Decentralized Multi-Lottery Management System with Smart Contracts

Yakup Altay, *B.A. Student, Boğaziçi University Department of Economics*
Tekin Orkun Hengirmen, *M.Sc. Student, Boğaziçi University Department of Electrical-Electronics Engineering*

I. Introduction

The **CompanyLotteries** smart contract project was developed to create and manage a multi-lottery system for a company. The system allows the user to create and operate autonomous lotteries on the Ethereum blockchain. Each lottery follows a well-defined lifecycle, consisting of a **purchase phase** and a **reveal phase**, to ensure fairness and transparency.

A. Purpose

The primary objective of the project is to create a decentralized platform that:

- Enables users to purchase lottery tickets using ERC20 tokens.
- Automatically selects winners based on random numbers revealed by participants.
- Handles ticket refunds if minimum participation criteria are not met.
- Facilitates transparent, trustless management of lottery operations.

B. Key Features

The **CompanyLotteries** project emphasizes autonomy, transparency, flexibility, and scalability as its core features. Lotteries operate independently after their creation, requiring no manual intervention once initialized. Transparency is ensured through random numbers for winner selection, fostering fairness and eliminating bias. The platform is highly flexible, enabling users to configure parameters such as ticket price, the number of winners, and lottery duration.

Key functionalities include lottery creation, where users define essential parameters, and ticket purchasing, allowing up to 30 tickets per transaction with hashed random numbers for validation. Participants later reveal their numbers during the reveal phase, which determines the winners. The system also manages refunds for canceled lotteries and facilitates proceeds withdrawal from successful ones.

C. Significance

This project highlights the potential of blockchain technology in creating fair and efficient lottery systems. By leveraging smart contract automation, it eliminates the need for intermediaries, ensuring transparency and trust among participants. This decentralized approach exemplifies how blockchain can transform traditional processes into trustless and efficient systems.

This report is prepared for the CMPE 483 Sp. Top. in CMPE Blockchain Programming course taught at Boğaziçi University in term 2024/2025-1, lectured by Prof. Dr. Can Özturan.
Report submitted on December 4, 2024.

II. Contract Structure

The smart contract is structured with the following key components:

1) State Variables:
   - Store the details of each lottery (e.g., ticket prices, end times, number of winners).
   - Maintain mappings for ticket ownership, random numbers, and lottery statuses.
2) Events:
   - *LotteryCreated*: Emitted when a lottery is created.
   - *TicketPurchased*: Emitted when tickets are purchased.
   - *TicketRefunded*: Emitted when a ticket refund occurs.
   - *NewPaymentTokenSet*: Emitted when a new payment token is set.
   - *LotteryCancelled*: Emitted for canceled lotteries.
   - *LotteryFinalized*: Emitted when a lottery is finalized.
3) Main Functions:
   The contract includes several key functions, grouped based on their purpose:
   a) Lottery Management:
      - *getCurrentLotteryNo*(): public function. Retrieves the current active lottery number.
      - *createLottery*: public function. Creates a new lottery with specified parameters, including the start time, number of tickets, winners, and associated metadata.
      - *getLotteryInfo*: public function. Retrieves detailed information about a specific lottery.
      - *getLotteryURL*: public function. Returns the URL associated with the specified lottery for additional information.
      - *getLotterySales*: public function. Retrieves the total number of tickets sold for the specified lottery.
      - *finalizeLottery*: public function. Finalizes the lottery, determines winners, and completes the lottery lifecycle.

b) Ticket Operations:
- *getIthPurchasedTicket*: public function. Retrieves the details of the i-th ticket purchased for a given lottery.
- *checkIfMyTicketWon*: public function. Checks if a specific ticket belonging to the caller is among the winning tickets for a lottery.
- *checkIfAddressTicketWon*: public function. Verifies if a specific ticket belonging to the given address has won in the lottery.
- *getIthWinningTicket*: public function. Retrieves the details of the i-th winning ticket for the specified lottery.
- *buyTicketTx*: public function. Allows users to purchase tickets for a specified lottery, with a hashed random number for security.
- *withdrawTicketRefund*: public function. Processes refunds for tickets in cases where a lottery is canceled or a refund is applicable.

c) Uncategorized :
- *getNumPurchaseTxs*: public function. Returns the total number of ticket purchase transactions for the specified lottery.
- *getPaymentToken*: public function. Retrieves the payment token associated with the specified lottery.
- *setPaymentToken*: public function. Allows the owner to set or change the payment token for future lotteries.
- *revealRndNumberTx*: public function. Allows revealing of the random number for verifying ticket results in a lottery.
- *isContract*: internal function. Utility function to check if an address corresponds to a contract.

