UNIVERSITY OF EXETER

Computer Science Department

B.Sc. Computer Science
Computer Science Department

Gambling on the Ethereum Blockchain

Candidate

Jeroen Mijer

Student ID 700046371

Supervisor

Dr. Diego Marmsoler

University of Exeter

Academic Year
2022/2023

Abstract

Online casinos, if willing, could easily act maliciously and take advantage of its clients due to much less strict regulation and controls that physical casino's face. Online casinos can mislead users about their odds of winning, as their code operates in a black box and users must take their word for it. As online gambling is getting increasingly popular more and more people are getting exposed to this risk. This paper aims to demonstrate how building a gambling application using blockchain technology can increase the transparency to users, including the possibility to verify the fairness of the casino and their bets. This paper demonstrates the minimum viable product that was built how this could work including an evaluation of the model and potential further improvements and works.

	Yes	No
I certify that all material in this dissertation which is not my own work has been identified.	☑	☐
I give the permission to the Department of Computer Science of the University of Exeter to include this manuscript in the institutional repository, exclusively for academic purposes.	☑	☐

Contents

1 Introduction

Online gambling in the UK has recently seen an increase in popularity in recent years with the online gambling participation rate increasing to 27% from September 2021 to September 2022[10][1]. Traditionally it was much easier to regulate the gambling industry as it was tied to a physical location. However online casinos can be set up at very little cost and when shut down simply reopen under a different domain name. Online casinos can also register in countries where gambling laws and regulations are more relaxed and advise users to access them via a VPN to get around regulations [8].

In order to ensure that an online casino is reliable and safe to use as a customer a gambling regulator is often required to enforce legislation. This is because a casino may prevent users from withdrawing any deposits, manipulating payouts or playing with pirated versions of games, that allow the casino to win at a higher percentage than is advertised to the user. This can lead to users having their funds unfairly lost or stolen. In order to prevent this a user may attempt to play at reputable casinos however it is often difficult to fully verify whether a casino is being honest. There have been multiple cases where various online casinos have been using pirated and altered games. These games then offer lower odds than promised or work with predetermined numbers that ensure that the casino win more than the promised odds. Casinos can offer features to verify that a game took place in a fair manner, but these often only work retroactively (after the money has been spent). The verification features are also often technically difficult to use and are therefore rarely utilized. In some cases when they were used they exposed the casino as fraudulent but this was months into operating, meaning many users were already taken advantage of. [11][9]

The casino may also have problems where the users try to cheat them. An example of this is chargeback fraud, where the user places bets and then claims they never placed them. Often this complaint is made to their bank or credit card company who will refund them their deposit while the fraudster keeps their winnings from any gambling. The casino may be unable to definitively prove whether the bets were placed or not, relying only on their logging system which could be faulty. There is also a risk for casinos who have to keep this data reliably stored and if the data is lost or corrupted casinos may have to rely on the user's word on their account balance or winnings. Having this data available on the blockchain to be viewed long after the casino stops operating provides a reliable way to recover or analyse the data. The data would also be available to users and they would not have to rely on the casino in any way to access

their data.

In this paper a method to use a decentralised blockchain in order to run the gambling games is proposed. An online casino can host their casino services as normal however the actual gambling occurs on a blockchain using smart contracts. A smart contract is code that is executed and stored on the blockchain. It cannot be altered and when executed it operates in a deterministic manner. This means that if a casino uses a smart contract to run a game then they cannot alter the game or any details about it. The casino can also provide a method for the user to verify whether the game was altered or interfered with in any way. Therefore the user can be sure that their bet was executed fairly and that they will receive the payout.

The fairness of the game can be verified because each transaction is available to audit on the blockchain. User's and casino's alike can view a history of bets that have been taken place and can ensure that the correct smart contracts were used. The history of transactions is publicly available and cannot be manipulated either by users or casinos. This history of transactions is maintained by a decentralised blockchain which is not subject to a single point of failure. In this paper it will be the Ethereum blockchain chosen for its popularity and reliability. The Ethereum blockchain will be used to execute, verify and store any smart contracts. This means that neither the casino nor the user needs to trust one another but can ensure that the other party acted honestly. This solves any claims of non-repudiation since it is clearly specified on the blockchain which party carried out which actions.

Given enough technical knowledge this allows users to verify whether a casino is provably fair and follows through on their guarantee, greatly reducing the potential for a user to lose their funds unfairly. A third party consultant can also be used to verify the smart contracts if the user is not technically proficient enough to do this themselves.

There are other benefits to online casinos interacting with the blockchain but this is the one the paper focuses on.

Recently the cryptocurrency space has come under a lot of scrutiny for being untrustworthy and in some cases outright scamming people such as in the case of FTX [7]. However the reason for these frauds has been either a company introducing its own blockchain and having the majority of control over it, making it centralised or a company committing fraud and lying to stakeholders. The blockchain itself has remained mathematically sound, still acting deterministically. Therefore if a decentralised blockchain such as Ethereum is used and it is ensured that a casino is actually deploying smart contracts on a blockchain then it is extremely difficult for a casino to commit fraud by offering dishonest or malicious gambling options to users. By being hosted on the blockchain users can themselves verify that a transaction happened as promised and users can ensure that no one party such as the casino was able to influence the bet in any way. Therefore using a decentralised blockchain adds security, verifiability and transparency to a user's bet

2

Related Works

The idea of gambling online is a relatively new one and the idea of gambling using blockchain is even more recent. Therefore there is little formal literature on it, however there are commercial decentralized applications (Dapps) who release whitepapers containing technical details on how their applications are built and why they are built in this way.

The most cited academic work on the subject is by Gainsbury. The work discusses online casinos that are able to evade or not subject to regulation[10]. This greatly increases the risk to users who no longer benefit from the protection of government oversight. These casinos have had the ability to manipulate the odds of the games that they offer unbeknownst to users as well withholding user's funds at will. The paper discusses the option of utilising blockchain technology to remove the abilities of the casino to do these things. It also mentions that the limit on transactions on blockchains presents an issue to scaling a Dapp. While untrustworthy casinos will still be able to exist users now have more trustworthy alternatives should they wish to use them.

Dapps such as SatoshiDICE do allow users to execute bets to happen directly on a blockchain[16]. A bet is placed by sending a transaction to one of the smart contract offered by SatoshiDDICE. However the bet works with 0 confirmations which allows the winnings to be near instant and means that the random number was not generated using a decentralised oracle. It also means that the transaction has small chance of being reverted in the case of a reorganization attack.[6] Therefore while the bet is present on the blockchain afterwards it is not guaranteed that the random number was generated securely.

There is a different type of Dapp that allows users to place bets on top of a blockchain using ERC-20 tokens[**Smith_2023**]. FUN is an example of such a token developed by FUNFAIR casinos. These are tokens whose issuance and rules are regulated and follow the ERC-20 standard. These tokens can be bought and sold on exchanges such as binance but are also used to place bets[**binance**]. This allows users to use the blockchain but users are now exposed to the price fluctuations of the token, which has no fixed price, and must go through the process and cost associated with buying and selling these tokens. They also once again do not have the guarantee of using decentralised oracles to generate random numbers.

3 High Level Project Specification

The final product of this paper is a proof of concept that a gambling application can work using smart contracts. The focus will be on the smart contracts and the interaction between them and the front end. This is because the front end is simply a website that an online casino would normally operate and the user experience should be similar to a regular online casino. The main difference being that they sign transactions using an Ethereum wallet. The user can play a simple game of roulette where the user can place a bet according to roulette rules [13]. The user will then be able to see their transactions that took place on the Ethereum blockchain. The user will be given all the available information necessary to fully verify that the bets were carried out as promised. This means that the correct smart contract was used, the transaction of the bet had the correct values and that the random number generated was generated honestly. If the bet is on the blockchain but the smart contract does not generate the numbers in a provably fair manner then the user cannot guarantee the honesty of the bet.

The product will only be deployed to a testnet. This is because gambling poses a large risk to users due to its addictive nature and potentially damaging consequences [15][17]. There are also no available funds to test this on a real blockchain. Any actual deployment of this application will cause the deployer to take on significant risk since anyone is able to interact with the publicly available smart contract. So the entity that is hosting the smart contract is enabling gambling and may be subject to the laws and regulations of any country that a user is gambling from. While the technical skill required for this is relatively high it is still a major risk[12]. Therefore the smart contract has a requirement that only bets that have been accompanied by a signature from the Dapp are processed.

3.1 Low Level Project Specification

3.1.1 Login

Logging in is done through the user's Metamask wallet. This means that they don't require a username or password for this Dapp and reduces the sensitive user information stored by the casino. It is necessary to use a wallet such as Metamask since the Dapp needs to connect to the blockchain to function. The user will then use that wallet to sign a transaction using their private keys. This verifies that a user can access the funds contained in a wallet. The user will

use this wallet to sign any subsequent transactions required to place bets. The login currently only works with the Metamask browser extension.

3.1.2 Register

For a user to place a bet they must be on a whitelist of approved addresses. This is to ensure only users legally allowed to gamble can gamble, because it prevents people from placing bets using the smart contract directly on the blockchain. Since the laws and regulations vary vastly between different countries and jurisdictions no checks have been implemented. It is however required that the request be sent through the Dapp to add an address to the whitelist. This is done by requiring a message that has been signed by the Dapp to be sent with the transaction. It will be up to further extensions of this project to implement KYC/AML checks according to local laws.

3.1.3 Verify Contract, Bets Random Number Generated

The contract and the user's bets are made available to the user in order to verify their bets. The contract address is provided and a link to the deployed contract is given on the page. This is viewed through the block explorer Etherscan; however with the contract address the contract can be viewed through a user's node or a block explorer trusted by the user. For each transaction the transaction hash is provided. Similar to the contract address the transaction hash can be used by the user to verify that the transaction did indeed take place and with the results shown by the Dapp. If the user really has doubts they are able to view the transactions in their Metamask wallet and extract all the needed information from there. The verification process is technically difficult and time consuming for the average user to do. However it is possible for them to make these checks and not have to depend on any third party. It is also possible for the user to enlist a third party to help them in this and this task is trivial for any entity with extensive knowledge of EVM blockchains. It is also not the goal that the user should verify every transaction, as the vast majority of online gambling occurs with no issues. The goal is to allow for a decentralised reliable record of what happened so as to resolve any issues or disputes that may arise between the casino and the user.

The user can verify that the random number was generated honestly by examining the smart contract and ensuring that it uses the Chainlink VRF [3]. This is an industry standard method of generating provably random numbers in a decentralised manner on a blockchain. It has undergone security audits and has widespread use on the Ethereum blockchain and the user can examine these in the case of extreme doubt. However it is currently the only option the user has when using this Dapp to place bets.

4 Design

4.1 High-level System Architecture

The Dapp is permanently deployed on the sepolia testnet as a minimum viable product (MVP). It is on there so long as the testnet is maintained and can be viewed using any blockexplorer under the address "0xfe56B6A93Faec2878DB7731FC61F7BCAfeD6517e". It was deployed on the sepolia testnet because it is EVM compatible and it is relatively easy to get testnet ETH. The testnet must be EVM compatible to ensure that the contract can be deployed to the Ethereum mainnet if desired with no changes. It also means that the contract will behave the same way on the sepolia testnet as the Ethereum mainnet. It was also necessary to deploy on the sepolia testnet because the Chainlink vrf is not available on all networks. This is because the providers of the oracle have to deploy their token and their contracts to the relevant chain for it to work correctly. They also need to ensure that there is a decentralised group of nodes acting as validators, generating random numbers and ensuring that the numbers that are generated randomly are validated.

The back end is the main focus of this project. It consists of one primary solidity smart contract which contains the functionality needed to successfully play a game of roulette and anything else needed to successfully do so such as call an external oracle. The contract inherits and uses code by other smart contract such as a slimmed down version of OpenZeppelin's SafeMath contract and the contracts required by Chainlink to interact with their verifiable randomness function in order to generate a random number[3] [4]. This is however deployed in one large contract because it is more efficient in terms of gas costs than to deploy each contract seperately. This does not come at the expense of readability as each contract is still defined in its individual .sol file.

