecover(prefixedBlindedBid, v, r, s);
    return buyer;
}
```

```
As can be seen from the above two functions, the verifyBid() function verifies both agent B's signature and
```

- Remove Buy Order

  During the blind bidding period, agent B can remove the buy order at any time.

```
// Remove a bid
// Can only be called by agent B to prevent A from being exposed
// Parameters: bid number
function removeBid(uint256 bidNumber) public {
    // Verify mode
    require(currentMode == Mode.BidOpening, "Not in BidOpening mode");
    // Verify bid number
    require(bids[bidNumber].bidNumber != 0, "Bid not found");
    // Authentication, only agent B can operate, to prevent A from being exposed
    require(bids[bidNumber].buyer == msg.sender, "Only the bid creator can remove the bid");
    // Delete the bid
    delete bids[bidNumber];
    // Decrement the total bid count
    bidsCount--;
}
```

- Retrieve Buy Order

  Anyone can obtain actual existing buy order information through the buy order number, but for unrevealed buy orders, only some meaningless data can be retrieved.

```
// Get bid information
function getBid(uint256 bidNumber) public view returns (Bid memory) {
    // Validate bid number
    require(bids[bidNumber].bidNumber != 0, "Bid not found");
    // Return bid information
    return bids[bidNumber];
}
```

Offer

```
// Sell order structure
struct Offer {
    // Token contract
    address tokenAddress;
    // Sell price
    uint256 price;
    // Quantity
    uint256 quantity;
    // Sell order number
    uint256 offerNumber;
    // Seller address
    address seller;
    // Whether the order has been matched
    bool matched;
}
```

- Create Sell Order

  During the Mode.Offer period, sellers can create sell orders. When creating, basic elements such as token address, price, and quantity will be checked. After the successful creation of a sell order, corresponding events will also be triggered.

```
// Create offer
// Parameters: token address, sell price, sell quantity
function addOffer(address tokenAddress, uint256 price, uint256 quantity) public {
    // Validate mode
    require(currentMode == Mode.Offer, "Not in Offer mode");
    // Ensure a valid address
    require(tokenAddress != address(0) && tokenAddress != address(this), "Invalid Token Address");
    // Sell price must be greater than 0
    require(price > 0, "Price must be greater than 0");
    // Sell quantity must be greater than 0
    require(quantity > 0, "Quantity must be greater than 0");
    // Increase global offer number
    lastOfferNumber++;
    // Increase offer count
    offersCount++;
    // Create offer
    offers[lastOfferNumber] = Offer(tokenAddress, price, quantity, lastOfferNumber, msg.sender, false);
    // Emit offer added event
    emit OfferAdded(tokenAddress, lastOfferNumber, price, quantity, msg.sender);
}
```

- Modify Sell Order

  During the Mode.Offer period, sellers can modify sell orders. When modifying a sell order, the order number and price will be checked. At the same time, the sell order is only allowed to have its price reduced, not increased. After a successful modification, corresponding events will also be triggered.

```
// Modify offer
// Parameters: offer number, sell price
function changeOffer(uint256 offerNumber, uint256 price) public {
    // Validate mode
    require(currentMode == Mode.Offer, "Not in Offer mode");
    // Authorization: only the offer creator can modify the offer
    require(offers[offerNumber].seller == msg.sender, "Only the offer creator can change the offer");
    // Sell price must be greater than 0
    require(price > 0, "Price must be greater than 0");
    // Seller can only lower the price, not raise it
    require(offers[offerNumber].price > price, "Price can only be decreased");
    // Update sell price
    offers[offerNumber].price = price;
    // Emit offer changed event
    emit OfferChanged(offerNumber, price, msg.sender);
}
```

- Remove Sell Order

  During the Mode.Offer period, sellers can remove sell orders. Only the creator of the sell order can remove their own order. After the successful removal, corresponding events will also be triggered.

```
// Remove offer
function removeOffer(uint256 offerNumber) public {
    // Validate mode
    require(currentMode == Mode.Offer, "Not in Offer mode");
    // Authorization: only the offer creator can remove the offer
    require(offers[offerNumber].seller == msg.sender, "Only the offer creator can remove the offer");
    // Delete offer
    delete offers[offerNumber];
    // Decrease offer count
    offersCount--;
    // Emit offer removed event
    emit OfferRemoved(offerNumber, msg.sender);
}
```

- Retrieve Sell Order Information
  Anyone can retrieve actual existing sell order information at any time, including token address, price, quantity, sell order number, seller's address, and whether the order has been executed.

