ECE-519C PROJECT REPORT- BLACKJACK USING SOLIDITY | University of Victoria

The project involves creating a game of blackjack using solidity on the Remix IDE. To explain the motivation and reasons behind the project, the following questions were answered to justify the use of blockchain as a platform for online legal gambling. (*Note: We highly suggest visiting the following link to learn the rules of blackjack before assessing the report and code: https://bicyclecards.com/how-to-play/blackjack/*)

Why blackjack?

Among the various card gambling games, blackjack is the most widely played game in the entire world. It is also one of the few games that involve betting against the dealer. The rules of the game are also simple. Whoever can reach closer to, or a total of 21, wins. The simple, yet addicting nature of the game, makes it attractive for players to employ unique strategies while placing bets and make them as profitable as possible.

What are the problems/concerns with traditional gambling websites?

Online gambling has a huge presence over the internet. Many online casinos rake in huge amounts of money from thousands of players all over the world. While some contend that it brings in international trade due to cross-border activity, others believe it involves a great amount of risk and fraud. That's not to say that there aren't any legitimate or legal gambling sites. Websites like Jackpot City and Casino.com are examples of trusted casino sites that have been active for a long time. An online gambling site should have the following properties to be deemed a trusted platform:

- There should not be ads or pop-up messages.
- It must be free of all scams and hacking.
- The user details must be kept private and confidential.
- The platform should be transparent with the users regarding their terms of usage and payment.

However, some gambling websites do not promote fair terms of usage and payment. The use of the card decks goes out of the line of statistical norms. The centralized nature of these online casinos gives them the advantage to outplay the player a greater number of times than expected. Therefore, it is necessary to decentralize the gambling platform to ensure a fair and transparent means of playing.

Why use Blockchain as a platform for online gambling?

The dealing of cards in any card-based gambling game is supposed to be random. With a centralized platform, the degree of randomness is not transparent, and players can feel cheated out of their money. The odds are always in favor of the casino.

Introducing Blockchain as an online gambling platform will ensure the following advantages:

- A fair game: A smart contract works on the principle of "if… then" which is executed automatically based on the condition. The terms and conditions of the contract are available to be viewed by anyone and are completely transparent. The randomness generators in the contract are also just and fair and can be viewed by anyone. This ensures that the player is given a fair opportunity to bet and win money.
- Efficiency of transactions: No third parties are involved in payments. The transactions are processed directly from the sender to the receiver. Therefore, the cost of transaction fees can be considered very less or negligible. The processing of payments is also much faster. Players can deposit and withdraw funds anytime without waiting for several hours or days.
- Anonymous access to the platform: Player information is kept confidential by the smart contract.
- Advantage of a decentralized system: A decentralized system ensures that the data is stored in the nodes at the same time making it difficult to be tampered with.

The fair play offered by blockchain can be further explained by introducing the concept of **provably fair gambling**. This concept allows players to verify that the outcome of any gambling game is fair in real-time. The concept makes use of the following three components:

- Hashed seed sent by the service operator.
- Hashed seed sent by player.
- Random number generator.

Before the bet is placed, the dealer will send a hash of the original shuffled deck to the player. The player will then add his/her own hashed seed along with a nonce from a random number generator. Once the hand is played out and the outcome is shown, both the seeds are sent to the player unencrypted. The player can verify by hashing the seeds and comparing them to the original two hashes. If both are the same, it is verified that the game was played fair, else a foul play can be suspected. The correlation between the server seed and the client/player seed depends on the rules of the game that is being played.

The concept of provably fair gambling was first implemented in Blackjack by the bitcoin casino BitZino in 2012. Since the implementation, most of the bitcoin casinos are also implementing provably fair games to all players. All the games can be independently verified by the players.

How to ensure secure random numbers in a smart contract? *(some real-world scenarios/implementations.)*
Even though blockchain offers transparency in gambling, it is very hard to generate a perfect random number that is secure. Miners can influence or call the random function headers since the contracts are publicly visible. There are a few solutions that offer different methods to verify a random number but none of them are without drawbacks.

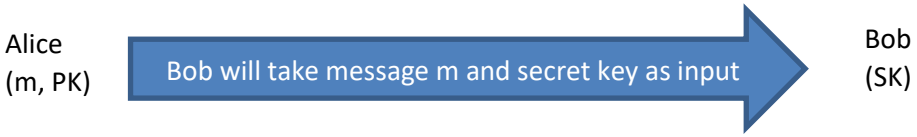
One solution is to fetch random data from outside into the smart contract by using a service like Oraclize/Provable. Oraclize delivers data from sources like random.org into the blockchain. However, this will require trusting the external source and the service, at its core, is a centralized data service. This contradicts the very nature of blockchain.