4.2 Front-end structure

The front end is the minimum required in order to allow a user to interact with the smart contracts and play a game of roulette. It was developed using React. This is because it allows for faster and simpler frontend development that comes from the components offered by react. The Javascript library web3 was used to allow the frontend to interact with Infura. Infura was used as an ethereum node to query and interact with the sepolia tesnter. The frontend is styled

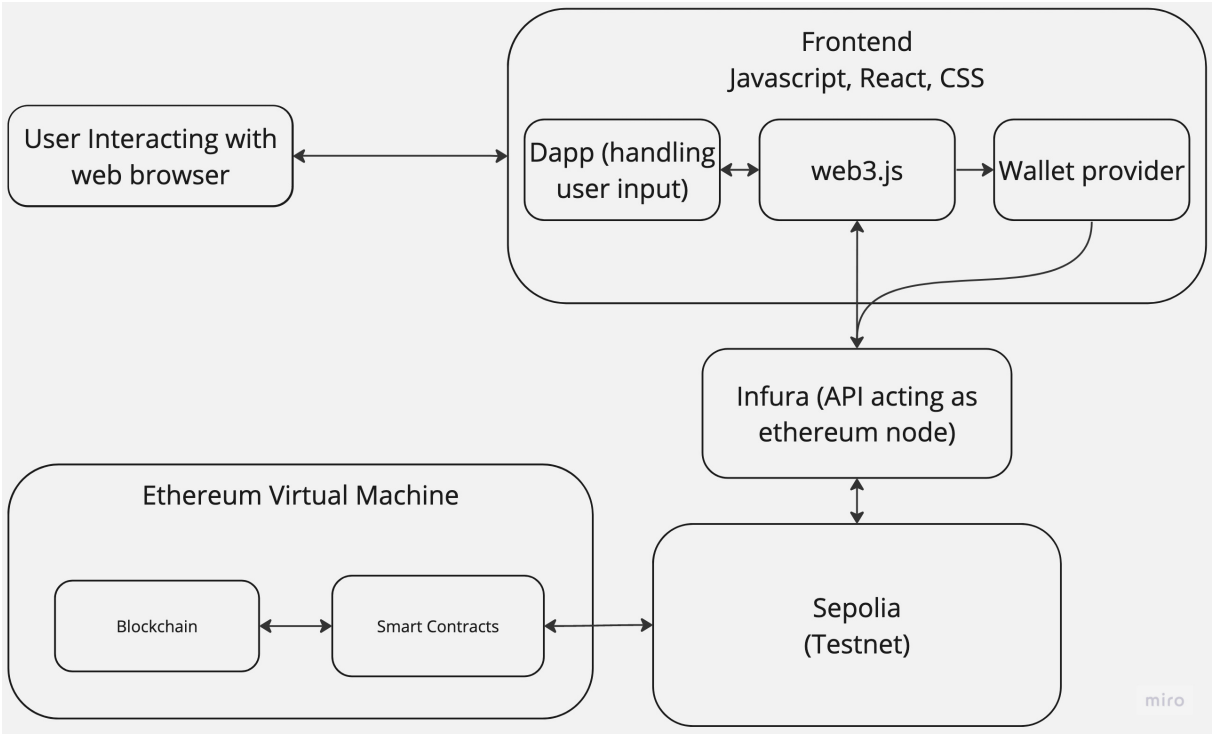

Figure 4.1: High Level Architecture.

using a mixture of react components and a css library bulma. This allows for a frontend to be styled with extremely limited css. This was because the frontend was minimised to allow any future developers to focus on the backend of this Dapp and implement the frontend to their specifications.

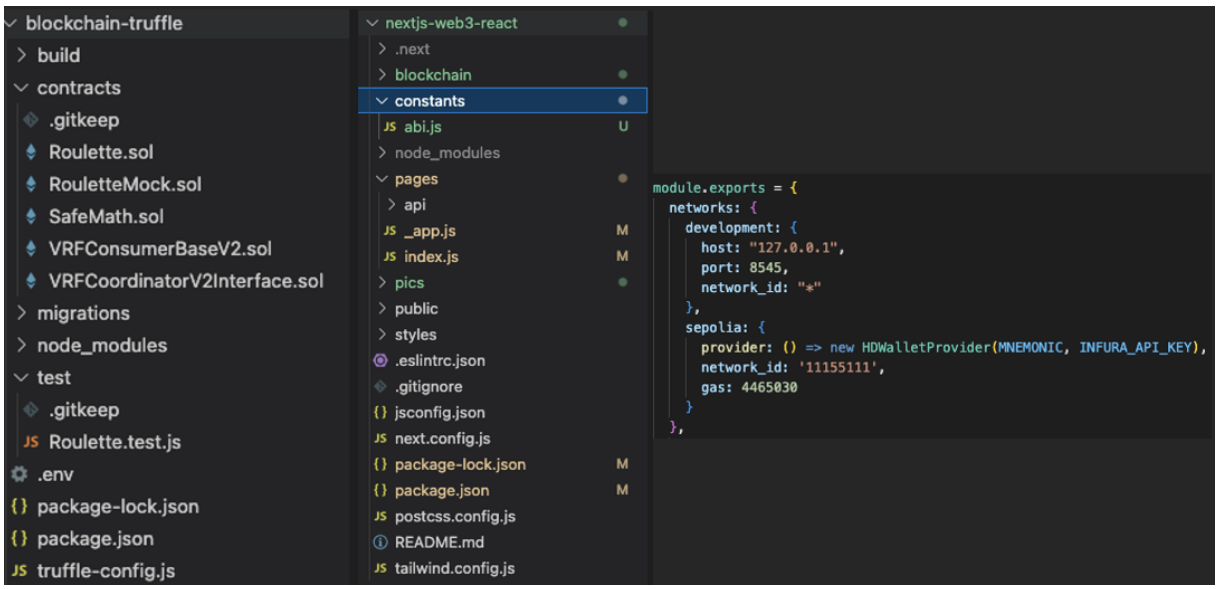

Figure 4.2: Back end file structure (L), Front end file structure (C), Truffle config file (R).

4.3 Back-end structure

The backend consists of 4 different smart contracts. The primary one is "Roulette.sol" responsible for aggregating and calling any other smart contracts necessary and the majority of the functionality. There is "SafeMath.sol" which is a stripped down version of the "SafeMath.sol" by OpenZeppelin that removes functions that aren't needed by "Roulette.sol". This is to ensure there is no arithmetic underflow or overflow during mathematical operations.

This isn't strictly necessary in version 0.8 and upwards of solidity. However it ensures the Roulette.sol contract will also be safe to use with previous versions of solidity and provides an extra check in addition to the compiler's run time checks. The other 2 contracts are the "VRFConsumerBaseV2.sol" and "VRFCoordinatorV2Interface.sol". That allow the Roulette contract to request chainlink generate a random number and retrieve it when the number is generated.

The truffle framework is used to compile and deploy these contracts . The truffle framework is responsible for compiling the contracts. It does this by first compiling the different contracts into build artifacts in the build/contracts file. Build artifacts are JSON files which contain anything needed to deploy the contracts such as the application binary interface (abi), bytecode and metadata. The abi of a smart contract is a JSON formatted external interface for a contract that allows external applications and other smart contracts the ability to interact with it. It is similar to an API in traditional web development. The truffle framework is then responsible for migrating or deploying the smart contracts to a network or blockchain. How to deploy these contracts is specified in the "1_initial_migrations.js" and "truffle-config.js" files. The migrations file specifies how the contracts should be deployed to the blockchain. In this project the contracts were aggregated and deployed in one smart contract as opposed to 4 different contracts to save on gas fees. This does reduce the readability when viewing the code of the contract on the blockchain. However on etherscan where the source code has been published the contracts are seperated again to make it more readable for developers. The config file is responsible for specifying the network to deploy the contracts to (in this case the sepolia testnet), which address is responsible for deploying the contract and what node is used to query the blockchain and deploy the contract. An Infura node was used to deploy the blockchain . Infura is an industry standard node used to avoid the hassle of having to spin up an ethereum node oneself and is largely known as reliable. If there are any concerns a different service could be used or one can host their own node. The private keys are necessary since the deployment to the blockchain requires gas which must be paid for, the deployer of the contract is also set to be the owner, this can be changed later using the changeOwner function. The private keys for the address responsible for deploying the contract and the API key used to use the Infura node are held in a seperate .env file. These need to be configured by any potential developers themselves and should be kept private as leaking them can lead to lost funds.

5 Development

The primary focus was on allowing the user to place a verifiably honest bet. This meant that they should be able to see on the blockchain how a bet was placed and how the random number for that bet was generated. The placing of the bet and the generating a random number were split into functions of the smart contract.

5.1 Generating a Random Number

For a user to play a game of roulette a random number must be generated. This simulates rolling a ball around the roulette wheel and having it randomly stop on a number. This is done using a decentralised oracle offered by Chainlink[3]. This works by having nodes generate random numbers upon request. These numbers are then broadcast and verified by other nodes. The proof for these has been extensively audited by third party blockchain security companies and can be viewed on the chainlink website. The random number that is generated is then stored on the blockchain and can be retrieved at any point.

The requests are made and retrieved using 2 contracts provided by chainlink. One is the "VRFCoordinatorV2Interface" contract and is deployed by chainlink on the Sepolia testnet. It is responsible for making the request to generate random numbers and managing the verifiable randomness function (VRF) subscription. The subscription is a way to add "consumers" or smart contracts that can request random numbers. The subscription must be funded with the LINK token. This is to pay the nodes responsible for generating the random numbers. If a smart contract is not a consumer of a subscription or if the subscription does not have enough LINK then no request will be sent. This contract is responsible for sending the request using the function requestRandomWords. The roulette contract calls this function and passes in 5 variables. The keyhash, the subscription id, the minimum request confirmations, the call back gas limit and the number of random numbers to be generated. The keyhash is used as a key to generate the VRF proof and has a gas price ceiling, in this case 30 Gwei. The call back gas limit is how much gas should be used when calling the fulfillRandomWords callback. The gas limit is 30 Gwei which means that if the gas price is above that then random numbers cannot be generated. This is unlikely to be an issue since the gas price has been nowhere near that for multiple years and if it does increase then a different key hash can be used [5]. However this stops a user from spending more than 30 Gwei on gas, which wouldn't make sense given the current value of

bets being placed on the Dapp. After calling the function requestRandomWords the number is generated and verified and when this is done the contract calls the rawFulfillRandomWords function in the other contract.

The minimum request confirmations is the amount of blocks the oracle should wait after generating the random number before responding to the request. This is of significance because it represents a direct trade off between security and speed in this Dapp. The more confirmations a block has the more secure it is, being less vulnerable to a reorganization attack[6][2]. This is because it requires more blocks to be reorganised in order for the original block to be altered or removed from the blockchain and therefore comes at a greater cost to an attacker. If an attacker were to reorganise the blocks and alter the random number that is generated on the blockchain this may mean that a bet has been altered. If a casino offers large enough bets it may be beneficial to an attacker to alter the outcome to ensure that they win. Therefore it is desirable to have a high number of block confirmations. However each block confirmation is on average 15 seconds and this is a large amount of time for a user to wait for a bet to process. Currently 3 block confirmations are used, which is standard however this means that in addition to the inital block where the bet is placed and the block to end a game 1 bet takes on at best 75 seconds. This does not include any time waiting for a user to confirm the second end game transaction and assumes that the transactions are immediately processed and not waiting in the mempool. The 3 block confirmations were selected because of the balance between speed and security however it may be a consideration that if high amounts of money are bet that the random number requires more block confirmations before being processed.

```
function fulfillRandomWords(uint256 _requestId, uint256[] memory _randomWords) internal override {
    require (LinkRequests[_requestId].exists, "Request was not found"); //checks whether a request has actually been made
    LinkRequests[_requestId].randomWord = _randomWords[0];
    LinkRequests[_requestId].requestFulfilled = true;
    emit RequestStored(_requestId, LinkRequests[_requestId].randomWord);
}
```

Figure 5.1: fulfillRandomWords function, called by a Chainlink node when the number has been stored.

The second contract is called the "VRF2ConsumerBase". It contains two functions fulfillRandomWords() and rawFulfillRandomWords(). The rawFulfillRandomWords function calls the fulfillRandomWords function but can only be called by the coordinator. The fulfillRandomWords() function is an abstract function that is implemented by the "Roulette" contract. It is called by the coordinator contract and takes in the request id and an array of random numbers generated. The roulette contract simply stores the random number generated in a struct called LinkRequest.

The gas to call the fulfillRandomWords function is paid by the user (who sent the request to generate a random number). The amount of gas used to do so is the callBackGasLimit defined in the Roulette contract and the maximum amount is defined by the coordinator. Adding more functionality to this implementation of fulfillRandomWords leads to this gas limit being reached. This means that the request is made, the LINK token is paid but no actual number is generated or returned. That is why the implementation does nothing more than store the

```
struct LinkRequest {
    uint256 requestId;
    bool exists;
    bool requestFulfilled;
    uint256 randomWord;
}
```

Figure 5.2: linkRequest struct.

random number generated and change the randomWord field of the linkRequest to the random number generated and the requestFulfilled field to true. It is also important to make sure the fulfillRandomWords function uses a consistent amount of gas for the same reason as above.