```
// Get offer information
function getOffer(uint256 offerNumber) public view returns (Offer memory) {
    // Validate offer number
    require(offers[offerNumber].offerNumber != 0, "Offer not found");
    // Return offer information
    return offers[offerNumber];
}
```

orderMaching

- Matching Elements
  1.Sell order price must be less than or equal to the buy order price.
  2.Buy and sell orders are processed in order of time, with the longest waiting time given priority. In other words, they are sorted first.
  3.Orders that have been completely executed are skipped.
  4.Unrevealed buy orders are also skipped.
  5.When matching a sell offer at price p with a buy order at price q ≥ p, the buyer always pays the lower price p.
  6.Negative account balances are not allowed. If executing a match causes the balance of the buyer or seller to become negative, the match will not be executed.
  7.Buyers and sellers cannot be the same account.
  8.The system must be in Mode.Matching mode.
  9.After a buy or sell order is completely executed, corresponding events will be triggered and the order will be marked as completely executed.
- Matching Process

```
// Sort all sell orders in ascending order by price.
// This allows buy orders to prioritize matching with the lowest-priced sell orders, and when two sell o
// This is a private function that can only be called within the contract.
function sortOffers() private view returns (uint256[] memory) {
    uint256[] memory sortedOfferIndices = new uint256[](lastOfferNumber);
    // Initialize the sorting array.
    for (uint256 i = 0; i < lastOfferNumber; i++) {
        sortedOfferIndices[i] = i + 1;
    }
    // Use bubble sort to sort the sell orders.
    for (uint256 i = 0; i < lastOfferNumber - 1; i++) {
        for (uint256 j = 0; j < lastOfferNumber - i - 1; j++) {
            bool needSwap = false;
            // Compare prices.
            if (offers[sortedOfferIndices[j]].price > offers[sortedOfferIndices[j + 1]].price) {
                needSwap = true;
            // If the prices are equal, compare the order number (prioritizing the one with the longest
            } else if (offers[sortedOfferIndices[j]].price == offers[sortedOfferIndices[j + 1]].price)
                if (offers[sortedOfferIndices[j]].offerNumber > offers[sortedOfferIndices[j + 1]].offerN
                    needSwap = true;
                }
            }
            if (needSwap) {
                // Swap elements.
                (sortedOfferIndices[j], sortedOfferIndices[j + 1]) = (sortedOfferIndices[j + 1], sortedO
            }
        }
    }
    return sortedOfferIndices;
}

// Order Matching
// Match all sell orders in order with buy orders until the buy order is fully executed.
function orderMaching() public {
    // Verify the mode
    require(currentMode == Mode.Matching, "Not in Matching mode");

    // Get the sorted sell order indices
    uint256[] memory sortedOfferIndices = sortOffers();

    // Loop through all buy orders, starting from order number 1 to prioritize orders with longer wait t
    for (uint256 i = 1; i <= lastBidNumber; i++) {
        // Skip buy orders that have already been fully executed
        if (bids[i].matched == true) continue;
        // Skip unrevealed buy orders
        if (bids[i].revealed == false) continue;

        // Loop through the sorted sell orders
        for (uint256 k = 0; k < sortedOfferIndices.length; k++) {
            // Current sell order index
            uint256 j = sortedOfferIndices[k];

            // If the buy order has already been fully executed during the matching process, stop matchi
            if (bids[i].matched == true) break;

            // If neither the buy nor the sell order has been fully executed
            // And the token types of both the buy and sell orders are the same
            // And the sell order price <= the buy order price, and the sell order quantity > 0, and the
            // And the buyer and seller cannot be the same person
            if (
                bids[i].matched == false &&
```

```
                offers[j].matched == false &&
                offers[j].tokenAddress == bids[i].tokenAddress &&
                offers[j].price <= bids[i].price &&
                offers[j].quantity > 0 && bids[i].quantity > 0 &&
                offers[j].seller != bids[i].bidder
            ) {
                // The trade quantity = sell order quantity < buy order quantity ? sell order quantity
                uint256 quantity = offers[j].quantity < bids[i].quantity ? offers[j].quantity : bids[i]

                // The trade cost = trade quantity * sell order price
                // When a sell offer at price p matches a buy order at price q ≥ p, the buyer always pay
                uint256 cost = quantity * offers[j].price / 10 ** 18;

                // Prevent account balances from going negative. If matching would result in a negative
                // If the seller does not have enough tokens, skip this sell order and move on to the ne
                if (accounts[offers[j].seller].tokenBalances[offers[j].tokenAddress] < quantity) continu
                // If the buyer does not have enough ETH, stop and match with the next buy order direct
                if (accounts[bids[i].bidder].balance < cost) break;

                // Seller receives ETH
                accounts[offers[j].seller].balance += cost;
                // Seller deducts tokens
                accounts[offers[j].seller].tokenBalances[offers[j].tokenAddress] -= quantity;
                // Sell order quantity decreases
                offers[j].quantity -= quantity;
                // If the sell order quantity decreases to 0, mark the sell order as fully executed
                if (offers[j].quantity == 0) {
                    // Mark as fully executed
                    offers[j].matched = true;
                    // Decrease the total sell order count
                    offersCount--;
                }