Another solution is to use the concept of **Verifiable Random Function (VRF).** It is a pseudo-random function that takes an input and produces an output along with a proof of correctness. They can also be described as public-key versions of hashing algorithms like SHA256 and Keccak (which is the hashing algorithm used in the RNG of our code).

In this function, only the person with the secret key can compute the output but anyone with a public key can verify the correctness. The algorithm can be represented in the following three steps:

- **Key Generation (PK, SK)**: On a random input, a secret key and public key, also known as the verification key is generated by the key generation algorithm.
- **Evaluation (m, SK) => (c, p)**: The evaluation algorithm will take the secret key and the message ‘m’ to generate a pseudorandom output ‘c’ and a proof of correctness ‘p’.
- **Verification**: The verification algorithm will take the public key PK, the message m, the output c, and the proof p as input. The algorithm will verify if c is the output of the evaluation algorithm and if so, will return true.

A clearer explanation can be given by taking the example of Alice and Bob. Alice has the input m and the public key PK. Bob has the secret key SK.



- Bob will use his secret key to evaluate the pseudorandom output ‘c’ of the message m sent by Alice. Alice does not know Bob’s secret key and Bob cannot change the input since it is controlled by Alice.
- Since Alice does not know the secret key, she cannot know the random number unless Bob calculates it. Also, Alice cannot change the secret key as it will render the public key invalid during verification. Hence, neither Alice nor Bob can influence the random output.

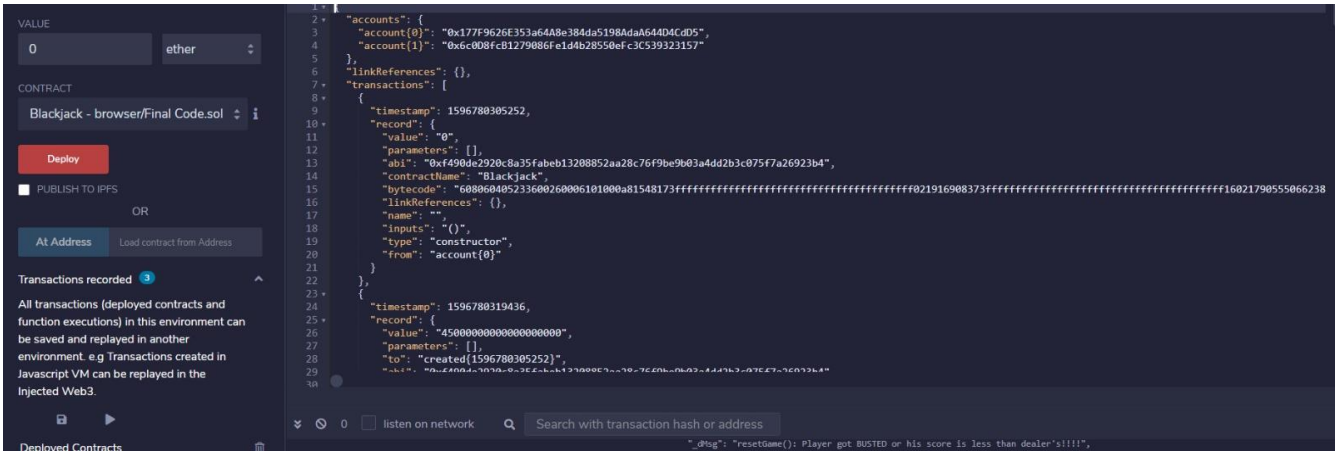
- Bob will generate the random number as well as the proof of generation.
- Alice can then verify the generated output using the public key and the proof using the verification algorithm.

In this way, a secure random number can be generated without consuming huge computational power. Another main advantage of VRFs is that they produce a single unique output, unlike digital signatures which can have more than one signature for the same input.

How does our code introduce fair randomness in Blackjack?

As discussed earlier, traditional gambling websites cannot be trusted for playing fair with the player. Using blockchain as a legal gambling platform will ensure fair play and introduce transparency. The implementation of Blackjack in our code is not only simple and easy to interact with, but also provide a sense of fairness to the player.

Every transaction that occurs during deployment of the contract is recorded and thus the information cannot be tampered with.



The above JSON file shows records of the player and dealer addresses as well as the actions taken by the player in the game, the timestamps during which the actions took place, etc.