5.2 Placing a bet

Placing a bet consists of three separate actions. The first is choosing the numbers that the user wants to bet on. The second is choosing the amount they want to bet. The third is sending the transaction to the blockchain. This is the most complex and will be split up into its backend and frontend components.

Choosing numbers can be done in 2 separate ways. The first is by inputting up to 6 comma separated numbers from 0-37 into a text field. Any change to the input will trigger an event and the numbers that are stored will update and show on screen accordingly. No more than 6 numbers will be processed and sent to the smart contract. The number 37 corresponds to the second green tile "00" a common abstraction in roulette. The other option is to click on one of the buttons with different options that allow for a group of numbers to be selected. Clicking on a button calls a function that selects a group of numbers corresponding to the buttons pressed. For example, the 1st twelve numbers correspond to [1,2,3,4,5,6,7,8,9,10,11,12]. Clicking a number or updating the input field overrides any previous choices. The most updated choice that will be sent to the smart contract is shown to the user.

Choosing the amount that a user wants to bet is done similarly to inputting the 6 comma separated numbers. The user inputs a number in ETH and every time this field is updated the previous value is overwritten. The value of the bet is displayed to the user in ETH and WEI. If they are below or above the minimum or maximum stake an error message will display. If the stakes of the smart contract are updated, then this does not have to be updated on the frontend as the stakes are dynamically called every time a user logs in.

5.3 Placing a Bet: Frontend

Placing a bet is where the frontend interacts with the smart contract. The placeBet method of the contract is called from the user's address. The bet that was placed by the user is passed

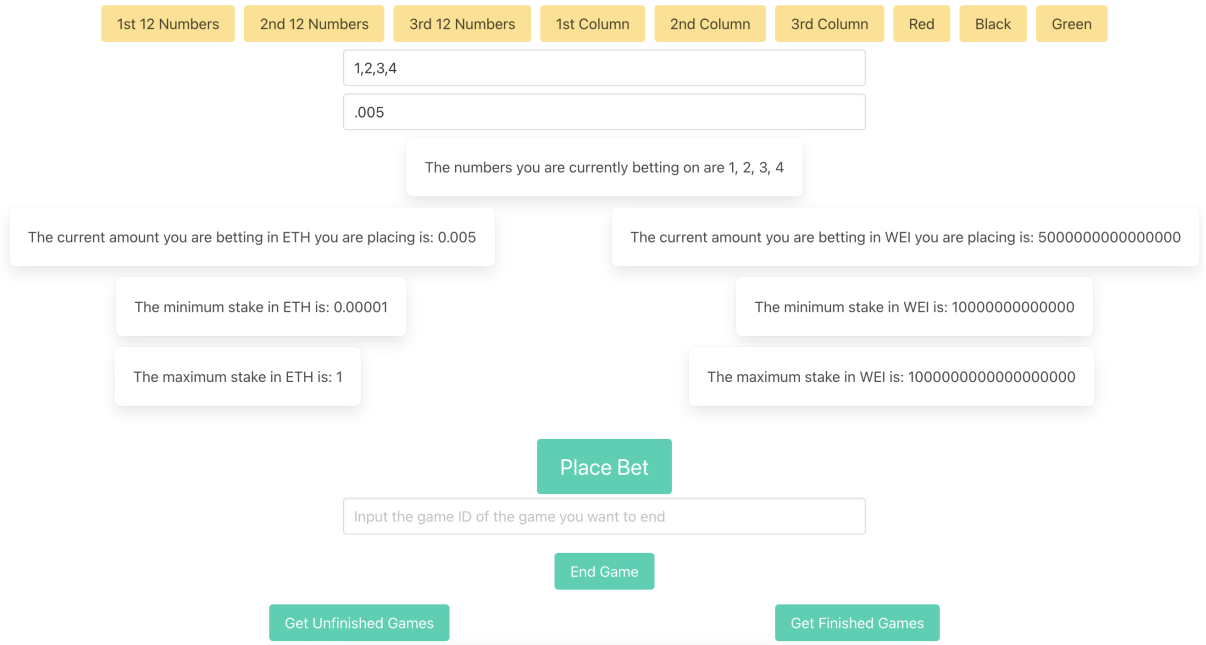

Figure 5.3: The UI that allows a user to customise and place a bet.

through as an argument and the value that the user wants to bet is passed in as a msg.value. When the method is called a Metamask prompt will pop up requiring the user to authorize the bet. The price of the bet will be the value they bet in addition to any gas fees. The gas limit of this transaction is set. This does not mean that the gas limit will be used it simply means that no more than that can be used. The actual gas limit will be set by the user through Metamask when they approve the transaction. It is however necessary to set this gas limit higher than any actual gas limit set by the user to prevent the transaction reverting due to gas estimation errors. After the user has approved the transaction they will be unable to place any other bet in the meantime.

The frontend will then create a listener that waits for the Event BetPlaced that is emitted by the smart contract. This Event is triggered when the bet is placed and the request to generate a random number has been sent. The front-end will store the game id so that it can later call the method endGame that takes in the game id as an input. It will then let the user know that the bet has been successfully placed. The frontend will also then create a listener that waits for the event RequestFulfilled that is emitted by the smart contract. This event means that the random number has been generated, verified and has had three confirmations. The front-end will then call the endGame method on the contract with the game id as an argument from the user's address. This means that the user again must verify a transaction to fully complete a game of roulette. If the user rejects the transaction and does not want to finish the game or wishes to do so later then they can end the game later using the endGame button. The user is however heavily incentivized to finish each game since when they place the bet. This is because the smart contract holds the user's funds in escrow until the game is finished. This is to prevent users starting a game then viewing the blockchain to see what random number was generated

and only continuing the game unless they win. If the user sees that they lost they can not end the game using the endGame function in order to save on gas fees. It will not make a difference to the balance of the smart contract.

```
await contract.events.BetPlaced({ filter: { gameId: desiredGameId } }).once("data", function (event) {
  setSuccess(true)
  setSuccessMessage('Successfully placed bet!')
  gameId = event.returnValues.gameId
  filteredRequestId = event.returnValues.requestId
  setBetOngoing(false)
}).on("error", console.error);

await contract.events.RequestStored({ filter: { requestId: filteredRequestId } }).once("data", function (event) {
  setSuccess(true)
  endGameHandler(gameId)
}).on("error", console.error);
```

Figure 5.4: Listeners registered by the frontend to check when a bet has been placed and when a number has been successfully generated.

The frontend will then create a listener that waits for the event GameFinished that is emitted by the smart contract. This means that the smart contract has finished the game of roulette, determined whether the user won any money and sent money if they did. At this point the transactions of the roulette game are added to the History section of the website and the user can place another bet if they wish to continue playing.

When a user places a bet or ends a previously unfinished game the Place Bet and End Game buttons are unable to be used by the user. This is so that each transaction by the user is placed in a different block. This is to ensure that the user's bet or ending of the game was successful before allowing another to occur. In the case of an error occuring the transaction will revert and the user will be made aware of any error. The user is allowed to place bets after the bet has been stored on the blockchain. The user does not need to wait for the random number to be generated and be 3 confirmations deep before placing another bet. This is primarily because this entire process takes about 60 seconds and this is a very long time to wait for a bet to complete.

The frontend also does not allow the user to input any data that will cause the smart contract to revert. As can be seen in Figure 5.5 functions have multiple modifiers and revert statements such as making sure that the bet placed is above the minimum stake. The modifiers ensure that certain conditions are met such as onlyDapp requiring the user to register through the frontend. The frontend can still handle issues if the contract reverts, however aims to prevent the user from doing so as it makes for a worse user experience. The reverts are still useful because there may be a bug in the frontend that allows unssucessful to be placed through. It also allows the contract to be directly copied and used by others without having to add in any frontend checks themselves.

5.4 Placing a Bet: Backend

The placeBet function takes as an input an integer of uints representing the numbers the the user placed a bet on. It has three modifiers which ensure that the user has bet the minimum

stake and that the bet is not above the max bet and that the user address has been registered. The exact stakes can be decided by the casino and modified while the contract is deployed. There are then multiple require statements, which contain conditions that must be met or the transaction will revert.

If these conditions are met then a request is made to the coordinator contract and a random number is generated. A Game is added to the Games struct containing all the necessary information to complete the roulette game as well as provide a dev or the user any information they may want. At this point the BetPlaced event is emitted and the frontend can process the event and act accordingly. The function is shown below.

```
function placeBet(uint256[] memory _numbersSelected) public payable onlyValidUserStake(msg.value) onlyRegistered(msg.sender) onlyValidHouseFunds(msg.value){
    require (msg.sender.balance > msg.value, "Not enough money in wallet to place bet");
    require (_numbersSelected.length < 7 && _numbersSelected.length > 0 || _numbersSelected.length == 12 || _numbersSelected.length == 18, "Need to make a valid bet");

    uint256 requestId = COORDINATOR.requestRandomWords(keyHash, subscriptionId, requestConfirmations, callbackGasLimit, numWords); //will revert if subscription is not set
    and funded
    LinkRequests[requestId] = LinkRequest(requestId, true, false, 0);
    emit RequestSent(requestId);

    gameId = SafeMath.add(gameId, 1);
    Games[gameId] = Game(gameId, _numbersSelected, msg.value, payable(msg.sender), false, 0, false, requestId, 0);
    emit BetPlaced(gameId, _numbersSelected, requestId, msg.value, payable (msg.sender));
}
```

Figure 5.5: The placeBet function that is called by the frontend.

Once the random number has been generated and is 3 block confirmations deep the fulfillRandomWords function is called by the Chainlink contracts. This stores the random number as described above. The event RequestFulfilled is emitted and the frontend can process the event and act accordingly. The reason that endGame is not directly called from fulfillRandomWords is because of the gas call back limit as previously described.

The endgame function is called by the frontend and the transaction is verified by the user. Similar to the placeBet function the endGame function contains require statements with conditions that have to be satisfied in order for the transaction not to revert. These require statements ensure that the game has not yet been completed, that the game exists and that the random number has been generated. The random number generated is processed to be modulo 38 so that it corresponds to a number on the roulette wheel (37 represents 00). It is then checked whether the number matches any of the numbers that the user selected as a bet. If it is a match the payout is determined based on the amount of numbers that the user chose and they receive their winnings accordingly. The Game struct is updated according to the results and the GameFinished event is emitted which the frontend processes. The endGame uses helper functions such as calculatePayout and safeTransferWinnings to split up functionality. This makes the code more readable and maintainable. It also means that functionality is abstracted away from the user, because these helper functions can only be called by the smart contract. The endGame function does not have any restrictive modifiers, this is because regardless of who ends the game any winnings are transferred to the address that started the game.

6 Testing

The smart contract is the most important thing to be tested. This is not only because it is the focus of the paper but because it is the aspect most vulnerable to exploit and hardest to change. Once a smart contract has been placed on the Ethereum Blockchain it cannot be changed and altered. Therefore it should be ensured that the smart contract behaves exactly as desired. This was done using unit testing and code coverage.

Unit testing ensures that the desired functionality happens when the smart contract is called and executed under different conditions. This means that the contract doesn't only have the desired functionality but also that it fails and rejects attempted bets under the right conditions. The truffle framework was used to test the code in a local environment. This allowed for rapid testing with instant block confirmations that is not limited by testnet fees. It means that if the contract is deployed in an insecure manner there is no malicious actor on the testnet network which can exploit any vulnerabilities. This does come with the drawback that the random number verification cannot be unit tested. Therefore a mock contract called 'RouletteMock.sol' was made with minor alterations from the 'Roulette.sol' contract. The difference between the two contracts is that the commented out line in the image below is not commented out in Roulette.sol. To replace the functionality of this line and Chainlink calling the fulfillRandomWords function with a random number 3 lines were added to generate a number used as a mock and to call the fulfillRandomWords function.

```
// uint256 requestId = COORDINATOR.requestRandomWords(keyHash, subscriptionId, requestConfirmations, callbackGasLimit, numWords); //will revert if subscription is not set and funded
uint256 requestId = SafeMath.add(gameId, 1);
uint[] memory randomWordsMock = new uint[](1);
randomWordsMock[0] = uint (39);
fulfillRandomWords(gameId, randomWordsMock);
```

Figure 6.1: The code added to the 'RouletteMock.sol' contract.

The truffle framework hosts a local EVM compatible blockchain using Ganache. All the tests ran successfully on the local blockchain and the results can be viewed in the appendix. The Chainlink vrf itself did not have to be tested as this has been done extensively by Chainlink and 3rd party security audits and has been done far more extensively than the scope of this project allows. The functionality of the 'Roulette.sol' contract using Chainlink has been tested extensively when evaluating the product by allowing users to use it. The tests run can be viewed in the appendix.