                // The actual buyer A deducts ETH
                accounts[bids[i].bidder].balance -= cost;
                // The actual buyer A receives tokens
                accounts[bids[i].bidder].tokenBalances[bids[i].tokenAddress] += quantity;
                // Buy order quantity decreases
                bids[i].quantity -= quantity;
                // If the buy order quantity decreases to 0, mark the buy order as fully executed
                if (bids[i].quantity == 0) {
                    // Mark the buy order as fully executed
                    bids[i].matched = true;
                    // Decrease the total buy order count
                    bidsCount--;
                }

                // Emit the trade event
                // Publicize the sell order number, buy order number, price, quantity, seller, and actua
                emit Trade(offers[j].tokenAddress, offers[j].offerNumber, bids[i].bidNumber, offers[j].p
            }
        }
    }
}
```

Token Contract

The Token contract inherits the ERC20 contract by default, and only needs to mint tokens in the constructor for testing purposes.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract Token is ERC20 {
    constructor(string memory name, string memory symbol) ERC20(name, symbol) {
        // By default, 1 billion tokens will be minted.
        _mint(msg.sender, 100000000 * (10**18));
    }
}
```

Deployment Script

The deployment script deploys an ERC20 token contract named MTK and the DutchMarket contract.

The deployer authorizes the DutchMarket contract to manage all the tokens and transfers 100,000 tokens to seller 1 and seller 2 respectively.

The sellers also authorize the DutchMarket contract to manage their tokens.

This is done to facilitate testing and avoid the need for repeated transfers and authorizations.

The relevant code is as follows:

```
const Token = artifacts.require("Token")
const DutchMarket = artifacts.require("DutchMarket")

module.exports = async function(deployer, network, accounts) {
    // Deploy an ERC20 token contract
    await deployer.deploy(Token, 'My Token', 'MTK');
    const tokenInstance = await Token.deployed()

    // Deploy a Dutch Market contract
    await deployer.deploy(DutchMarket);
    const dutchMarketInstance = await DutchMarket.deployed()
    // Authorize the Dutch Market contract for the ERC20 token
    await tokenInstance.approve(dutchMarketInstance.address, web3.utils.toWei("100000000", "ether"))

    // Transfer 100,000 tokens to Seller 1
    await tokenInstance.transfer(accounts[1], web3.utils.toWei("100000", "ether"))
    // Seller 1 authorizes the Dutch Market contract for all of their tokens
    await tokenInstance.approve(dutchMarketInstance.address, web3.utils.toWei("100000000", "ether"), { from:

    // Transfer 100,000 tokens to Seller 2
    await tokenInstance.transfer(accounts[2], web3.utils.toWei("100000", "ether"))
    // Seller 2 authorizes the Dutch Market contract for all of their tokens
    await tokenInstance.approve(dutchMarketInstance.address, web3.utils.toWei("100000000", "ether"), {from:
}
```

Testing

The system includes five test cases that can be run with the truffle test command.

1.1_deposit_withdraw_eth.js tests the deposit and withdrawal of ETH, by asserting the account balance after depositing and withdrawing ETH.

```
const DutchMarket = artifacts.require("DutchMarket");
const { toWei, fromWei } = web3.utils;

contract('DutchMarket', (accounts) => {
    it('deposit 1 ETH in the contract escrow account', async () => {
        const dutchMarketInstance = await DutchMarket.deployed();
        // Deposit 1 ETH.
        const amount = toWei("1", "ether");
        await dutchMarketInstance.depositETH({ from: accounts[0], value: amount });
        // Check the current balance of ETH held in the contract's account
        const balance = await dutchMarketInstance.getAccountBalance({
            from: accounts[0]
        });
        // Check if it is equal to 1.
        assert.equal(fromWei(balance, "ether"), 1, "successful deposit 1 ETH");
    });
    it('withdraw 1 ETH in the first account', async () => {
        const dutchMarketInstance = await DutchMarket.deployed();
        // Withdraw 1 ETH.
        const amount = toWei("1", "ether");
        await dutchMarketInstance.withdrawETH(amount, { from: accounts[0] });
        // Check the current balance of ETH held in the contract's account
        const balance = await dutchMarketInstance.getAccountBalance({ from: accounts[0] });
        // Check if it is equal to 0
        assert.equal(fromWei(balance, "ether"), 0, "successful withdraw 1 ETH");
    });
});
```