TABLE I
FUNCTION CHECKLIST

Required Function	Implemented?	Corresponding Function
createLottery	✓Yes	createLottery
buyTicketTx	✓Yes	buyTicketTx
revealRndNumberTx	✓Yes	revealRndNumberTx
getNumPurchaseTxs	✓Yes	getNumPurchaseTxs
getIthPurchasedTicketTx	✓Yes	getIthPurchasedTicket
checkIfMyTicketWon	✓Yes	checkIfMyTicketWon
checkIfAddrTicketWon	✓Yes	checkIfAddressTicketWon
getIthWinningTicket	✓Yes	getIthWinningTicket
withdrawTicketRefund	✓Yes	withdrawTicketRefund
getCurrentLotteryNo	✓Yes	getCurrentLotteryNo
withdrawTicketProceeds	✓Yes	withdrawTicketProceeds
setPaymentToken	✓Yes	setPaymentToken
getPaymentToken	✓Yes	getPaymentToken
getLotteryInfo	✓Yes	getLotteryInfo
getLotteryURL	✓Yes	getLotteryURL
getLotterySales	✓Yes	getLotterySales

A. Workflow

1) Lottery Creation:
- The company initializes a lottery with specified parameters using createLottery.
- Users can view the lottery details through query functions.

2) Ticket Purchase:
- Users buy tickets during the purchase phase by calling buyTicketTx.

3) Random Number Submission:
- Users submit hashes of their random numbers during ticket purchase.

4) Random Number Reveal:
- During the reveal phase, users call revealRndNumberTx to reveal their random numbers.

5) Lottery Finalization:
- The lottery is finalized after the reveal phase ends. Winners are selected, or refunds are issued if participation criteria are unmet.

B. Data Flow

The data flow in the system ensures that all user inputs, such as ticket purchases and random number submissions, are validated and securely recorded on the blockchain. Each transaction is checked for correctness to maintain the integrity of the system. Additionally, the contract automatically manages state transitions, such as moving from the purchase phase to the reveal phase, based on predefined time intervals and user participation levels. This automation ensures that the lottery progresses without requiring manual intervention. To determine winners fairly, the system relies on randomness derived directly from user-supplied random numbers, eliminating the need for centralized randomness generators.

C. Error Handling

Error handling in the contract is robust and ensures the system operates reliably under various scenarios. Reverts are extensively used to prevent invalid operations, such as attempting to buy tickets for lotteries that do not exist or revealing random numbers outside the designated reveal phase. Furthermore, the contract safeguards against improper actions like withdrawing proceeds from canceled lotteries. These measures help maintain the security and functionality of the system, protecting both users and the integrity of the lotteries.

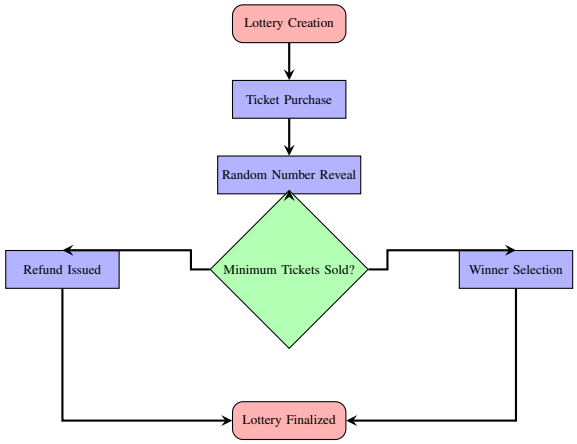
III. IMPLEMENTATION DETAILS

The implementation of the **CompanyLotteries** smart contract is divided into several key functional areas, reflecting the requirements of the homework. Below, we outline the implementation specifics, highlighting design decisions, challenges, and their resolutions.