7 Evaluation of the Final Product

In order to evaluate the product I conducted a usability study with 17 people made up of friends and family who did not have any known history of having any kind of gambling problem. The goal was for the users to successfully place bets and then verify themselves that these bets have happened and are present on the blockchain. I also wanted them to be able to view the contract and ensure that they could view the code and be satisfied that it functioned as promised. A usability study was conducted because it directly measures the ability of the Dapp to provide users a way to place fair and transparent bets. If the users were unable to verify the fairness of the bets or place bets for any reason then the Dapp would not be successful.

The users first had to place a bet. This task was consistently successfully completed, after the initial setup of the wallet and transferring them some testnet funds they were all able to easily place a bet. The instructions on the page made it clear how to get registered and how to place a bet with their desired values and amount of testnet funds. The instructions can be viewed in the appendix. The users also found the error messages simple and clear. The errors that were present were due to people betting too much or too little money. This was due to their infamiliarity with ETH and WEI and how much it is worth relative to GBP, however after seeing the fields saying the minimum and maximum stake this was no issue. They also found the information on how the random numbers were generated and the Chainlink website useful.

There were two main critiques. The first was how long the bet took to process. This is primarily because the smart contract waits for 3 block confirmations before using the random number which theoretically takes 45 seconds. However because the number has to be generated and may have to wait for a previous block to be mined it normally takes around a minute. The second was the issue of requiring to confirm two separate transactions. This was unavoidable in order to generate random numbers without incurring very high gas fees. Since there were a limited amount of test funds the gas fees must be kept low. This can be changed in the future simply by calling the endGame function from the fulfillRandomWords function. However it would require that the bets be of a large amount for this to be economically feasible.

Despite these critiques they have to do with the user experience of placing a bet they do not affect the honesty of the game or the ability of a user to verify whether a game happened honestly. Users all unanimously believed that the transaction had truly taken place on the blockchain and were satisfied when the details of the bet appeared on screen when they were done. The reason they trusted this was because the Dapp provided a transaction hash that matched the one

provided by their Metamask wallet which ensured that the transactions actually did happen. They were also able to verify that the smart contract that the Dapp said it used and they took a look at was the same one that they were sending their Metamask transaction to. This can be viewed in the Metamask pop up that asks the user to verify this transaction.

The users then attempted to verify that their bet happened on the Ethereum blockchain. They were able to use Etherscan to view the transaction by inputting the transaction hash. This provided the user with the transaction details for the end of the game. The user was able to view the events emitted by the contract during the endGame function as well as the change in state that was caused by the transaction. This means that the user was able to see that their bet was processed properly. Users were also able to view the initial transaction where the random number was generated. They are able to view the requestId of the Chainlink VRF request as well as the chainlink node that is responsible for generating the number and the number that it generated. In this way the user can guarantee that the random number was generated fairly and is available on the blockchain for them to view at any point. They are also able to see the full path of their bet and all the values that were used to process. This ensures that the user can verify that the bet was carried out as intended. This means that users do not have to rely on the Dapp and can settle any potential disputes using the data stored on the blockchain.

The users are also able to access the bets that they placed on the blockchain without the website providing them the information. Each transaction that interacts with the smart contract can be viewed using any block explorer. This allows the user to view their transactions without needing to rely on the casino. Each method and event is clearly labelled to make it as easy as possible for the user to verify what exactly happened.

The final evaluation was done by having users attempt to verify that the contract said what it did. This was also done by using Etherscan. The link to where the contract can be viewed on Etherscan as well as the address of the contract is available on the Dapp. The users were able to follow this and ensure that the contract was hosted on Etherscan and functioned as was promised. This represented a bit of a technical barrier as it required the ability to read code. However most of the code was simple and the users were left satisfied that there was no foul play. The main confusion was on how the random number was generated but since they had previously been able to view that it was generated safely by a group of decentralised Chainlink nodes there was no doubt that the number was generated in a provably fair manner. The events that can be viewed by the user can be seen in the Appendix.

The main critique was that users would enjoy having different decentralised oracles to choose from in case they the considered the VRF offered by Chainlink to be unreliable. This could be implemented in future versions of the Dapp. The users did however enjoy the simple UI and said they enjoyed this because using blockchain technology was already new and complex for them to deal with so the simple UI helped to focus exclusively on what was happening on the blockchain. The users also really enjoyed not having to rely on the casino to store and verify their bets and the assurances that the blockchain provides. They especially enjoyed that multiple block explorers could be used to view the transactions and that the transactions would be stored if the casino were to go offline or bar them from accessing the site.

8 Conclusion

8.1 Critical Assessment

The project was successful in meeting its primary goal of providing a way for users to ensure that the game of roulette they played was fair and the casino did not alter the outcome in any way at all. It also provides a method to settle any potential disputes, especially those concerning non-repudiation. It met the majority of the goals set out in the literature review. The functional requirements of the literature review were to be able to process a bet, customise a bet, have the game be provably fair and securely generate random numbers. The non-functional requirements of the literature review were to be usable, auditable, scalable and secure.

The smart contract was able to process a bet successfully having the placeBet and endGame functions called. The user was able to customise what numbers they wanted to bet on as well as the amount in accordance with classical roulette rules. The Dapp is therefore usable.

The bet is provably fair but not as described in the literature review. The literature review involved having the casino generate a seed, hashing it and sending it the user. The user would then generate their own seed, concatenate it with the hash and provide it to the casino. This was prone to the user influencing the bet by manipulating the seed they generated. The implementation where a decentralised oracle generated a verifiable random number that cannot be pre-determined means the bet cannot be influenced by the casino or the user in any way. It also means that proof that the generation of the number was random is present on the blockchain and can be verified. The actual implementation is an improvement on the proposed solution so the requirement has been surpassed.

The bet having its random number generated on the blockchain ensures that the bet is fully auditable by users who can view the generation of the number on the Ethereum blockchain. The bet being processed is also able to be viewed by users on the blockchain. This is in addition to the smart contract being able to be viewed on the blockchain, allowing users to assess what it does. A user can view all of the above, even less technicaly capable users. The information is clearly displayed and readable on Etherscan and it is very easy to extract the necessary information provided to audit and view a transaction from the information that has been provided to the user on the Dapp. This was made clear by the user evaluation.

The Dapp is scalable. It can handle as many transactions as the Ethereum blockchain can. If this were to reach scale it would however cause gas fees to significantly increase. This would

limit the gambling to large amounts which would limit the accessibility of gambling to those unable or unwilling to place larger bets. This limits the scalability of the application however it is unlikely that it would grow to such size. The main issue to scalability is the time it takes to place bets with each bet taking over a minute to fully process. This may lead users to place less bets as they wait for the outcome of previous bets, however they are still able to place as many as they want it simply takes a while. This is unfortunately a tradeoff between speed and security discussed in the development section. It could perhaps be an option where a user can opt for a faster bet that is less secure or a slower bet that is more secure.

The Dapp is relatively secure. This is because the smart contract's functionality cannot be changed in any meaningful way once it has been deployed to the blockchain. It also requires any changes to be done by the owner of the contract and this will prevent malicious actors from making unwanted changes. The smart contract also only allows bettors who have been approved by the casino to bet which removes the danger of illegally allowing users to bet using the smart contract. The smart contract has undergone unit testing in order to determine that it has the desired functionality and been tested through different branches or paths in the code. The contract cannot be claimed to be fully secure until it has undergone a full security audit, however that is beyond the scope and budget of this project.

8.2 Future Work

There was not a complete implementation of a user verification feature because it depends on the local laws a casino is subject to and is beyond the scope of the project. This could be done off or on the blockchain using a smart contract but this may raise privacy issues as a wallet is pseudonymous and is a tradeoff that should be addressed.

There was also only the availability of one decentralised oracle for the user to choose from. This means that if they distrust the Chainlink VRF then they cannot use the Dapp. Providing more options would give the users more choice and ultimately trust. The main reason this was not done is because the logic to implement each oracle is vastly different and could be its own project. Many decentralised oracles also required a token to pay for their services which were not free unlike the Chainlink VRF token which was free on test nets.

The DApp can also be extended to provide more game options that more typically represent an online casino which presents a plethora of options. A very interesting extension would be how games with multiple players who each act in turn such as poker would work as before any player makes a choice they need to see previous player's actions. This means that there would be very long wait times for each game. Perhaps some sort of Second Layer solution could be brought in to escrow the funds from the users during the duration of the game.

In order to provide more scalability without compromising security as severely a blockchain with a shorter confirmation time could be used. This would allow for users to place more bets and have them process faster which makers for a more enjoyable user experience. If the project were to reach enough scale it could however start to raise the transaction costs on that particular blockchain and is something that should be kept in mind. This can however be avoided by

being deployed on an EVM blockchain with low fees and short confirmation time.

The Dapp could also be made more scalable by using a layer 2 solution. This is where the user funds are held in escrow during the game. The game then happens off chain with the user's balance and casino's balances being updated during each game. This could be implemented using a method similar to the lightning network [14]. This does however require that the user and the casino each store a revoke transaction for each update. If there are a large number of users are playing a lot of games this can pose a significant issue of where and how to store these transactions. There is also the risk that a party loses their revoke transactions, allowing the other party to steal funds if they become aware of this by posting a previous transaction state where their balance is greater than the most recent state. It also requires both party to monitor the blockchain while the games are being played off chain. This is an extra concern for the user who must now pay fees to a lighthouse to monitor the blockchain or monitor it themselves, which is technically difficult. For more information of this layer 2 solution consult the source[14], as it is beyond the scope of this paper. However because of scaling limitations different layer 2 solutions should be considered.

8.3 Conclusion

The main focus of the project was on the backend and could be directly implemented by a casino to provide fairer and more transparent gambling environment. For the purpose of this project the frontend was kept simple, it was however sufficient to create an MVP and is therefore sufficient for the project. If the Dapp were to be deployed in real life this is something that should be addressed, however it is outside the scope of this project.

The most important requirement, is the ability to place a bet that is dependent on a decentralised blockchain and not an online casino has been achieved. The random number generated for the bet was generated in a demonstrably and provably fair manner, ensuring that the bet cannot be influenced in any way. These two requirements being met ensures that the user had a demonstrably fair and transparent bet. The bet being hosted on the blockchain ensures that the user can view their bet at any point in time regardless of whether the casino is operational or even when the casino would not allow the user to do so. There are many potential additional features, such as those specified in the future works, that could be added to make the Dapp more scalable, secure, attractive and able to be used in a real world regulatory environment.

Overall the aim of this project was to create a Dapp that enables users to ensure that the casino was providing a demonstrably and provably fair method of betting using blockchain technology. The project has been successful in this matter with the MVP meeting the majority of its requirements and if it were to be implemented as is, it would function as set out in the objectives of the project and provide the user with a demonstrably and provably fair bet. Consequently it was fully transparent and relies on no single party. Therefore this project provides a solid foundation to build on and could be turned into a commercial real life experience providing a fair and transparent environment for casinos and gamblers.

References

[1] Oct. 2022. URL: https://www.gamblingcommission.gov.uk/statistics-and-research/publication/statistics-on-participation-and-problem-gambling-for-the-year-to-sept-2022.

[2] Aug. 2022. URL: https://ethereum.org/en/developers/docs/.

[3] URL: https://chain.link/vrf.

[4] URL: https://docs.openzeppelin.com/contracts/2.x/api/math.

[5] URL: https://ycharts.com/indicators/ethereum_average_transaction_fee.

[6] A. Averin and O. Averina. "Review of Blockchain Technology Vulnerabilities and Blockchain-System Attacks". In: *2019 International Multi-Conference on Industrial Engineering and Modern Technologies (FarEastCon)*. 2019, pp. 1–6. DOI: 10.1109/FarEastCon.2019.8934243.

[7] David Yaffe Bellany, Matthew Goldstein, and Emily Flitter. "Prosecutors Say FTX Was Engaged in a 'Massive, Yearslong Fraud'". In: *NY Times* (Dec. 2022). URL: https://www.nytimes.com/2022/12/13/business/ftx-sam-bankman-fried-fraud-charges.html.

[8] Sinyong Choi. "An Examination of Interactions of U.S. Players with Offshore Gambling Sites and Online Casino Reviews: Are They Offenders or Victims?" PhD thesis. ProQuest, 2022.

[9] Feelin froggy. *Rogue Casino Group Affpower caught hosting fake netent slots*. Jan. 2022. URL: https://lcb.org/news/9980-pirated-netent-games-sail-affpower-brands-straight-to-lcb-blacklist.

[10] Sally M. Gainsbury and Alex Blaszczynski. "How blockchain and cryptocurrency technology could revolutionize online gambling". In: *Gaming Law Review* 21.7 (Sept. 2017), pp. 482–492. DOI: 10.1089/glr2.2017.2174.