2. `2_deposit_withdraw_token.js` tests the deposit and withdrawal of tokens, by asserting the account token balance after depositing and withdrawing tokens.

```
const Token = artifacts.require("Token");
const DutchMarket = artifacts.require("DutchMarket");
const { toWei } = web3.utils;

contract('DutchMarket', (accounts) => {
    it('deposit 1000 Token in the contract escrow account', async () => {
        const tokenInstance = await Token.deployed();
        const dutchMarketInstance = await DutchMarket.deployed();
        // Deposit 1000 tokens.
        const amount = toWei("1000", "ether");
        await dutchMarketInstance.depositToken(tokenInstance.address, amount, { from: accounts[0] });
        // Check the current balance of tokens held in the contract's account
        const balance = await dutchMarketInstance.getAccountTokenBalance(tokenInstance.address, {
            from: accounts[0]
        });
        // Check if it is equal to 1000
        assert.equal(balance, amount, "successful deposit 1000 Token");
    });
    it('Withdraw 1000 Token in the first account', async () => {
        const tokenInstance = await Token.deployed();
        const dutchMarketInstance = await DutchMarket.deployed();
        // Withdraw 1000 tokens
        const amount = toWei("1000", "ether");
        await dutchMarketInstance.withdrawToken(tokenInstance.address, amount, { from: accounts[0] });
        // Check the current balance of tokens held in the contract's account
        const balance = await dutchMarketInstance.getAccountTokenBalance(tokenInstance.address, {
            from: accounts[0]
        });
        // Check if it is equal to 0
        assert.equal(balance, 0, "successful withdraw 1000 Token");
    });
});
```

3. `3_add_change_remove_offer.js` tests the creation, modification, and removal of sell orders, and asserts the correctness of the logic.

```
const Token = artifacts.require("Token");
const DutchMarket = artifacts.require("DutchMarket");

const { toWei } = web3.utils;

contract('DutchMarket', (accounts) => {
    it('Add a offer order of 1 ETH for 1000 Token', async () => {
        const tokenInstance = await Token.deployed();
        const dutchMarketInstance = await DutchMarket.deployed();
        // Deposit 1000 tokens into the contract's account
        await dutchMarketInstance.setMode(0, { from: accounts[0] });
        await dutchMarketInstance.depositToken(
            tokenInstance.address,
            toWei("1000", "ether"),
            { from: accounts[0] }
        );
        // Check the number of sell orders
        await dutchMarketInstance.setMode(1, { from: accounts[0] });
        const offersCount1 = await dutchMarketInstance.offersCount({ from: accounts[0] });
        // Add a sell order for 1000 tokens at a price of 1 ETH
        await dutchMarketInstance.addOffer(
            tokenInstance.address,
            toWei("1", "ether"),
            toWei("1000", "ether"),
            { from: accounts[0] }
        );
        // Check the number of sell orders again
        const offersCount2 = await dutchMarketInstance.offersCount({ from: accounts[0] });
        // Check if the number of sell orders has increased by 1
        assert.equal(offersCount1.toNumber() + 1, offersCount2.toNumber(), "successful addOffer 1 ETH for 10
    });
    it('change offer order of 0.9 ETH for 1000 Token', async () => {
        const dutchMarketInstance = await DutchMarket.deployed();
        // Get the global sell order ID
        const lastOfferNumber = await dutchMarketInstance.lastOfferNumber({ from: accounts[0] });
        // Modify the sell order price
        const price = toWei("0.9", "ether");
        await dutchMarketInstance.changeOffer(lastOfferNumber, price, { from: accounts[0] })
        // Check the sell order price
        const offer = await dutchMarketInstance.getOffer(lastOfferNumber, { from: accounts[0] });
        // Check if the sell order price is equal to 0.9
        assert.equal(Number(offer[1]), Number(price), "successful change a offer order of 0.9 ETH for 1000 T
    });
    it('remove offer order', async () => {
        const dutchMarketInstance = await DutchMarket.deployed();
        // Get the global sell order ID
        const lastOfferNumber = await dutchMarketInstance.lastOfferNumber({ from: accounts[0] });
        // Get the number of sell orders
        const offersCount1 = await dutchMarketInstance.offersCount({ from: accounts[0] });
        // Delete this sell order
        await dutchMarketInstance.removeOffer(lastOfferNumber, { from: accounts[0] })
        // Get the number of sell orders again.
        const offersCount2 = await dutchMarketInstance.offersCount({ from: accounts[0] });
        // Check if the number of sell orders has decreased by 1
        assert.equal(Number(offersCount1) - 1, Number(offersCount2), "successful remove offer order");
    });
});
```