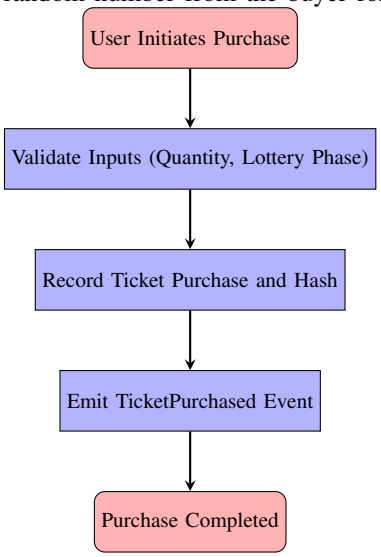
A. Core Functionality

The contract leverages Solidity to implement a fully automated multi-lottery system. Below are the details of the main features:

1) Lottery Creation
   - Implemented using the createLottery function, which accepts parameters such as:
     - unixbeg: Start time in Unix format.
     - nooftickets: Total number of tickets.
     - noofwinners: Number of winners.
     - minpercentage: Minimum percentage of tickets sold for the lottery to proceed.
     - ticketprice: Price of each ticket (in an ERC20 token).
   - Ensures data integrity with input validation.



2) Ticket Purchase
   - The buyTicketTx function allows users to purchase up to 30 tickets per transaction.
   - Records ticket ownership and accepts a hashed random number from the buyer for future reveal.



3) Random Number Reveal
   - The revealRndNumberTx function facilitates the reveal phase.
   - Buyers must reveal their random numbers for ticket validation, ensuring fairness.

4) Winner Selection:
   - Winner selection is a crucial part of the lottery lifecycle, ensuring transparency and fairness. The selection process is driven by randomness provided by users during the purchase phase. Each user submits a hashed random number when buying tickets, committing to their input without revealing it upfront. This commitment prevents tampering, as the user cannot alter their random number once it has been recorded on-chain.
   - During the reveal phase, users must reveal the original random number that corresponds to the submitted hash. The contract verifies the revealed number by comparing its hash to the stored value:

```
require(keccak256(abi.encodePacked(
    revealedNumber)) == storedHash, "
    Invalid␣random␣number");
```

Listing 1. Validation of Revealed Random Numbers

   - The reliance on user-supplied randomness ensures that the winner selection process is decentralized and resistant to manipulation. The randomness provided by all participants collectively determines the outcome, eliminating the need for centralized random number generators.
   - Furthermore, the reveal phase plays a key role in maintaining fairness. Users who fail to reveal their random numbers within the designated timeframe forfeit their participation, ensuring that only valid and committed entries are considered during winner selection. By structuring the process into distinct phases, the contract guarantees a tamper-proof and transparent method for determining winners.

B. Data Structures

The contract utilizes robust data structures to manage lotteries efficiently. Key mappings are used to store critical information. The *lotteries* mapping holds the details of all created lotteries, while the *tickets* mapping tracks ticket ownership and the associated random numbers submitted by users. Additionally, the *winners* mapping maintains a list of winning tickets for each lottery. Structs further enhance data organization by capturing complex information. The *Lottery* struct defines parameters such as the phases and sales status of each lottery, while the *Ticket* struct links ticket ownership to buyers and their random numbers, ensuring traceability and fairness.

C. Modifiers

To ensure secure and correct usage of the contract, modifiers play a vital role. The *onlyOwner* modifier restricts sensitive operations, such as *createLottery* and *setPaymentToken*, to the contract owner, preventing unauthorized access. The *validLottery* modifier validates the existence of a lottery before executing dependent functions, ensuring that operations are applied only to valid lottery instances.

D. Error Handling

The contract employs *require* statements extensively to enforce proper error handling. These statements validate inputs, such as ensuring ticket quantities fall within acceptable limits or that operations occur during the correct phase. Unauthorized access to critical functions is also prevented through these checks. For example, the following Solidity statement ensures valid ticket quantities:

```
require(quantity > 0 && quantity <= 30, "Invalid ticket quantity");
```

Listing 2. Example of Error Handling in Solidity

This approach safeguards the contract against invalid operations, enhancing its reliability and user experience.