[11] Priscillia Mudiaki. *Online casino groups accused of stealing netent content*. May 2016. URL: https://www.gamblinginsider.com/news/2067/online-casino-groups-accused-of-stealing-netent-content.

[12] Janne Nikkinen. "The Global Regulation of Gambling: a General Overview". In: *Helsinki University, Department of Sociology* 03 (2014).

[13] David Oldman. "Chance and skill: A study of Roulette". In: *Sociology* 8.3 (1974), pp. 407–426. DOI: 10.1177/003803857400800304.

[14] Joseph Poon and Thaddeus Drja. *The Bitcoin Lightning Network*. Jan. 2016. URL: https://lightning.network/lightning-network-paper.pdf.

[15] Marc N. Potenza et al. "Gambling". In: *Journal of General Internal Medicine* 17.9 (2002), pp. 721–732. DOI: 10.1046/j.1525-1497.2002.10812.x.

[16] *Satoshi Dice*. URL: https://satoshidice.com/.

[17] Emanuel V Towfigh, Andreas Glockner, and Rene Reid. "Dangerous games: The psychological case for regulating gambling". In: *Charleston L. Rev.* 8 (2013), p. 147.

Appendix

8.4 Tests and Rules

```
Contract: RouletteMock
  ✔ Contract deployed correctly
  ✔ Successfully registers user (321ms)
  ✔ Successfully fails non-owner registering user (118ms)
  ✔ Successfully rejects an unverified user from playing
  ✔ Successfully fails if user stake is above or below the maximum or minimum stake (54ms)
  ✔ Successfully fails if casino does not have enough funds  (48ms)
  ✔ Successfully add funds to the casino (61ms)
  ✔ Successfully rejects invalid bets (137ms)
  ✔ Winning bet is Placed correctly, random number is stored correctly and events are emitted with correct values (399ms)
  ✔ Won game ends successfully and events are emitted with correct values (151ms)
  ✔ Lost bet is Placed correctly, random number is stored correctly and events are emitted with correct values (169ms)
  ✔ Lost game ends successfully and events are emitted with correct values (149ms)
  ✔ End Game Successfully fails if game has already finished
  ✔ End Game Successfully fails if game does not exist (47ms)
  ✔ Successfully doesn't allow non-owner to deregister users
  ✔ Successfully allow owner to deregister users (324ms)
  ✔ Successfully rejects non-owner to withdraw funds and withdrawal of more funds than available
  ✔ Successfully fails if attempting to withdraw more funds than available
  ✔ Successfully withdraw a specified amount of money (83ms)
  ✔ Successfully withdraw all remaining funds (67ms)
  ✔ Successfully prevents non-owner from changing owner
  ✔ Successfully changes owner (157ms)


22 passing (3s)
```

Figure 8.1: Successfully run tests using Truffle.

How a game Works: You select the bet and bet amount. You then click on the Place Bet button. You must then approve the MetaMask transaction that pops up. After a short wait a second MetaMask transaction will pop up. If you wish to end the game now then approve the transaction. Otherwise the game will be stored on the blockchain and can be finished later by entering the desired gameId and clicking the "End Game button". Finished and unfinished games can be viewed by clicking on the "Get Finished Games" and "Get Finished Games" button.

Rules: You may place a bet on a maximum of 6 numbers between 0 and 37. 0 represents the first green space (0) and 37 represents the second green space(00) If you wish to bet on a group of numbers use the buttons above the bet input to do so. The first 6 numbers ranging from 1-36 will be selected. The bet that will be passed through once you place a bet can be viewed by clicking "See numbers". Entering numbers other than 1-36 may lead to unintended consequences and your bet may be different than intended. You will still place a bet and be charged for any gas fees. The bet you have placed that will be sent to the smart contract is available on screen.

Random Number Generation: The random number generator used to generate random numbers is provided by Chainlink. This uses the method described on their website where a random number is generated by a group decentralised nodes. This number is then published on the Ethereum blockchain and verified by a decentralised group of nodes. The number can be proved to be randomly generated using the proof provided by Chainlink. Only if the number is valid is it used. The smart contract used here waits 3 block confirmations to ensure that the random number is confirmed and that the result of the game will not change.

Figure 8.2: Information shown to the user on the Dapp.

8.5 Transaction Information

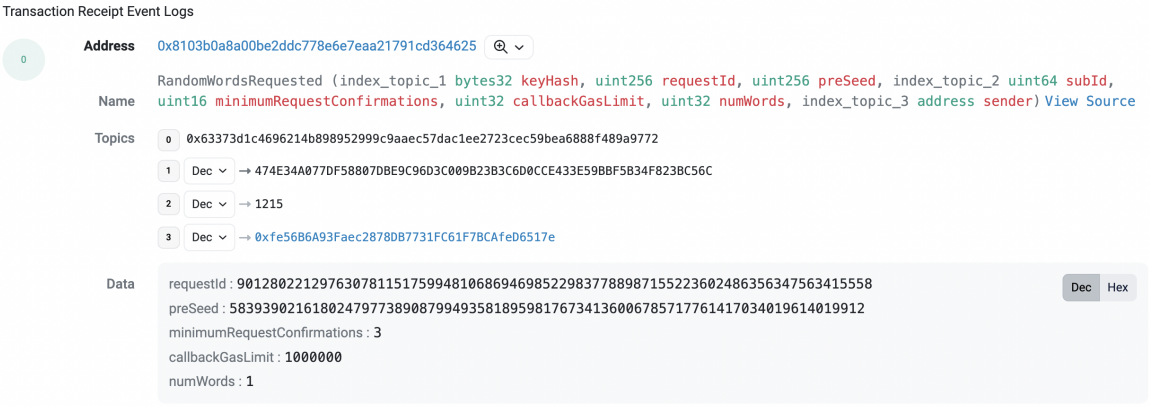

Figure 8.3: Information on Etherscan that can be viewed by the user.

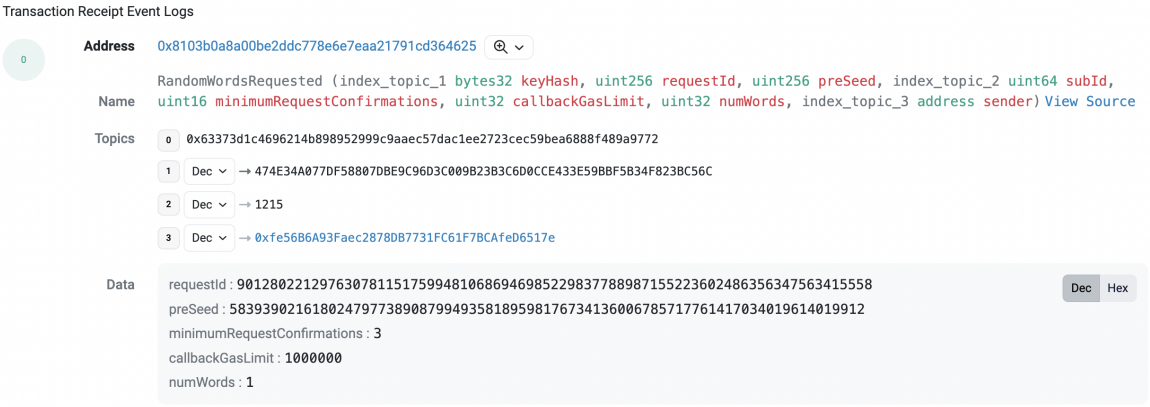

Figure 8.4: Information on Etherscan that can be viewed by the user.

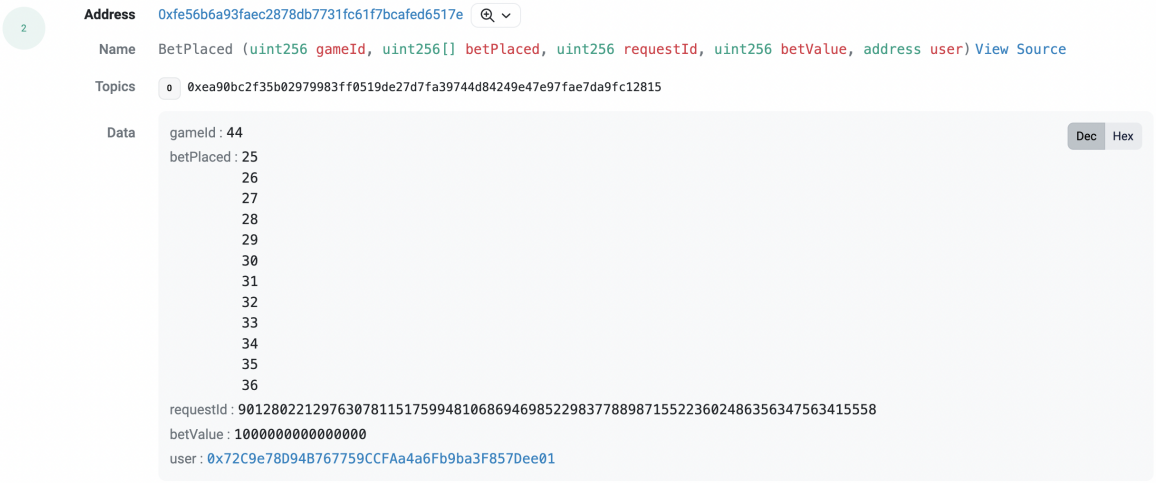

Figure 8.5: Information on Etherscan that can be viewed by the user.

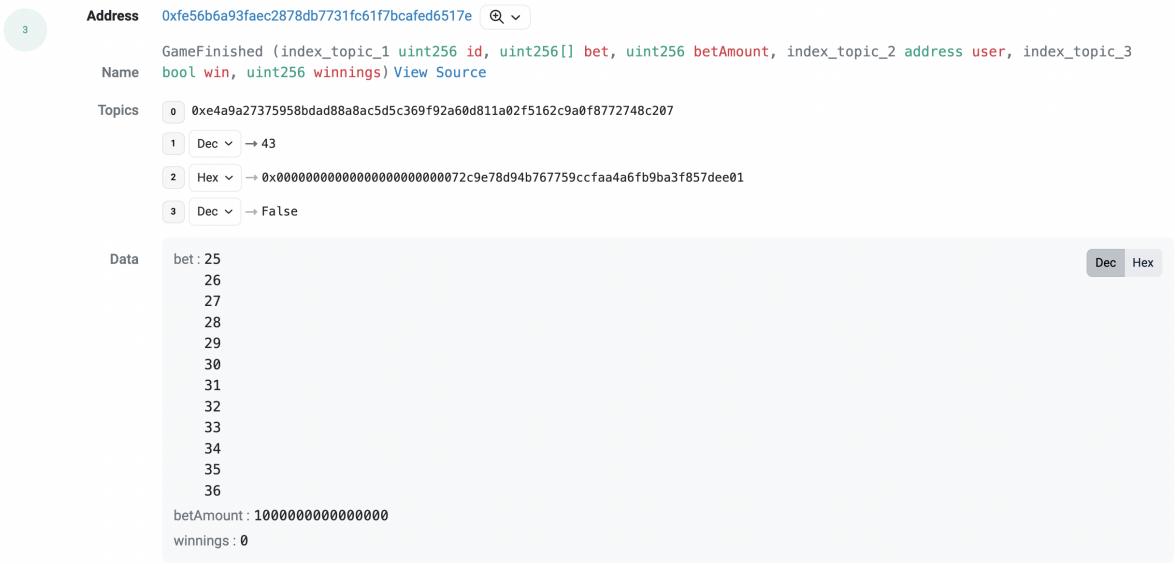

Figure 8.6: Information on Etherscan that can be viewed by the user.