4. `4_add_reveal_remove_bid.js` tests the creation, reveal, and removal of buy orders, and asserts the correctness of the logic.

```
const Token = artifacts.require("Token");
const DutchMarket = artifacts.require("DutchMarket");

const ethers = require('ethers');
const { toWei } = web3.utils;

// Get the hash
// Parameters: token address, price, quantity, and actual buyer
function getHash(tokenAddress, price, quantity, trueBuyer) {
    // Calculate the bytes32 value for the actual buyer
    const trueBuyerBytes32 = ethers.utils.padZeros(ethers.utils.arrayify(trueBuyer), 32);
    const encodedData = ethers.utils.defaultAbiCoder.encode(
        ['address', 'uint256', 'uint256', 'bytes32'],
        [tokenAddress, price, quantity, trueBuyerBytes32]
    );
    const hash = ethers.utils.keccak256(encodedData);
    return hash;
}

// Get the signature
// Parameters: hash and agent
async function getSignature(messageHash, buyer) {
    return await web3.eth.sign(messageHash, buyer);
}

contract('DutchMarket', (accounts) => {
    it('Add a bid order of 1 ETH for 10 Token', async () => {
        const tokenInstance = await Token.deployed();
        const dutchMarketInstance = await DutchMarket.deployed();
        // Check the number of buy orders
        await dutchMarketInstance.setMode(2, { from: accounts[0] });
        const bidsCount1 = await dutchMarketInstance.bidsCount({ from: accounts[0] });
        // Add a buy order for 10 tokens at a price of 1 ETH
        // Understood. Based on the assumption that the agent B is accounts[0] and the actual buyer A is acc
        const price = toWei("1", "ether");
        const quantity = toWei("10", "ether");
        // Calculate the hash that includes the order information and the actual buyer A
        const messageHash = getHash(tokenInstance.address, price, quantity, accounts[9]);
        // Calculate the signature for agent B
        const signature = await getSignature(messageHash, accounts[0]);
        // Buy order created. Anyone monitoring the blockchain cannot see my data, as I only submitted a has
        await dutchMarketInstance.addBid(
            messageHash,
            signature,
            { from: accounts[0] }
        );
        // Check the number of buy orders again
        const bidsCount2 = await dutchMarketInstance.bidsCount({ from: accounts[0] });
        // Check if the number of buy orders has increased by 1
        assert.equal(bidsCount1.toNumber() + 1, bidsCount2.toNumber(), "successful addBid");
    });
    it('Reveal a bid order', async () => {
        const tokenInstance = await Token.deployed();
        const dutchMarketInstance = await DutchMarket.deployed();
        // Get the global buy order ID
        const lastBidNumber = await dutchMarketInstance.lastBidNumber({ from: accounts[0] });
        // Price and quantity
        const price = toWei("1", "ether");
        const quantity = toWei("10", "ether");
        // Calculate the hash that includes the order information and the actual buyer A
        const messageHash = await getHash(tokenInstance.address, price, quantity, accounts[9]);
```

```
        // Simulate actual buyer A revealing the buy order
        await dutchMarketInstance.setMode(2, { from: accounts[0] });
        await dutchMarketInstance.BidReveal(
            lastBidNumber,
            tokenInstance.address,
            price,
            quantity,
            messageHash,
            { from: accounts[9] }
        );
        // Check the information of the buy order
        const bid = await dutchMarketInstance.getBid(lastBidNumber.toNumber(), { from: accounts[0] });
        // Check if the price and quantity have been updated, and if the buy order has been revealed
        assert.equal(bid[1], price, "successful verify price");
        assert.equal(bid[2], quantity, "successful verify quantity");
        assert.equal(bid[8], true, "successful verify revealed");
    });
    it('remove bid order', async () => {
        const dutchMarketInstance = await DutchMarket.deployed();
        // Get the global sell order ID
        const lastBidNumber = await dutchMarketInstance.lastBidNumber({ from: accounts[0] });
        // Check the number of buy orders
        const bidsCount1 = await dutchMarketInstance.bidsCount({ from: accounts[0] });
        // Delete this buy order
        await dutchMarketInstance.removeBid(lastBidNumber, { from: accounts[0] })
        // Check the number of buy orders again
        const bidsCount2 = await dutchMarketInstance.bidsCount({ from: accounts[0] });
        // Check if the number of buy orders has decreased by 1
        assert.equal(bidsCount1.toNumber() - 1, bidsCount2.toNumber(), "successful remove bid order");
    });
});
```