E. Challenges and Resolutions

The development of the contract involved addressing two key challenges. The first was ensuring secure randomness without using on-chain generators, which are prone to exploitation. This was resolved by relying on user-supplied random numbers during ticket purchases and their subsequent reveal. The second challenge was optimizing gas costs, particularly for ticket purchases and winner selection. By structuring data efficiently and minimizing computational loops, the contract achieved significant gas savings, making it more cost-effective for users.

F. Testing

The implementation was tested using the Hardhat framework. The tests covered both valid and invalid cases for all functions, ensuring robustness under various conditions. Edge cases, such as scenarios with low ticket sales or invalid random number reveals, were also included to verify the contract's reliability. This thorough testing process ensures that the contract operates as intended in all expected scenarios.

G. Known Gaps

The test cases with 100/200/300/300+ user addresses will be performed before moving onto Homework 2, the deployment.

IV. Testing Methodology

A. Testing Framework

The contract was tested with:

- Hardhat: A development environment for Solidity.
- JavaScript: For writing test cases using the Mocha and Chai libraries.

B. Test Coverage

The tests comprehensively covered the following aspects:

- Core Functionalities:
  - Lottery Creation:
    - Validation of input parameters.
    - Emission of the LotteryCreated event.
  - Ticket Purchase:
    - Successful ticket purchases within defined limits.
    - Proper ticket ownership recording.
    - Emission of the TicketPurchased event.
  - Random Number Reveal:
    - Validation of submitted random numbers.
    - Ensuring only ticket owners can reveal numbers.
    - Emission of the RandomNumberRevealed event.
  - Winner Selection:
    - Ensuring winners are determined fairly based on revealed numbers.
    - Correct handling of insufficient ticket sales.
- Edge Cases:
  - Invalid input handling, such as zero ticket purchases or nonexistent lotteries.
  - Enforcement of phase-based restrictions (e.g., revealing numbers only in the reveal phase).
- Error Handling:
  - Validation of proper reverts for unauthorized or invalid actions.
  - Scenarios like incorrect ticket quantities or invalid random numbers.
- Query Functions:
  - Accurate retrieval of lottery information, ticket ownership, and sales data.

C. Test Cases

Tests were grouped by function and category, ensuring both successful executions and expected failures were validated. Example Test Scenarios:

1) createLottery:
   a) Pass: Successfully creates a lottery with valid parameters.
   b) Fail: Reverts if the ticket price or number of winners is zero.
2) buyTicketTx:
   a) Pass: Allows purchases within the limit of 30 tickets per transaction.
   b) Fail: Reverts for invalid ticket quantities or nonexistent lotteries.
3) revealRndNumberTx:
   a) Pass: Successfully reveals random numbers for valid tickets.
   b) Fail: Reverts for invalid or non-owned tickets.

D. Testing Tools

- Local Blockchain: The Hardhat network was used to simulate the Ethereum environment.
- ERC20 Mock Contract: A mock implementation of OpenZeppelin ERC20 token was deployed to test payment functionality.