8.6 Smart Contract Code

8.6.1 Roulette

```
pragma solidity ^0.8.11;

import './SafeMath.sol';
import "../node_modules/@chainlink/contracts/src/v0.8/interfaces/
    VRFCoordinatorV2Interface.sol";
import "../node_modules/@chainlink/contracts/src/v0.8/VRFConsumerBaseV2.sol";

contract Roulette is VRFConsumerBaseV2{

    address public casinoAddress; //only this address can acept and end games

    /// @dev Min value user needs to deposit for creating game session.
    uint128 public minStake;

    /// @dev Max value user can deposit for creating game session.
    uint128 public maxStake;

    uint256 public gameId;

    address payable[] public approvedAddresses;

    uint64 subscriptionId = 1215;
    bytes32 constant keyHash = 0
    x474e34a077df58807dbe9c96d3c009b23b3c6d0cce433e59bbf5b34f823bc56c; //keyhash
    VRFCoordinatorV2Interface COORDINATOR;

    uint32 constant callbackGasLimit = 1 * 10 ** 6;// how much gas they are willing
    to spend for you (cus they call function in your smart contract to send you the
    result)
    uint32 constant numWords = 1; //number of random numbers generated
    uint16 constant requestConfirmations = 3; //how many blocks need to be confirmed
     (So there can be no manipulations)

    mapping (uint256 => Game) public Games;
    mapping (uint256 => LinkRequest) public LinkRequests;

    constructor ()
    VRFConsumerBaseV2(0x8103B0A8A00be2DDC778e6e7eaa21791Cd364625)
    {
        COORDINATOR = VRFCoordinatorV2Interface(0
    x8103B0A8A00be2DDC778e6e7eaa21791Cd364625);
        casinoAddress = payable(msg.sender);
        gameId = 0;

        minStake = 1 * 10 ** 13; //0.00001 ETH (0.01862 USD as of 8 April)
        maxStake = 1 * 10 ** 18; //equivalent to 1 eth (1,862 USD as of 8 April)
```

```
41      }
42
43      struct LinkRequest {
44          uint256 requestId;
45          bool exists;
46          bool requestFulfilled;
47          uint256 randomWord;
48      }
49
50      struct Game {
51          uint256 gameId;
52          uint256[] bet;
53          uint256 betAmount;
54          address payable user;
55          bool win;
56          uint256 winnings;
57          bool gameDone;
58          uint256 requestId;
59          uint256 gameResult;
60      }
61      //may wantr to use indexed for gameFinished
62      event BetPlaced(uint256 gameId, uint256[] betPlaced, uint256 requestId, uint256
        betValue, address payable user);
63      event RequestSent(uint256 indexed requestId);
64      event RequestStored(uint256 indexed requestId, uint256 indexed randomWords);
65      event GameFinished(uint256 indexed id, uint256[] bet, uint256 betAmount, address
         payable indexed user, bool indexed win, uint256 winnings);
66      event Recieved(address indexed depositor,uint256 indexed amount);
67      event Withdrawn(address indexed casino, uint256 indexed amount);
68      event OwnershipTransferredSuccessfully(address indexed previousOwner, address
        indexed newOwner);
69      event RegisterUser(address indexed user);
70      event VerifiedAddress(address recoveredAddress);
71
72      function placeBet(uint256[] memory _numbersSelected) public payable
        onlyValidUserStake(msg.value) onlyRegistered(msg.sender) onlyValidHouseFunds(msg
        .value){
73          require (msg.sender.balance > msg.value, "Not enough money in wallet to
        place bet");
74          require (_numbersSelected.length < 7 && _numbersSelected.length > 0 ||
        _numbersSelected.length == 12 || _numbersSelected.length == 18, "Need to make a
        valid bet");
75
76          uint256 requestId = COORDINATOR.requestRandomWords(keyHash, subscriptionId,
        requestConfirmations, callbackGasLimit, numWords); //will revert if subscription
         is not set and funded
77          LinkRequests[requestId] = LinkRequest(requestId, true, false, 0);
78          emit RequestSent(requestId);
79
80          gameId = SafeMath.add(gameId, 1);
```

```
81          Games[gameId] = Game(gameId, _numbersSelected, msg.value, payable(msg.sender
    ), false, 0, false, requestId, 0);
82          emit BetPlaced(gameId, _numbersSelected, requestId, msg.value, payable (msg.
    sender));
83      }
84
85      function fulfillRandomWords(uint256 _requestId, uint256[] memory _randomWords)
    internal override {
86          require (LinkRequests[_requestId].exists, "Request was not found"); //checks
     whether a request has actually been made
87          LinkRequests[_requestId].randomWord = _randomWords[0];
88          LinkRequests[_requestId].requestFulfilled = true;
89          emit RequestStored(_requestId, LinkRequests[_requestId].randomWord);
90      }
91
92      function endGame(uint256 _gameId) public {
93          require (Games[_gameId].gameDone == false, "Game has already been finished
    .");
94          require (Games[_gameId].gameId != 0, "Game does not exist.");
95          require (LinkRequests[Games[_gameId].requestId].requestFulfilled == true, "
    Random number has not yet been generated, try again after a short wait.");
96          uint256 result = SafeMath.mod(LinkRequests[Games[_gameId].requestId].
    randomWord, 38); //This line extracts the randomly generated number that has
    been stored by the request struct
97
98          bool won = checkResult(result, Games[_gameId].bet);
99          if (won == true) {//Means that the user placed their bet successfully
100             uint256 payout = calculatePayout(Games[_gameId].bet.length);
101             payout = SafeMath.div(SafeMath.mul(payout, Games[_gameId].betAmount),
    10);
102
103             safeTransferWinnings(payable (Games[_gameId].user), payout);
104
105             Games[_gameId].win = true;
106             Games[_gameId].winnings = SafeMath.sub(payout, Games[_gameId].betAmount)
    ;
107         } else {
108         }
109         Games[_gameId].gameDone = true;
110         Games[_gameId].gameResult = result;
111         emit GameFinished(_gameId, Games[_gameId].bet, Games[_gameId].betAmount,
    Games[_gameId].user, Games[_gameId].win, Games[_gameId].winnings);
112     }
113
114     function checkResult (uint256 _result, uint256[] memory _numbersSelected)
    private pure returns (bool) {
115         for (uint256 i = 0; i< 38 && i < _numbersSelected.length ; i++) {
116             if (_numbersSelected[i] == _result) {
117                 return true;
118             }
```

```
119         }
120         return false;
121     }
122 
123     function calculatePayout(uint256 _amountOfNumbersSelected) private pure returns
    (uint256) {
124         //All the payouts are multiplied by ten so that the odds 1.5 can be returned
     (solidity cannot handle floating point numbers)
125         //The payout is later divided by ten after being multiplide with the user's
    bet
126         //Odds are dependent on the amount of numbers selected not which ones.
127         if (_amountOfNumbersSelected == 18) {
128             //If red/black/odd/even numbers are bet on (payout is 1:1)
129             return 20;
130         } else if (_amountOfNumbersSelected == 12) {
131             //If the 1st/2nd/3rd dozen or the 1st/2nd/3rd column is bet on (payuout
    is 2:1)
132             return 15;
133         } else if (_amountOfNumbersSelected < 7 && _amountOfNumbersSelected > 0) {
134             // When selecting individual numbers up to 6 can be selected with the
    odds being (36/# numbers selected) - 1
135             uint256 payout = SafeMath.div(36, _amountOfNumbersSelected);
136             payout = SafeMath.sub(payout, 1);
137             return SafeMath.mul(payout, 10);
138         }
139         return 0;
140     }
141 
142     //Sends money from the casino to the user
143     function safeTransferWinnings(address payable _userAddress, uint256 _userPayout)
     private {
144         require (_userPayout > 0, "The user did not win any money");
145         require (casinoAddress.balance > _userPayout, "Casino does not have enough
    funds to reward user after successful bet");
146         (bool sent,) = _userAddress.call{value: _userPayout}("");
147         require(sent, "Casino failed to send Ether");
148         }
149 
150     function addFunds() public payable {
151         emit Recieved(msg.sender, msg.value);
152         //If users pay to the wrong function, consider it a donation.
153     }
154 
155     function withdrawFundsWei(uint256 _withdrawAmount) public onlyCasino {
156         require (_withdrawAmount <= address(this).balance, "Attempting to withdraw
    more funds than casino has available");
157         (bool sent,) = casinoAddress.call{value: _withdrawAmount}("");
158         require(sent, "Failed to withdraw ether");
159 
160         emit Withdrawn(msg.sender, _withdrawAmount);
```

```
    }

    function withdrawAllFunds() public onlyCasino {
        require (0 < address(this).balance, "Casino has no funds to withdraw");
        uint _withdrawAmount = address(this).balance;
        (bool sent,) = casinoAddress.call{value: address(this).balance}("");
        require(sent, "Failed to withdraw ether");
        emit Withdrawn(msg.sender, _withdrawAmount);
    }

    function registerUser(address payable _address, bytes32 _messageHash, uint8 _v,
    bytes32 _r, bytes32 _s) public onlyDapp(_messageHash, _v, _r, _s) {
        approvedAddresses.push(_address);
        emit RegisterUser(_address);
    }

    function unregisterUser(address payable _address) public onlyCasino{
        for (uint i; i < approvedAddresses.length; i++) {
            if (_address == approvedAddresses[i]) {
                approvedAddresses[i] = approvedAddresses[approvedAddresses.length -
    1];
                approvedAddresses.pop();
                //no break in case there are multiple instances of a single address
            }
        }
    }

    function verifyUser(address _address) private view returns (bool) {
        for (uint i; i < approvedAddresses.length; i++) {
            if (_address == approvedAddresses[i]) {
                return true;
            }
        }
        return false;
    }

    function getApprovedUsers() public view returns( address payable[] memory) {
        return approvedAddresses;
    }

    function changeOwner(address payable _newOwner) public onlyCasino {
        require(_newOwner != address(0), "Need to transfer ownership to a valid
    address");
        emit OwnershipTransferredSuccessfully(casinoAddress, _newOwner);
        casinoAddress = _newOwner;
    }

    modifier onlyDapp(bytes32 _messageHash, uint8 _v, bytes32 _r, bytes32 _s) {
    address recoveredAddress = ecrecover(_messageHash, _v, _r, _s);
    require(recoveredAddress == casinoAddress, "Only the Dapp can register a user");
```

```
        _;
    }
    //ensures only 1% of house funds can be bet
    modifier onlyValidHouseFunds (uint256 _userBet) {
        require (SafeMath.mul(_userBet, 100) <= address(this).balance, "Casino does
    not have enough funds");
        _;
    }
    modifier onlyRegistered(address _address) {
        require (verifyUser(_address) == true, "User is not verified");
        _;
    }
    modifier onlyValidUserStake (uint256 _userBet) {
        require (_userBet >= minStake, "Stake is below the minimum stake");
        require (_userBet <= maxStake, "Stake is above the maximum stake");
        _;
    }
    modifier onlyCasino() {
        require(msg.sender == casinoAddress, "Address is not the casino");
        _;
    }
}
```

8.6.2 SafeMath

```
// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v4.6.0) (utils/math/SafeMath.sol)

pragma solidity ^0.8.11;

// CAUTION
// This version of SafeMath should only be used with Solidity 0.8 or later,
// because it relies on the compiler's built in overflow checks.

/**
 * @dev Wrappers over Solidity's arithmetic operations.
 *
 * NOTE: `SafeMath` is generally not needed starting with Solidity 0.8, since the
    compiler
 * now has built in overflow checking.
 */
library SafeMath {
    /**
     * @dev Returns the addition of two unsigned integers, with an overflow flag.
     *
     * _Available since v3.4._
     */
    function tryAdd(uint256 a, uint256 b) internal pure returns (bool, uint256) {
        unchecked {
            uint256 c = a + b;
            if (c < a) return (false, 0);
```

```
            return (true, c);
        }
    }

    /**
     * @dev Returns the subtraction of two unsigned integers, with an overflow flag.
     *
     * _Available since v3.4._
     */
    function trySub(uint256 a, uint256 b) internal pure returns (bool, uint256) {
        unchecked {
            if (b > a) return (false, 0);
            return (true, a - b);
        }
    }

    /**
     * @dev Returns the multiplication of two unsigned integers, with an overflow flag.
     *
     * _Available since v3.4._
     */
    function tryMul(uint256 a, uint256 b) internal pure returns (bool, uint256) {
        unchecked {
            // Gas optimization: this is cheaper than requiring 'a' not being zero, but the
            // benefit is lost if 'b' is also tested.
            // See: https://github.com/OpenZeppelin/openzeppelin-contracts/pull/522
            if (a == 0) return (true, 0);
            uint256 c = a * b;
            if (c / a != b) return (false, 0);
            return (true, c);
        }
    }

    /**
     * @dev Returns the division of two unsigned integers, with a division by zero flag.
     *
     * _Available since v3.4._
     */
    function tryDiv(uint256 a, uint256 b) internal pure returns (bool, uint256) {
        unchecked {
            if (b == 0) return (false, 0);
            return (true, a / b);
        }
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers, with a division
```

```
     by zero flag.
     *
     * _Available since v3.4._
     */
    function tryMod(uint256 a, uint256 b) internal pure returns (bool, uint256) {
        unchecked {
            if (b == 0) return (false, 0);
            return (true, a % b);
        }
    }

    /**
     * @dev Returns the addition of two unsigned integers, reverting on
     * overflow.
     *
     * Counterpart to Solidity's `+` operator.
     *
     * Requirements:
     *
     * - Addition cannot overflow.
     */
    function add(uint256 a, uint256 b) internal pure returns (uint256) {
        return a + b;
    }

    /**
     * @dev Returns the subtraction of two unsigned integers, reverting on
     * overflow (when the result is negative).
     *
     * Counterpart to Solidity's `-` operator.
     *
     * Requirements:
     *
     * - Subtraction cannot overflow.
     */
    function sub(uint256 a, uint256 b) internal pure returns (uint256) {
        return a - b;
    }

    /**
     * @dev Returns the multiplication of two unsigned integers, reverting on
     * overflow.
     *
     * Counterpart to Solidity's `*` operator.
     *
     * Requirements:
     *
     * - Multiplication cannot overflow.
     */
    function mul(uint256 a, uint256 b) internal pure returns (uint256) {
```

```
        return a * b;
    }

    /**
     * @dev Returns the integer division of two unsigned integers, reverting on
     * division by zero. The result is rounded towards zero.
     *
     * Counterpart to Solidity's `/` operator.
     *
     * Requirements:
     *
     * - The divisor cannot be zero.
     */
    function div(uint256 a, uint256 b) internal pure returns (uint256) {
        return a / b;
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned
    integer modulo),
     * reverting when dividing by zero.
     *
     * Counterpart to Solidity's `%` operator. This function uses a `revert`
     * opcode (which leaves remaining gas untouched) while Solidity uses an
     * invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     *
     * - The divisor cannot be zero.
     */
    function mod(uint256 a, uint256 b) internal pure returns (uint256) {
        return a % b;
    }

    /**
     * @dev Returns the subtraction of two unsigned integers, reverting with custom
    message on
     * overflow (when the result is negative).
     *
     * CAUTION: This function is deprecated because it requires allocating memory
    for the error
     * message unnecessarily. For custom revert reasons use {trySub}.
     *
     * Counterpart to Solidity's `-` operator.
     *
     * Requirements:
     *
     * - Subtraction cannot overflow.
     */
    function sub(uint256 a, uint256 b, string memory errorMessage) internal pure
```

```
    returns (uint256) {
        unchecked {
            require(b <= a, errorMessage);
            return a - b;
        }
    }

    /**
     * @dev Returns the integer division of two unsigned integers, reverting with custom message on
     * division by zero. The result is rounded towards zero.
     *
     * Counterpart to Solidity's `/` operator. Note: this function uses a
     * `revert` opcode (which leaves remaining gas untouched) while Solidity
     * uses an invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     *
     * - The divisor cannot be zero.
     */
    function div(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        unchecked {
            require(b > 0, errorMessage);
            return a / b;
        }
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
     * reverting with custom message when dividing by zero.
     *
     * CAUTION: This function is deprecated because it requires allocating memory for the error
     * message unnecessarily. For custom revert reasons use {tryMod}.
     *
     * Counterpart to Solidity's `%` operator. This function uses a `revert`
     * opcode (which leaves remaining gas untouched) while Solidity uses an
     * invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     *
     * - The divisor cannot be zero.
     */
    function mod(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        unchecked {
            require(b > 0, errorMessage);
            return a % b;
```

```
        }
    }
}
```