5. `5_order_matching.js` : Test the logic of order matching by creating buy and sell orders, executing matching, and asserting that the remaining balance of accounts is correct. This test case simulates 2 buyers and 2 sellers. Relevant code and analysis are as follows:

```
const Token = artifacts.require("Token");
const DutchMarket = artifacts.require("DutchMarket");

const ethers = require('ethers');
const { toWei, fromWei } = web3.utils;

const num1 = toWei("1", "ether");
const num2 = toWei("2", "ether");
const num5 = toWei("5", "ether");
const num10 = toWei("10", "ether");
const num20 = toWei("20", "ether");
const num30 = toWei("30", "ether");
const num40 = toWei("40", "ether");
const num60 = toWei("60", "ether");

// Get the hash
// Parameters: token address, price, quantity, and actual buyer
function getHash(tokenAddress, price, quantity, trueBuyer) {
    const trueBuyerBytes32 = ethers.utils.padZeros(ethers.utils.arrayify(trueBuyer), 32);
    const encodedData = ethers.utils.defaultAbiCoder.encode(
        ['address', 'uint256', 'uint256', 'bytes32'],
        [tokenAddress, price, quantity, trueBuyerBytes32]
    );
    const hash = ethers.utils.keccak256(encodedData);
    return hash;
}

// Get the signature
// Parameters: hash and agent
async function getSignature(messageHash, buyer) {
    return await web3.eth.sign(messageHash, buyer);
}

contract('DutchMarket', (accounts) => {
    it('Match 2 bid orders and 2 offer orders', async () => {
        const tokenInstance = await Token.deployed();
        const dutchMarketInstance = await DutchMarket.deployed();
        // Order Book:
        // Sell2 2 10
        // Sell1 1 10
        // Buy1 2 5
        // Buy2 2 10

        // Theoretical transaction steps:
        // 1. Buy 1 matches with Sell 1 first, and completes a transaction of 5 TOKENs at a price of 1 ETH.
        // 2. Buy 1 spends a total of 1 * 5 = 5 ETH.
        // 3. Buy 2 matches with Sell 1 next, and completes a transaction of the remaining 5 TOKENs of Sell
        // 4. Buy 2 matches with Sell 2 and completes a transaction of 5 TOKENs at a price of 2 ETH. Buy 2 s
        // 5. Buy 2 spends a total of 1 * 5 + 2 * 5 = 15 ETH.

        // 6. Contract account balance:
        // 7. Sell 1 deposits 20 tokens, sells 10 tokens, and has 10 tokens + 10 ETH remaining.
        // 8. Sell 2 deposits 30 tokens, sells 5 tokens, and has 25 tokens + 10 ETH remaining.
        // 9. Buy 1 deposits 40 ETH, spends 5 ETH, and has 35 ETH + 5 tokens remaining.
        // 10. Buy 2 deposits 60 ETH, spends 15 ETH, and has 45 ETH + 10 tokens remaining.

        //-------------------------------------Seller 1-------------------------------------
        // Seller 1 deposits 20 tokens into the contract's account
        await dutchMarketInstance.setMode(0, { from: accounts[0] });
        await dutchMarketInstance.depositToken(tokenInstance.address, num20,{
            from: accounts[1],
```

```
});
await dutchMarketInstance.setMode(1, { from: accounts[0] });
// Seller 1 creates a sell order for 10 tokens at a price of 1 ETH
await dutchMarketInstance.addOffer(
    tokenInstance.address,
    num1,
    num10,
    { from: accounts[1] }
);
//-------------------------------------Seller 1-------------------------------------

//-------------------------------------Seller 2-------------------------------------
// Seller 2 deposits 30 tokens into the contract's account
await dutchMarketInstance.setMode(0, { from: accounts[0] });
await dutchMarketInstance.depositToken(tokenInstance.address, num30, {
    from: accounts[2],
});
// Seller 2 creates a sell order for 10 tokens at a price of 2 ETH
await dutchMarketInstance.setMode(1, { from: accounts[0] });
await dutchMarketInstance.addOffer(
    tokenInstance.address,
    num2,
    num10,
    { from: accounts[2] }
);
//-------------------------------------Seller 2-------------------------------------

//-------------------------------------Buyer 1--------------------------------------
// Actual buyer 1 is accounts[3], and the agent is accounts[8]
// Actual buyer 1 deposits 40 ETH into the contract's ETH account
await dutchMarketInstance.setMode(0, { from: accounts[0] });
await dutchMarketInstance.depositETH({
    from: accounts[3],
    value: num40,
});
await dutchMarketInstance.setMode(2, { from: accounts[0] });
// Calculate the hash that includes the order information and actual buyer 1.
const hash1 = getHash(tokenInstance.address, num2, num5, accounts[3]);
// Calculate the signature for agent accounts[8]
const signature1 = await getSignature(hash1, accounts[8]);
// Agent accounts[8] creates a buy order for 5 tokens at a price of 2 ETH
await dutchMarketInstance.addBid(
    hash1,
    signature1,
    { from: accounts[8] }
);
// Get the global buy order ID
const lastBidNumber1 = await dutchMarketInstance.lastBidNumber({ from: accounts[0] });
// Buyer 1 reveals the buy order.
await dutchMarketInstance.BidReveal(
    lastBidNumber1,