E. Testing Results and Observations

```
Compiled 8 Solidity files successfully (evm target: paris).


  CompanyLotteries - buyTicketTx
    Successful buyTicketTx
      ✓ should buy 1 ticket for the lottery for the given id and emit TicketPurchased event
      ✓ should allow buying tickets within valid limit (1 - 30) and emit TicketPurchased event
    Failed buyTicketTx
      ✓ should revert if quantity is 0
      ✓ should revert if quantity is greater than 30
      ✓ should revert if random hash is empty
      ✓ should revert if the lottery does not exist
      ✓ should revert if the lottery is not in the purchase phase
      ✓ should revert if the buyer does not have enough balance
      ✓ should revert if the requested quantity exceeds the remaining tickets

  CompanyLotteries - checkIfAddressTicketWon
    Successful checkIfAddressTicketWon
      ✓ should return a boolean value if the ticket won or not
    Failed checkIfAddressTicketWon
      ✓ should revert if the lottery does not exist
      ✓ should revert if the reveal phase has not ended yet
      ✓ should revert if the ticket does not exist or is unowned
      ✓ should revert if the ticket does not belong to the address

  CompanyLotteries - checkIfMyTicketWon
    Successful checkIfMyTicketWon
      ✓ should return a boolean value if the buyer ticket won or not
      ✓ should return a boolean value if the otherUser ticket won or not
    Failed checkIfMyTicketWon
      ✓ should revert if the lottery does not exist
      ✓ should revert if the reveal phase has not ended
      ✓ should revert if the ticket does not exist or unowned
      ✓ should revert if the ticket does not belong to the caller

  CompanyLotteries - createLottery
    Successful createLottery
      ✓ should create a new lottery with LotteryCreated event and return the lottery number (1 here)
    Failed createLottery
      ✓ should revert if the ticket price is 0
      ✓ should revert if the number of tickets is 0
      ✓ should revert if the number of winners is 0
      ✓ should revert if the number of winners is greater than the number of tickets
      ✓ should revert if the minimum percentage is 0
      ✓ should revert if the minimum percentage is greater than 100
      ✓ should revert if the lottery end time is not in the future

  CompanyLotteries - finalizeLottery
    Successful finalizeLottery
      ✓ should finalize the lottery and emit LotteryFinalized event if the lottery has met the minimum ticket sales
      ✓ should cancel the lottery and emit LotteryCancelled event if the lottery has not met the minimum ticket sales
    Failed finalizeLottery
      ✓ should revert if the lottery does not exist
      ✓ should revert if the reveal phase has not ended yet
      ✓ should revert if the lottery has already been canceled or finalized

  CompanyLotteries - getCurrentLotteryNo
    Successful getCurrentLotteryNo
      ✓ should return 0 if no lottery is created
      ✓ should return 1 if a lottery is created
      ✓ should return 2 if two lotteries are created

  CompanyLotteries - getIthPurchasedTicket
    Successful getIthPurchasedTicket
      ✓ should return 1 for starting ticket no and 2 for quantity if the buyer has bought 2 tickets at once
      ✓ should return 5 for starting ticket no and 3 for quantity if the otherUser has bought 3 tickets at once
    Failed getIthPurchasedTicket
      ✓ should revert if the lottery does not exist
      ✓ should revert if the ith ticket is greater than the total number of tickets bought

  CompanyLotteries - getIthWinningTicket
    Successful getIthWinningTicket
      ✓ should return the winning ticket no at the ith position
    Failed getIthWinningTicket
      ✓ should revert if the lottery does not exist
      ✓ should revert if the reveal phase has not ended
      ✓ should revert if the lottery has not been finalized
      ✓ should revert if the ith ticket is greater than the total number of winners

  CompanyLotteries - getLotteryInfo
    Successful getLotteryInfo
      ✓ should return the lottery info
    Failed getLotteryInfo
      ✓ should revert if the lottery does not exist

  CompanyLotteries - getLotterySales
    Successful getLotterySales
      ✓ should return 0 if no tickets are sold
      ✓ should return 1 if 1 ticket is sold
      ✓ should return 2 if 2 tickets are sold
    Unsuccessful getLotterySales
      ✓ should revert if lottery does not exist

  CompanyLotteries - getLotteryURL
    Successful getLotteryURL
      ✓ should return the lottery URL
    Failed getLotteryURL
      ✓ should revert if the lottery does not exist

  CompanyLotteries - getNumPurchaseTxs
    Successful getNumPurchaseTxs
      ✓ should return 0 purchase transactions for a new lottery
      ✓ should return 1 purchase transaction after a ticket purchase
      ✓ should return 2 purchase txs with one for 2 tickets and another for 1 ticket
    Failed getNumPurchaseTxs
      ✓ should revert if the lottery does not exist

  CompanyLotteries - getPaymentToken
    Successful getPaymentToken
      ✓ should return ZeroAddress if no payment token is set
      ✓ should return the payment token address if set
    Failed getPaymentToken
      ✓ should revert if the lottery does not exist

  CompanyLotteries - getPaymentToken
    Successful getPaymentToken
      ✓ should return ZeroAddress if no payment token is set
      ✓ should return the payment token address if set
    Failed getPaymentToken
      ✓ should revert if the lottery does not exist

  CompanyLotteries - revealRndNumberTx
    Successful revealRndNumberTx
      ✓ should not revert if the randon number is revealed by the buyer
    Failed revealRndNumberTx
      ✓ should revert if the random number is not revealed by the buyer
      ✓ should revert if the random number is incorrect
      ✓ should revert if the lottery does not exist
      ✓ should revert if the reveal phase has not started
      ✓ should revert if the reveal phase has ended
      ✓ should not reveal the random number if the quantity is zero
      ✓ should not reveal the random number if the quantity is greater than 30

  CompanyLotteries - setPaymentToken
    Successful setPaymentToken
      ✓ should set the payment token for the lottery and emit NewPaymentTokenSet event
    Failed setPaymentToken
      ✓ should revert if the caller is not the owner
      ✓ should revert if the token address is ZeroAddress
      ✓ should revert if the token address is not a contract
      ✓ should revert if the new token is the same as the current token

  CompanyLotteries - withdrawTicketProceeds
    Successful withdrawTicketProceeds
      ✓ should withdraw the ticket proceeds and emit ProceedsWithdrawn event
    Failed withdrawTicketProceeds
      ✓ should revert if the lottery does not exist
      ✓ should revert if the lottery is not finalized or canceled
      ✓ should revert if the caller is not the owner
      ✓ should revert if the ticket proceeds have already been withdrawn
      ✓ should revert if the proceeds transfer fails

  CompanyLotteries - withdrawTicketRefund
    Successful withdrawTicketRefund
      ✓ should withdraw the ticket refund and emit TicketRefundWithdrawn event
    Failed withdrawTicketRefund
      ✓ should revert if the lottery does not exist
      ✓ should revert if the lottery is not finalized or canceled
      ✓ should revert if the caller is not the ticket owner
      ✓ should revert if the refund has already been withdrawn
      ✓ should revert if the refund transfer fails


  85 passing (3s)
```