8.6.3 VRFConsumerBaseV2

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.4;

/** ****************************************************************************
 * @notice Interface for contracts using VRF randomness
 * *****************************************************************************
 * @dev PURPOSE
 *
 * @dev Reggie the Random Oracle (not his real job) wants to provide randomness
 * @dev to Vera the verifier in such a way that Vera can be sure he's not
 * @dev making his output up to suit himself. Reggie provides Vera a public key
 * @dev to which he knows the secret key. Each time Vera provides a seed to
 * @dev Reggie, he gives back a value which is computed completely
 * @dev deterministically from the seed and the secret key.
 *
 * @dev Reggie provides a proof by which Vera can verify that the output was
 * @dev correctly computed once Reggie tells it to her, but without that proof,
 * @dev the output is indistinguishable to her from a uniform random sample
 * @dev from the output space.
 *
 * @dev The purpose of this contract is to make it easy for unrelated contracts
 * @dev to talk to Vera the verifier about the work Reggie is doing, to provide
 * @dev simple access to a verifiable source of randomness. It ensures 2 things:
 * @dev 1. The fulfillment came from the VRFCoordinator
 * @dev 2. The consumer contract implements fulfillRandomWords.
 * *****************************************************************************
 * @dev USAGE
 *
 * @dev Calling contracts must inherit from VRFConsumerBase, and can
 * @dev initialize VRFConsumerBase's attributes in their constructor as
 * @dev shown:
 *
 * @dev   contract VRFConsumer {
 * @dev     constructor(<other arguments>, address _vrfCoordinator, address _link)
 * @dev       VRFConsumerBase(_vrfCoordinator) public {
 * @dev         <initialization with other arguments goes here>
 * @dev       }
 * @dev   }
 *
 * @dev The oracle will have given you an ID for the VRF keypair they have
 * @dev committed to (let's call it keyHash). Create subscription, fund it
 * @dev and your consumer contract as a consumer of it (see VRFCoordinatorInterface
 * @dev subscription management functions).
 * @dev Call requestRandomWords(keyHash, subId, minimumRequestConfirmations,
 * @dev callbackGasLimit, numWords),
```

```
46  * @dev see (VRFCoordinatorInterface for a description of the arguments).
47  *
48  * @dev Once the VRFCoordinator has received and validated the oracle's response
49  * @dev to your request, it will call your contract's fulfillRandomWords method.
50  *
51  * @dev The randomness argument to fulfillRandomWords is a set of random words
52  * @dev generated from your requestId and the blockHash of the request.
53  *
54  * @dev If your contract could have concurrent requests open, you can use the
55  * @dev requestId returned from requestRandomWords to track which response is
        associated
56  * @dev with which randomness request.
57  * @dev See "SECURITY CONSIDERATIONS" for principles to keep in mind,
58  * @dev if your contract could have multiple requests in flight simultaneously.
59  *
60  * @dev Colliding `requestId`s are cryptographically impossible as long as seeds
61  * @dev differ.
62  *
63  * ****************************************************************************
64  * @dev SECURITY CONSIDERATIONS
65  *
66  * @dev A method with the ability to call your fulfillRandomness method directly
67  * @dev could spoof a VRF response with any random value, so it's critical that
68  * @dev it cannot be directly called by anything other than this base contract
69  * @dev (specifically, by the VRFConsumerBase.rawFulfillRandomness method).
70  *
71  * @dev For your users to trust that your contract's random behavior is free
72  * @dev from malicious interference, it's best if you can write it so that all
73  * @dev behaviors implied by a VRF response are executed *during* your
74  * @dev fulfillRandomness method. If your contract must store the response (or
75  * @dev anything derived from it) and use it later, you must ensure that any
76  * @dev user-significant behavior which depends on that stored value cannot be
77  * @dev manipulated by a subsequent VRF request.
78  *
79  * @dev Similarly, both miners and the VRF oracle itself have some influence
80  * @dev over the order in which VRF responses appear on the blockchain, so if
81  * @dev your contract could have multiple VRF requests in flight simultaneously,
82  * @dev you must ensure that the order in which the VRF responses arrive cannot
83  * @dev be used to manipulate your contract's user-significant behavior.
84  *
85  * @dev Since the block hash of the block which contains the requestRandomness
86  * @dev call is mixed into the input to the VRF *last*, a sufficiently powerful
87  * @dev miner could, in principle, fork the blockchain to evict the block
88  * @dev containing the request, forcing the request to be included in a
89  * @dev different block with a different hash, and therefore a different input
90  * @dev to the VRF. However, such an attack would incur a substantial economic
91  * @dev cost. This cost scales with the number of blocks the VRF oracle waits
92  * @dev until it calls responds to a request. It is for this reason that
93  * @dev that you can signal to an oracle you'd like them to wait longer before
94  * @dev responding to the request (however this is not enforced in the contract
```

```
 * @dev and so remains effective only in the case of unmodified oracle software).
 */
abstract contract VRFConsumerBaseV2 {
  error OnlyCoordinatorCanFulfill(address have, address want);
  address private immutable vrfCoordinator;

  /**
   * @param _vrfCoordinator address of VRFCoordinator contract
   */
  constructor(address _vrfCoordinator) {
    vrfCoordinator = _vrfCoordinator;
  }

  /**
   * @notice fulfillRandomness handles the VRF response. Your contract must
   * @notice implement it. See "SECURITY CONSIDERATIONS" above for important
   * @notice principles to keep in mind when implementing your fulfillRandomness
   * @notice method.
   *
   * @dev VRFConsumerBaseV2 expects its subcontracts to have a method with this
   * @dev signature, and will call it once it has verified the proof
   * @dev associated with the randomness. (It is triggered via a call to
   * @dev rawFulfillRandomness, below.)
   *
   * @param requestId The Id initially returned by requestRandomness
   * @param randomWords the VRF output expanded to the requested number of words
   */
  function fulfillRandomWords(uint256 requestId, uint256[] memory randomWords)
    internal virtual;

  // rawFulfillRandomness is called by VRFCoordinator when it receives a valid VRF
  // proof. rawFulfillRandomness then calls fulfillRandomness, after validating
  // the origin of the call
  function rawFulfillRandomWords(uint256 requestId, uint256[] memory randomWords)
    external {
    if (msg.sender != vrfCoordinator) {
      revert OnlyCoordinatorCanFulfill(msg.sender, vrfCoordinator);
    }
    fulfillRandomWords(requestId, randomWords);
  }
}
```

8.6.4 VRFCoordinatorV2Interface

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

interface VRFCoordinatorV2Interface {
  /**
   * @notice Get configuration relevant for making requests
   * @return minimumRequestConfirmations global min for request confirmations
```

```
8    * @return maxGasLimit global max for request gas limit
9    * @return s_provingKeyHashes list of registered key hashes
10   */
11  function getRequestConfig()
12    external
13    view
14    returns (
15      uint16,
16      uint32,
17      bytes32[] memory
18    );
19
20  /**
21   * @notice Request a set of random words.
22   * @param keyHash - Corresponds to a particular oracle job which uses
23   * that key for generating the VRF proof. Different keyHash's have different gas
      price
24   * ceilings, so you can select a specific one to bound your maximum per request
      cost.
25   * @param subId  - The ID of the VRF subscription. Must be funded
26   * with the minimum subscription balance required for the selected keyHash.
27   * @param minimumRequestConfirmations - How many blocks you'd like the
28   * oracle to wait before responding to the request. See SECURITY CONSIDERATIONS
29   * for why you may want to request more. The acceptable range is
30   * [minimumRequestBlockConfirmations, 200].
31   * @param callbackGasLimit - How much gas you'd like to receive in your
32   * fulfillRandomWords callback. Note that gasleft() inside fulfillRandomWords
33   * may be slightly less than this amount because of gas used calling the function
34   * (argument decoding etc.), so you may need to request slightly more than you
      expect
35   * to have inside fulfillRandomWords. The acceptable range is
36   * [0, maxGasLimit]
37   * @param numWords - The number of uint256 random values you'd like to receive
38   * in your fulfillRandomWords callback. Note these numbers are expanded in a
39   * secure way by the VRFCoordinator from a single random value supplied by the
      oracle.
40   * @return requestId - A unique identifier of the request. Can be used to match
41   * a request to a response in fulfillRandomWords.
42   */
43  function requestRandomWords(
44    bytes32 keyHash,
45    uint64 subId,
46    uint16 minimumRequestConfirmations,
47    uint32 callbackGasLimit,
48    uint32 numWords
49  ) external returns (uint256 requestId);
50
51  /**
52   * @notice Create a VRF subscription.
53   * @return subId - A unique subscription id.
```

```
54    * @dev You can manage the consumer set dynamically with addConsumer/
       removeConsumer.
55    * @dev Note to fund the subscription, use transferAndCall. For example
56    * @dev  LINKTOKEN.transferAndCall(
57    * @dev    address(COORDINATOR),
58    * @dev    amount,
59    * @dev    abi.encode(subId));
60    */
61   function createSubscription() external returns (uint64 subId);
62
63   /**
64    * @notice Get a VRF subscription.
65    * @param subId - ID of the subscription
66    * @return balance - LINK balance of the subscription in juels.
67    * @return reqCount - number of requests for this subscription, determines fee
       tier.
68    * @return owner - owner of the subscription.
69    * @return consumers - list of consumer address which are able to use this
       subscription.
70    */
71   function getSubscription(uint64 subId)
72     external
73     view
74     returns (
75       uint96 balance,
76       uint64 reqCount,
77       address owner,
78       address[] memory consumers
79     );
80
81   /**
82    * @notice Request subscription owner transfer.
83    * @param subId - ID of the subscription
84    * @param newOwner - proposed new owner of the subscription
85    */
86   function requestSubscriptionOwnerTransfer(uint64 subId, address newOwner) external
       ;
87
88   /**
89    * @notice Request subscription owner transfer.
90    * @param subId - ID of the subscription
91    * @dev will revert if original owner of subId has
92    * not requested that msg.sender become the new owner.
93    */
94   function acceptSubscriptionOwnerTransfer(uint64 subId) external;
95
96   /**
97    * @notice Add a consumer to a VRF subscription.
98    * @param subId - ID of the subscription
99    * @param consumer - New consumer which can use the subscription
```

```
   */
  function addConsumer(uint64 subId, address consumer) external;

  /**
   * @notice Remove a consumer from a VRF subscription.
   * @param subId - ID of the subscription
   * @param consumer - Consumer to remove from the subscription
   */
  function removeConsumer(uint64 subId, address consumer) external;

  /**
   * @notice Cancel a subscription
   * @param subId - ID of the subscription
   * @param to - Where to send the remaining LINK to
   */
  function cancelSubscription(uint64 subId, address to) external;

  /*
   * @notice Check to see if there exists a request commitment consumers
   * for all consumers and keyhashes for a given sub.
   * @param subId - ID of the subscription
   * @return true if there exists at least one unfulfilled request for the
    subscription, false
   * otherwise.
   */
  function pendingRequestExists(uint64 subId) external view returns (bool);
}
```