    tokenInstance.address,
    num2,
    num5,
    hash1,
    { from: accounts[3] }
);
//-------------------------------------Buyer 1--------------------------------------

//-------------------------------------Buyer 2--------------------------------------
// Actual buyer 2 is accounts[4], and the agent is accounts[7]
// Actual buyer 2 deposits 60 ETH into the contract's ETH account
```

```
await dutchMarketInstance.setMode(0, { from: accounts[0] });
await dutchMarketInstance.depositETH({
    from: accounts[4],
    value: num60,
});
await dutchMarketInstance.setMode(2, { from: accounts[0] });
// Calculate the hash that includes the order information and actual buyer 2
const hash2 = getHash(tokenInstance.address, num2, num10, accounts[4]);
// Calculate the signature for agent accounts[7]
const signature2 = await getSignature(hash2, accounts[7]);
// Agent accounts[7] creates a buy order for 10 tokens at a price of 2 ETH
await dutchMarketInstance.addBid(
    hash2,
    signature2,
    { from: accounts[7] }
);
// Get the global buy order ID
const lastBidNumber2 = await dutchMarketInstance.lastBidNumber({ from: accounts[0] });
// Buyer 2 reveals the buy order
await dutchMarketInstance.BidReveal(
    lastBidNumber2,
    tokenInstance.address,
    num2,
    num10,
    hash2,
    { from: accounts[4] }
);
//--------------------------------------Buyer 2-----------------------------------

//--------------------------------------Order matching----------------------------------
await dutchMarketInstance.setMode(3, { from: accounts[0] });
await dutchMarketInstance.orderMaching();
//--------------------------------------Order matching----------------------------------

//--------------------------------------Verify Seller 1---------------------------------
// Check Seller 1's balance in the contract's ETH account. The balance should be 10 ETH in theory
const seller1ETHBalance = await dutchMarketInstance.getAccountBalance({ from: accounts[1] });
assert.equal(fromWei(seller1ETHBalance, "ether"), 10, "seller1 ETH balance is 10 ETH");
// Check Seller 1's balance in the contract's token account. The balance should be 10 tokens in thec
const seller1TokenBalance = await dutchMarketInstance.getAccountTokenBalance(
    tokenInstance.address,
    { from: accounts[1] }
);
assert.equal(fromWei(seller1TokenBalance, "ether"), 10, "seller1 Token balance is 10");
//--------------------------------------Verify Seller 1---------------------------------

//--------------------------------------Verify Seller 2---------------------------------
// Check Seller 2's balance in the contract's ETH account. The balance should also be 10 ETH in thec
const seller2ETHBalance = await dutchMarketInstance.getAccountBalance({ from: accounts[2] });
assert.equal(fromWei(seller2ETHBalance, "ether"), 10, "seller1 ETH balance is 10 ETH");
// Check Seller 2's balance in the contract's token account. The balance should be 25 tokens in thec
const seller2TokenBalance = await dutchMarketInstance.getAccountTokenBalance(
    tokenInstance.address,
    { from: accounts[2] }
);
assert.equal(fromWei(seller2TokenBalance, "ether"), 25, "seller2 Token balance is 25");
//--------------------------------------Verify Seller 2---------------------------------

//--------------------------------------Verify Buyer 1----------------------------------
// Check Buyer 1's balance in the contract's ETH account. The balance should be 35 ETH in theory
const buyer1ETHBalance = await dutchMarketInstance.getAccountBalance({ from: accounts[3] });
assert.equal(fromWei(buyer1ETHBalance, "ether"), 35, "buyer1 ETH balance is 35 ETH");
```

```
        // Check Buyer 1's balance in the contract's token account. The balance should be 5 tokens in theory
        const buyer1TokenBalance = await dutchMarketInstance.getAccountTokenBalance(
            tokenInstance.address,
            { from: accounts[3] }
        );
        assert.equal(fromWei(buyer1TokenBalance, "ether"), 5, "buyer1 Token balance is 5");
        //-------------------------------------Verify Buyer 1---------------------------------

        //-------------------------------------Verify Buyer 2---------------------------------
        // Check Buyer 2's balance in the contract's ETH account. The balance should be 45 ETH in theory
        const buyer2ETHBalance = await dutchMarketInstance.getAccountBalance({ from: accounts[4] });
        assert.equal(fromWei(buyer2ETHBalance, "ether"), 45, "buyer2 ETH balance is 45 ETH");
        // Check Buyer 2's balance in the contract's token account. The balance should be 20 tokens in theor
        const buyer2TokenBalance = await dutchMarketInstance.getAccountTokenBalance(
            tokenInstance.address,
            { from: accounts[4] }
        );
        assert.equal(fromWei(buyer2TokenBalance, "ether"), 10, "buyer2 Token balance is 10");
        //-------------------------------------Verify Buyer 2---------------------------------
    });
});
```