Fig. 1. Test results as seen in the terminal. Please zoom in for readability.

A total of 85 test cases were executed during the testing phase, with all tests passing successfully. These results demonstrated full compliance with the functional requirements specified for the contract. The system effectively handled edge cases and errors, ensuring robustness and reliability in various scenarios. The testing process confirmed that the contract's logic was sound and capable of managing different operational conditions without failures.

The tests were conducted on a local blockchain using the Hardhat network, which simulated the Ethereum environment. To validate payment-related functionality, a mock implementation of an ERC20 token contract was deployed. This allowed the team to test ticket purchases, refunds, and proceeds withdrawal without requiring interactions with real tokens on the mainnet.

The observations from the testing process highlighted the system's performance and reliability. Gas costs were found to be reasonable, attributed to the optimized logic used for ticket handling and winner selection. Additionally, the contract demonstrated exceptional reliability, handling all edge cases and invalid operations effectively. These results underscore the robustness and efficiency of the implementation.

V. Gas Analysis

The gas usage data from the **CompanyLotteries** contract provides valuable insights into the cost efficiency and complexity of its functions. The functions exhibit a range of gas consumption values, reflecting their respective purposes and operational intricacies.

Functions with frequent state updates (e.g., *buyTicketTx* and *finalizeLottery*) tend to be more gas-intensive, reflecting the cost of writing data to the blockchain. Simpler, administrative functions (e.g., *setPaymentToken*) are consistently gas-efficient due to their limited scope and minimal state changes. Functions with highly variable gas consumption (e.g., *buyTicketTx*, *finalizeLottery*) highlight areas where optimizations could reduce peak usage, especially for large-scale operations.

Solidity and Network Configuration					
Solidity: 0.8.27	Optim: false	Runs: 200	viaIR: false	Block: 30,000,000 gas	
Methods					
Contracts / Methods	Min	Max	Avg	# calls	usd (avg)
CompanyLotteries					
buyTicketTx	190,728	2,217,121	519,590	83	-
createLottery	261,474	278,574	278,355	156	-
finalizeLottery	36,698	210,507	185,823	22	-
revealRndNumberTx	34,257	54,325	53,993	61	-
setPaymentToken	53,123	53,135	53,135	70	-
withdrawTicketProceeds	-	-	88,914	3	-
withdrawTicketRefund	-	-	76,694	3	-
MockERC20					
approve	46,911	46,923	46,920	107	-
mint	51,893	69,005	62,448	107	-
setFailTransfer	-	-	43,816	2	-
transfer			49,581	1	-
Deployments				% of limit	
CompanyLotteries	-	-	4,048,213	13.5 %	-
MockERC20	-	-	1,321,940	4.4 %	-
Key					
○ Execution gas for this method does not include intrinsic gas overhead					
△ Cost was non-zero but below the precision setting for the currency display (see options)					
Toolchain: hardhat					

Fig. 2. Gas Usage Analysis for **CompanyLotteries** Smart Contract. Please zoom in for readability.