8.6.5 RouletteMock

```
pragma solidity ^0.8.11;

import './SafeMath.sol';
import "../node_modules/@chainlink/contracts/src/v0.8/interfaces/
    VRFCoordinatorV2Interface.sol";
import "../node_modules/@chainlink/contracts/src/v0.8/VRFConsumerBaseV2.sol";

contract RouletteMock is VRFConsumerBaseV2{

    address public casinoAddress; //only this address can acept and end games

    /// @dev Min value user needs to deposit for creating game session.
    uint128 public minStake;

    /// @dev Max value user can deposit for creating game session.
    uint128 public maxStake;

    uint256 public gameId;

    address payable[] public approvedAddresses;

```

```
    uint64 subscriptionId = 1215;
    bytes32 constant keyHash = 0
    x474e34a077df58807dbe9c96d3c009b23b3c6d0cce433e59bbf5b34f823bc56c; //keyhash
    VRFCoordinatorV2Interface COORDINATOR;

    uint32 constant callbackGasLimit = 1 * 10 ** 6;// how much gas they are willing
    to spend for you (cus they call function in your smart contract to send you the
    result)
    uint32 constant numWords = 1; //number of random numbers generated
    uint16 constant requestConfirmations = 3; //how many blocks need to be confirmed
     (So there can be no manipulations)

    mapping (uint256 => Game) public Games;
    mapping (uint256 => LinkRequest) public LinkRequests;

    constructor ()
    VRFConsumerBaseV2(0x8103B0A8A00be2DDC778e6e7eaa21791Cd364625)
    {
        COORDINATOR = VRFCoordinatorV2Interface(0
    x8103B0A8A00be2DDC778e6e7eaa21791Cd364625);
        casinoAddress = payable(msg.sender);
        gameId = 0;

        minStake = 1 * 10 ** 13; //0.00001 ETH (0.01862 USD as of 8 April)
        maxStake = 1 * 10 ** 18; //equivalent to 1 eth (1,862 USD as of 8 April)
    }

    struct LinkRequest {
        uint256 requestId;
        bool exists;
        bool requestFulfilled;
        uint256 randomWord;
    }
    struct Game {
        uint256 gameId;
        uint256[] bet;
        uint256 betAmount;
        address payable user;
        bool win;
        uint256 winnings;
        bool gameDone;
        uint256 requestId;
        uint256 gameResult;
    }
    //may wantr to use indexed for gameFinished
    event BetPlaced(uint256 gameId, uint256[] betPlaced, uint256 requestId, uint256
    betValue, address payable user);
    event RequestSent(uint256 indexed requestId);
    event RequestStored(uint256 indexed requestId, uint256 indexed randomWords);
    event GameFinished(uint256 indexed id, uint256[] bet, uint256 betAmount, address
```

```
    payable indexed user, bool indexed win, uint256 winnings);
65  event Recieved(address indexed depositor,uint256 indexed amount);
66  event Withdrawn(address indexed casino, uint256 indexed amount);
67  event OwnershipTransferredSuccessfully(address indexed previousOwner, address
    indexed newOwner);
68  event RegisterUser(address indexed user);
69
70  function placeBet(uint256[] memory _numbersSelected) public payable
    onlyValidUserStake(msg.value) onlyRegistered(msg.sender) onlyValidHouseFunds(msg
    .value){
71      //should freeze any more functionality instantly actually (like dont let the
     user button spam)
72      require (msg.sender.balance > msg.value, "Not enough money in wallet to
    place bet");
73      require (_numbersSelected.length < 7 && _numbersSelected.length > 0 ||
    _numbersSelected.length == 12 || _numbersSelected.length == 18, "Need to make a
    valid bet");
74      require (verifyUser(msg.sender), "User has not been registered with the
    casino");
75
76      // This line has also been added since normally the coordinator would return
     a request ID. The original line is below
77      // uint256 requestId = COORDINATOR.requestRandomWords(keyHash,
    subscriptionId, requestConfirmations, callbackGasLimit, numWords); //will revert
     if subscription is not set and funded
78      uint256 requestId = SafeMath.add(gameId, 1);
79      LinkRequests[requestId] = LinkRequest(requestId, true, false, 0);
80      emit RequestSent(requestId);
81
82      gameId = SafeMath.add(gameId, 1);
83      Games[gameId] = Game(gameId, _numbersSelected, msg.value, payable(msg.sender
    ), false, 0, false, requestId, 0);
84      emit BetPlaced(gameId, _numbersSelected, requestId, msg.value, payable (msg.
    sender));
85
86      // This part has been added, normally would call the coordinator function
    and allow that to call the fulfillRandomWords function and the next 3 lines are
    not in the contract
87      uint[] memory randomWordsMock = new uint[](1);
88      randomWordsMock[0] = uint (39);
89      fulfillRandomWords(gameId, randomWordsMock);
90  }
91
92  function fulfillRandomWords(uint256 _requestId, uint256[] memory _randomWords)
    internal override {
93      require (LinkRequests[_requestId].exists, "Request was not found"); //checks
     whether a request has actually been made
94      LinkRequests[_requestId].randomWord = _randomWords[0];
95      LinkRequests[_requestId].requestFulfilled = true;
96      emit RequestStored(_requestId, LinkRequests[_requestId].randomWord);
```

```
97      }
98
99      function endGame(uint256 _gameId) public {
100         require (Games[_gameId].gameDone == false, "Game has already been finished
    .");
101         require (Games[_gameId].gameId != 0, "Game does not exist.");
102         require (LinkRequests[Games[_gameId].requestId].requestFulfilled == true, "
    Random number has not yet been generated, try again after a short wait.");
103         uint256 result = SafeMath.mod(LinkRequests[Games[_gameId].requestId].
    randomWord, 38); //This line extracts the randomly generated number that has
    been stored by the request struct
104
105         bool won = checkResult(result, Games[_gameId].bet);
106         if (won == true) {//Means that the user placed their bet successfully
107             uint256 payout = calculatePayout(Games[_gameId].bet.length);
108             payout = SafeMath.div(SafeMath.mul(payout, Games[_gameId].betAmount),
    10);
109
110             safeTransferWinnings(payable (Games[_gameId].user), payout);
111
112             Games[_gameId].win = true;
113             Games[_gameId].winnings = SafeMath.sub(payout, Games[_gameId].betAmount)
    ;
114         } else {
115         }
116         Games[_gameId].gameDone = true;
117         Games[_gameId].gameResult = result;
118         emit GameFinished(_gameId, Games[_gameId].bet, Games[_gameId].betAmount,
    Games[_gameId].user, Games[_gameId].win, Games[_gameId].winnings);
119     }
120
121     function checkResult (uint256 _result, uint256[] memory _numbersSelected)
    private pure returns (bool) {
122         for (uint256 i = 0; i< 38 && i < _numbersSelected.length ; i++) {
123             if (_numbersSelected[i] == _result) {
124                 return true;
125             }
126         }
127         return false;
128     }
129
130     function calculatePayout(uint256 _amountOfNumbersSelected) private pure returns
    (uint256) {
131         //All the payouts are multiplied by ten so that the odds 1.5 can be returned
     (solidity cannot handle floating point numbers)
132         //The payout is later divided by ten after being multiplide with the user's
    bet
133         //Odds are dependent on the amount of numbers selected not which ones.
134         if (_amountOfNumbersSelected == 18) {
135             //If red/black/odd/even numbers are bet on (payout is 1:1)
```

```
136             return 20;
137         } else if (_amountOfNumbersSelected == 12) {
138             //If the 1st/2nd/3rd dozen or the 1st/2nd/3rd column is bet on (payuout
    is 2:1)
139             return 15;
140         } else if (_amountOfNumbersSelected < 7 && _amountOfNumbersSelected > 0) {
141             // When selecting individual numbers up to 6 can be selected with the
    odds being (36/# numbers selected) - 1
142             uint256 payout = SafeMath.div(36, _amountOfNumbersSelected);
143             payout = SafeMath.sub(payout, 1);
144             return SafeMath.mul(payout, 10);
145         }
146         return 0;
147     }
148 
149     //Sends money from the casino to the user
150     function safeTransferWinnings(address payable _userAddress, uint256 _userPayout)
     private {
151         require (_userPayout > 0, "The user did not win any money");
152         require (casinoAddress.balance > _userPayout, "Casino does not have enough
    funds to reward user after successful bet");
153         (bool sent,) = _userAddress.call{value: _userPayout}("");
154         require(sent, "Casino failed to send Ether");
155         }
156 
157     function addFunds() public payable {
158         emit Recieved(msg.sender, msg.value);
159         //If users pay to the wrong function, consider it a donation.
160     }
161 
162     function withdrawFundsWei(uint256 _withdrawAmount) public onlyCasino {
163         require (_withdrawAmount <= address(this).balance, "Attempting to withdraw
    more funds than casino has available");
164         (bool sent,) = casinoAddress.call{value: _withdrawAmount}("");
165         require(sent, "Failed to withdraw ether");
166 
167         emit Withdrawn(msg.sender, _withdrawAmount);
168     }
169 
170     function withdrawAllFunds() public onlyCasino {
171         require (0 < address(this).balance, "Casino has no funds to withdraw");
172         uint _withdrawAmount = address(this).balance;
173         (bool sent,) = casinoAddress.call{value: address(this).balance}("");
174         require(sent, "Failed to withdraw ether");
175         emit Withdrawn(msg.sender, _withdrawAmount);
176     }
177 
178     function registerUser(address payable _address, bytes32 _messageHash, uint8 _v,
    bytes32 _r, bytes32 _s) public onlyDapp(_messageHash, _v, _r, _s) {
179         approvedAddresses.push(_address);
```

```
180         emit RegisterUser(_address);
181     }
182 
183     function unregisterUser(address payable _address) public onlyCasino {
184         for (uint i; i < approvedAddresses.length; i++) {
185             if (_address == approvedAddresses[i]) {
186                 approvedAddresses[i] = approvedAddresses[approvedAddresses.length -
    1];
187                 approvedAddresses.pop();
188                 //no break in case there are multiple instances of a single address
189             }
190         }
191     }
192 
193     function verifyUser(address _address) private view returns (bool) {
194         for (uint i; i < approvedAddresses.length; i++) {
195             if (_address == approvedAddresses[i]) {
196                 return true;
197             }
198         }
199         return false;
200     }
201 
202     function getApprovedUsers() public view returns( address payable[] memory) {
203         return approvedAddresses;
204     }
205 
206     function changeOwner(address payable _newOwner) public onlyCasino {
207         require(_newOwner != address(0), "Need to transfer ownership to a valid
    address");
208         emit OwnershipTransferredSuccessfully(casinoAddress, _newOwner);
209         casinoAddress = _newOwner;
210     }
211 
212     modifier onlyDapp(bytes32 _messageHash, uint8 _v, bytes32 _r, bytes32 _s) {
213     address recoveredAddress = ecrecover(_messageHash, _v, _r, _s);
214     require(recoveredAddress == casinoAddress, "Only the Dapp can register a user");
215         _;
216     }
217 
218     modifier onlyValidHouseFunds (uint256 _userBet) {
219         require (SafeMath.mul(_userBet, 100) <= address(this).balance, "Casino does
    not have enough funds");
220         _;
221     }
222 
223     modifier onlyRegistered(address _address) {
224         require (verifyUser(_address) == true, "User is not verified");
225         _;
226     }
```

```
227
228     modifier onlyValidUserStake (uint256 _userBet) {
229         require (_userBet >= minStake, "Stake is below the minimum stake");
230         require (_userBet <= maxStake, "Stake is above the maximum stake");
231         _;
232     }
233
234     modifier onlyCasino() {
235         require(msg.sender == casinoAddress, "Address is not the casino");
236         _;
237     }
238
239 }
```