Test Results.

```
Contract: DutchMarket
✓ deposit 1 ETH in the contract escrow account (57ms)
✓ withdraw 1 ETH in the first account (67ms)
```

Contract: DutchMarket
✓ deposit 1000 Token in the contract escrow account (92ms)
✓ Withdraw 1000 Token in the first account (98ms)

Contract: DutchMarket
✓ Add a offer order of 1 ETH for 1000 Token (233ms)
✓ change offer order of 0.9 ETH for 1000 Token (68ms)
✓ remove offer order (82ms)

Contract: DutchMarket
✓ Add a bid order of 1 ETH for 10 Token (271ms)
✓ Reveal a bid order (227ms)
✓ remove bid order (92ms)

Contract: DutchMarket
✓ Match 2 bid orders and 2 offer orders (1099ms)

11 passing (3s)

Cost analysis

1. The biggest operating cost of the application is gas fees. Gas fees depend on the congestion of the current blockchain network, and the higher the congestion, the higher the fees.
2. The order matching function `orderMatching()` in this system contains multiple `for` loops. This type of matching algorithm has a time complexity of O(n^2). Therefore, when there are a large number of orders, the

running time will increase accordingly, and the overall operating cost of the application will also continue to increase.

Security Considerations

When using this system, users should pay attention to the following security issues:

- Confirm transaction information: Before conducting a transaction, the transaction information should be carefully checked to ensure its accuracy.
- Avoid using public Wi-Fi: Public Wi-Fi is easily susceptible to hacker attacks, so it should be avoided when using the system.
- Protect private keys: Private keys are the key to accessing user accounts and must be protected properly to prevent theft.
- Guard against viruses: When using the system, anti-virus software should be installed to prevent virus infections.

To avoid Solidity security vulnerabilities, we can take the following measures:

- Avoid using outdated Solidity versions. This system uses Solidity 0.8, which can avoid known security vulnerabilities in the past, such as the SafeMath overflow. In version 0.8.0, this issue has been addressed at the language level: if an integer overflow occurs, the transaction will be directly reverted. Before version 0.8.0, you need to use the OpenZeppelin SafeMath library.
- Use library functions: Using library functions can reduce duplicate code and reduce the risk of vulnerabilities.
- Ensure contract state updates: Before modifying the contract state, necessary checks and validations should be performed.
- Strictly verify user input data: For example, the price, quantity, address, and other information submitted by the user should be strictly verified to ensure that they cannot be negative, etc.
- Strictly verify function callers: In the contract, the identity of function callers should be strictly verified to ensure that only the contract owner can call certain functions.
- Strictly verify function call parameters: In the contract, function call parameters should be strictly verified to ensure that the value of the parameters does not lead to inconsistencies in the contract state.
- Strictly verify function call return values: In the contract, function call return values should be strictly verified to ensure that the function call is successful.
- To prevent reentrancy attacks:
- This system defines a custom noReentrancy modifier and uses it in `withdrawETH` and `withdrawToken` functions to prevent reentrancy attacks during withdrawals.

```
// Reentrancy lock
bool private locked;
modifier noReentrancy() {
    require(!locked, "Reentrant call");
    locked = true;
    _;
    locked = false;
}

......

// Token withdrawal
function withdrawToken(address tokenAddress, uint256 amount) public noReentrancy

......

// ETH withdrawal
function withdrawETH(uint256 amount) public noReentrancy
```

Is Ethereum suitable for this program?

The Ethereum platform is suitable for this application because it provides a distributed computing platform for creating smart contracts and decentralized applications. Additionally, Ethereum supports secure and reliable execution of smart contracts and has high scalability and transparency. However, it is important to note that Ethereum's gas fees and transaction fees may increase operating costs, so these cost factors need to be carefully considered.

Off-chain computing.

Before buyer B submits a blind bid, the real buyer A needs to perform a signature calculation on B's blind bid. This calculation is done off-chain, because if it were done on-chain, it would expose buyer A's blind bid and compromise its confidentiality.

Signature construction

The code for signature construction is in `test/4_add_reveal_remove_bid.js` , as follows:

```
// Get the hash
// Parameters: token address, price, quantity, and actual buyer
function getHash(tokenAddress, price, quantity, trueBuyer) {
    // Calculate the bytes32 value for the actual buyer
    const trueBuyerBytes32 = ethers.utils.padZeros(ethers.utils.arrayify(trueBuyer), 32);
    const encodedData = ethers.utils.defaultAbiCoder.encode(
        ['address', 'uint256', 'uint256', 'bytes32'],
        [tokenAddress, price, quantity, trueBuyerBytes32]
    );
    const hash = ethers.utils.keccak256(encodedData);
    return hash;
}

// Get the signature
// Parameters: hash and agent
async function getSignature(messageHash, buyer) {
    return await web3.eth.sign(messageHash, buyer);
}

......

// Calculate the hash that includes the order information and the actual buyer A
const messageHash = getHash(tokenInstance.address, price, quantity, accounts[9]);
// Calculate the signature for agent B
const signature = await getSignature(messageHash, accounts[0]);
// Buy order created. Anyone monitoring the blockchain cannot see my data, as I only submitted a hash and a
await dutchMarketInstance.addBid(
    messageHash,
    signature,
    { from: accounts[0] }
);

......
```

Signature Verification

The code for signature verification is in the `verifyBid` function in `contracts/DutchMarket.sol`, as shown below:

```
// Blind bid verification
// Verify agent B + verify buyer A
function verifyBid(
    uint256 bidNumber,
    address tokenAddress,
    uint256 price,
    uint256 quantity,
    bytes32 _blindedBid
) public view returns (bool) {
    // Verify if the signer is agent B
    if (getBuyer(_blindedBid, bids[bidNumber].signature) != bids[bidNumber].buyer) {
        return false;
    }
    // Verify if the blinded bid hash matches the input parameters
    if (keccak256(abi.encode(tokenAddress, price, quantity, msg.sender)) != bids[bidNumber].blindedBid)
        return false;
    }
    return true;
}
```