The code has a **drawFromDeck()** function that is used to draw a card from a shuffled deck of cards. The following image shows the implementation of the function.

```
189     function _drawFromDeck(uint256 _gameId)
190         private
191         returns(uint8 _newCard){
192             bool _gotCard = false;
193             while(!_gotCard){
194                 uint256 _randVal = (_genRandom(_gameId) % 52) + 1;
195                 if(gameIdMap[_gameId].sourceDeck[_randVal % 13] > 0){
196                     _newCard = uint8(_randVal % 13 + 1);
197                     if(_newCard > 10){
198                         _newCard = 10;
199                     }
200                     gameIdMap[_gameId].sourceDeck[_randVal % 13]--;
201                     _gotCard = true;
202                 }
203             }
204         }
205
```

Since Blackjack involves using a shuffled deck of cards, this shuffle is realized through a random number generator in the code as shown below.

```
    function _genRandom(uint256 _gameId)
        internal
        returns(uint256 _randVal){
            bytes32 _hashval = keccak256(abi.encodePacked(block.timestamp, gameIdMap[_gameId].numOfCardsP, gameIdMap[_gameId].numOfCardsD, ++nonce));
            _randVal = uint256(_hashval);
        }
```

Using the Keccak algorithm, the hash of the current time, the number of player and dealer cards, and the nonce are generated. The 32-byte hash is then stored as a 256-bit integer value. This function is then invoked in the draw function to ensure that the card is drawn from a shuffled deck.

The main advantage of this random number generator is that value of the hash will keep changing for every draw as the timestamp and nonce values will be different for each case. This ensures that the randomness is fair and untampered with.

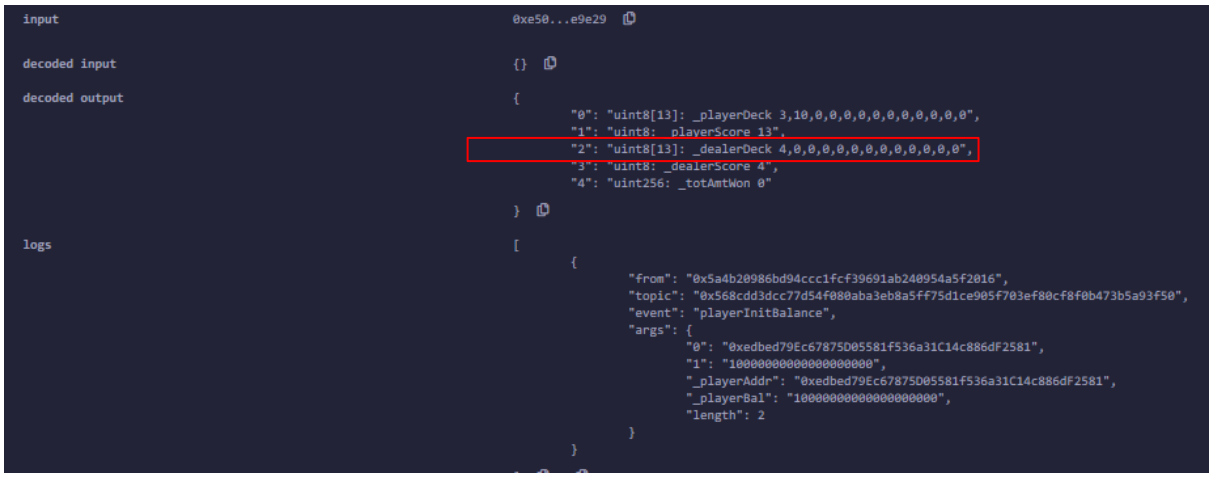
The code is also efficient in the sense that anytime the player wins or loses, the bet amount is instantly transferred from sender to receiver. *This prevents re-entrancy attacks from taking place where the player can keep calling the transfer function again and again before the amount is sent.*