VI. Directory Structure and Component Overview

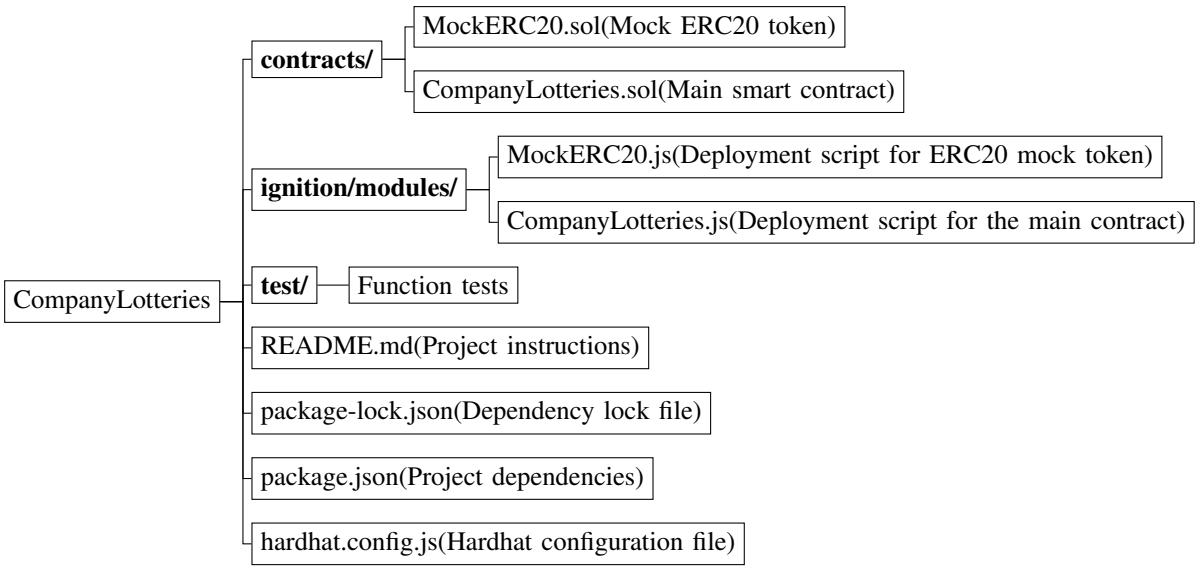


TABLE II
Task Achievement Table

	Yes	Partially	No
I have prepared documentation with at least 6 pages.	**X**		
I have provided average gas usages for the interface functions.	**X**		
I have provided comments in my code.	**X**		
I have developed test scripts, performed tests, and submitted test scripts as well-documented test results.	**X**		
I have developed smart contract Solidity code and submitted it.	**X**		
Function createLottery is implemented and works.	**X**		
Function buyTicketTx is implemented and works.	**X**		
Function revealRndNumberTx is implemented and works.	**X**		
Function getNumPurchaseTxs is implemented and works.	**X**		
Function getIthPurchasedTicketTx is implemented and works.	**X**		
Function checkIfMyTicketWon is implemented and works.	**X**		
Function checkIfAddrTicketWon is implemented and works.	**X**		
Function getIthWinningTicket is implemented and works.	**X**		
Function withdrawTicketRefund is implemented and works.	**X**		
Function getCurrentLotteryNo is implemented and works.	**X**		
Function withdrawTicketProceeds is implemented and works.	**X**		
Function setPaymentToken is implemented and works.	**X**		
Function getPaymentToken is implemented and works.	**X**		
Function getLotteryInfo is implemented and works.	**X**		
Function getLotteryURL is implemented and works.	**X**		
Function getLotterySales is implemented and works.	**X**		
I have tested my smart contract with 100 user addresses and documented the results of these tests.			**X**
I have tested my smart contract with 200 user addresses and documented the results of these tests.			**X**
I have tested my smart contract with 300 user addresses and documented the results of these tests.			**X**
I have tested my smart contract with more than 300 user addresses and documented the results of these tests.			**X**