The following **_leaveGame()** function is implemented in which once the player finishes playing the round, the funds are transferred to the respective address based on the outcome.

```
    function _leaveGame(uint _gameId)
        internal
        isCurrentlyPlaying
        ifPlayerWon
        returns(uint8[13] memory _playerDeck, uint8 _playerScore, uint8[13] memory _dealerDeck, uint8 _dealerScore, uint256 _totAmtWon){
            _totAmtWon = 0;
            _checkForAceP(_gameId);
            _checkDealerOutcome(_gameId);                                   // Dealer draws till dealer_score <= 17

            if ((gameIdMap[_gameId].playerScore > 21)){ // PLAYER: Busted or lost!!
                _totAmtWon = 0;
                emit _gameDecision("_leaveGame(): dealer got the balance!!!");
                dealerAddr.transfer(gameIdMap[_gameId].betAmt);
            }
            else if((gameIdMap[_gameId].dealerScore > gameIdMap[_gameId].playerScore) && gameIdMap[_gameId].dealerScore <= 21){
                _totAmtWon = 0;
                emit _gameDecision("_leaveGame(): dealer got the balance!!! else if()");
                dealerAddr.transfer(gameIdMap[_gameId].betAmt);
            }
            else{
                //_checkDealerOutcome(_gameId);                                 // Dealer draws till dealer_score <= 17

                if(gameIdMap[_gameId].insuranceOpted == true ){
                    if(gameIdMap[_gameId].dealerDeck[1] == 10){ //check faceDownCard whether it is 10 or a face card
                        _totAmtWon += (2*gameIdMap[_gameId].insuranceBet);
                    }
                    else if(gameIdMap[_gameId].dealerScore == 21 && gameIdMap[_gameId].playerScore == 21){
                        _totAmtWon += gameIdMap[_gameId].insuranceBet;
                    }
                }

                if(gameIdMap[_gameId].dealerScore > 21 || (gameIdMap[_gameId].playerScore > gameIdMap[_gameId].dealerScore)){ //dealer loses
                    gameIdMap[_gameId].rcvdWinAmt = true;
                    _totAmtWon += 2*gameIdMap[_gameId].betAmt;                  // ADD LOG - DEALER BUSTED ? OR PLAYER WON ?
                    gameIdMap[_gameId].playerAddr.transfer(_totAmtWon);             // Player wins 2*bet_amount
                }
                else if(gameIdMap[_gameId].playerScore == gameIdMap[_gameId].dealerScore){
                    gameIdMap[_gameId].rcvdWinAmt = true;                                       // TIE
                    _totAmtWon += gameIdMap[_gameId].betAmt;                            // Player gets back his bet_amount
                    gameIdMap[_gameId].playerAddr.transfer(_totAmtWon);
                }
            }

            (_playerDeck, _playerScore, _dealerDeck, _dealerScore) = resetGame(_gameId);

            return(_playerDeck, _playerScore, _dealerDeck, _dealerScore, _totAmtWon);
        }
```

The **_leaveGame()** function is always invoked when the player decides to **stand**. The function then checks the dealer's outcome which corresponds to the dealer score and the funds are transferred accordingly. Since the function is linked to the stand action, no player can call the leaveGame function seperately without playing.

The code also ensures that the facedown dealer card is not revealed until and after the player is done playing the hand. The player cannot know the face down card beforehand and this further adds to the fair play value of the game. The following screenshot shows that the facedown card is not revealed until the and after the player plays the hand.



All these factors ensure that our implementation of Blackjack in Solidity ensures transparency, fair play, and genuine randomness of the game. We genuinely believe that blockchain as a gambling platform will revolutionize online gambling in the coming years. As the online gambling market continues to grow at a rapid rate, the benefits offered by blockchain will only add to the sharp rising growth of the industry.

Steps to run the code:

- Copy the code on the Remix IDE.
- Ensure that the compiler version matches the contract.
- Compile the code.
- Select an account, set the value field to 0, increase the gas limit to 5000000 or more, and deploy.
- Select another account (i.e.; the player account), enter the desired bet amount (minimum 0.01 ether) in the Value field, and click on 'startGameNBet' in the deployed contract to start the game. The player account is now registered, and the bet will be placed. (Check output console for more information)
- Go to the logs section to view the scores of the player and dealer and use the options in the deployed contract to play the hand.
    - Only player/s can call **stand**, **hit, doubleDown, and Insurance** functions.
    - Only dealer (contract owner) can call **abortGame, changeValues, ifPlayerTimeout and reqMoreTime** functions.
    - Remaining functions are common and callable irrespective of player/dealer.
- You can check the balance of the accounts for each action taken (i.e. whenever the bet is placed or stand action is performed).

REFERENCES

- https://www.forbes.com/sites/geraldfenech/2019/01/30/blockchain-in-gambling-and-betting-are-there-real-advantages/#4f7fd3cd7c63
- https://medium.com/ginar-io/a-review-of-random-number-generator-rng-on-blockchain-fe342d76261b
- https://medium.com/algorand/algorand-releases-first-open-source-code-of-verifiable-random-function-93c2960abd61