Texas Hold'Em Poker on the Ethereum Blockchain

Principle of Decentralised Ledgers
Imperial College London

Louis de Beaumont

March 5, 2018

1 Introduction

To try and gain an in depth understanding of the Ethereum blockchain and Solidity, I chose to implement the popular game Texas Hold'Em Poker [6]. Due to time limitations, the difficulty of the project, and previous experience with front-end technologies, I chose to not implement a front-end for the game and to instead spend my time learning Solidity.
The flow gameplay of Texas Hold'Em is depicted in Figure 1.
When there are at least two players at the table the game may be started. The player to the left of the dealer pays the small blind, and the player to the left of him plays the big blind (two required monetary deposits to ensure there is some risk every round). The cards are then dealt, and betting ensues. When every player has either met the maximum bet, or folded (sacrificing their current stake in the game), some number of cards are placed on the table. At first three cards are played (called the "flop"), then one card is added (called the "turn"), then a final card is added (called the "river"). The flop, turn, and river cards are only played once each round of betting has concluded. After the river is played, there is a final round of betting, after which all players cards are revealed and the pot (the communal sum of bets) is distributed to the winner(s). There has been a lot of interest in implementing gambling games, including poker, on the blockchain. The decentralised nature of the blockchain removes the need for trust to be placed on the casino. The casino is responsible for storing all players hands, storing all players bets, performing correct game logic, and for paying the winners. In one case, online casino website Absolute Poker went bankrupt [7], leaving players currently owed by the casino with no payouts. By implementing the application on the blockchain, there is the potential to remove the requirement for trust in all of these features [1].

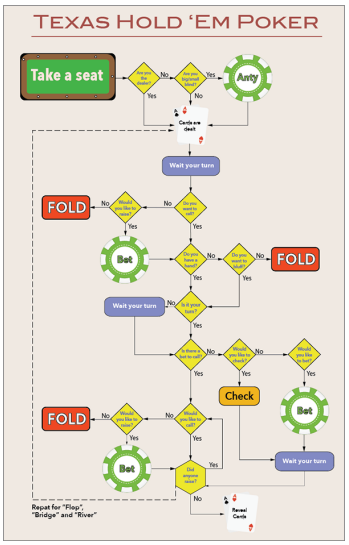

Figure 1: The gameplay flow of Texas Hold'Em Poker

2 Implementation

2.1 Game Logic

2.1.1 Joining and Starting the Game

One player creates the contract. Then he, and any other players with the address of the contract, may call `joinGame`. Between 2 and 7 players are allowed to partake in a single game.
Once there are sufficient players in the game, the owner can call `startGame` which shuffles the deck, and assigns the current player to the one paying the small blind.

2.1.2 Making Bets

The small blind, and the player to his right, the big blind, must call `paySmallBlind` and `payBigBlind` respectively, in order. Once the blinds have been paid, the cards are dealt. The blinds are prede-

termined in the contract as:

- Small Blind: 1 finney
- Big Blind: 2 finney [1]

At every round of betting, we allow each player the chance to raise, call or fold. Raising and calling are done through the public, payable function `makeBet`. If the bet is not sufficient to call the current bet, the transaction is reverted. The `makeBet` function is implemented as follows:

```
function makeBet() public payable onlyCurrentPlayer whenPlaying whenDealt {
  uint newBet = bets[msg.sender] + msg.value;
  if(newBet < maxBet)
   revert();

  bets[msg.sender] = newBet;
  setMaxBet(bets[msg.sender], msg.sender);
  incrementCurrentPlayer();
  tryIncrementRound();
}
```

The implementation is simple and easy to follow. Function call `incrementCurrentPlayer()` moves the play to the next player round the table. `tryIncrementRound()` will check whether all players, who have not folded, have matched the maximum bet, and that all players have had the chance to raise (tracked through the variable `lastPlayerToRaise`).
We can see the logic of setting a maximum bet and setting the last player to raise in the function, `setMaxBet`:

```
function setMaxBet(uint bet, address sender) private {
  if (bet > maxBet) {
      maxBet = bet;
      lastPlayerToRaise = sender;
  }
}
```

Then, we only try to increment the round (in `tryIncrementRound`) if the following condition is met:

```
currentPlayerAddress == lastPlayerToRaise && bets[currentPlayerAddress] == maxBet
```

2.1.3 Win Conditions

If enough players folding, leaving only one remaining, that remaining player is determined the winner and the pot is paid to him. Otherwise, once all rounds of betting have completed and the river is played, the contract inspects each players hand and determines a score based on their hand. The player or players with the highest score take the winnings (equal shares of the winnings, in the case where multiple players win).
The functions that determine which player has won are expensive to run, as they involve multiple array iterations. [2] They are paid for by the last person to call or check, as it is the `makeBet` function that calls, through `tryIterateRound`, `checkWin`. A possible improvement to this would be adding a fee to join the game that is used to perform functions with communal interest (including both shuffling and checking for the winner).

[1] A possible extension of this project would be to allow the owner to set these values on creating the contract (though the big blind is always twice the small blind). Then players could check the blind values before joining a game and see if the stakes suit what they want to bet.

[2] I propose that for a more practical implementation of poker on the blockchain, a third party service (on the blockchain) would perform these calculations, utilising a table of possible win situations instead of looping over the hand and river.

2.2 User Interaction

Once the game is started, anyone can query the `getCurrentPlayers` function to receive the list of current (non-folded) players, and the index of the current player in the list.
At any point, a player can call `getMaxBet` and `getMyBet` to determine the game's max bet and their bet. This informs the player how much they need to bet, or helps them decide whether they want to bet or fold.
On their turn, the user can make a call to `makeBet` with their bet, and complete the transaction using MetaMask.

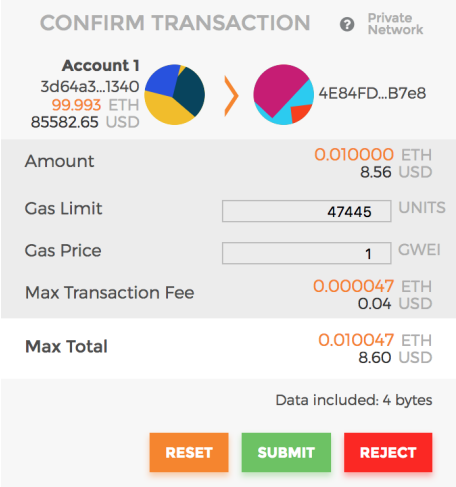

Figure 2: Making a bet using MetaMask

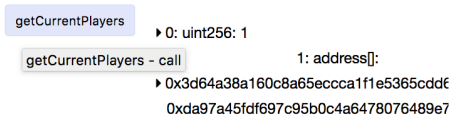

Figure 3: Inspecting the current players in the game

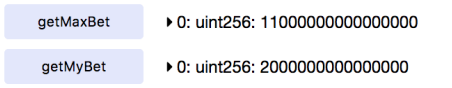

Figure 4: Inspecting the players current bet and the games max bet

2.3 Failures

Due to time limitations, a lack of Solidity programming experience, or other means, I experienced a number of failures while trying to implement this project.

2.3.1 Randomness

Randomness is a hot topic in the blockchain space. Obtaining unpredictable random numbers on the blockchain is difficult. A number of solutions exist or have been proposed [9], but I found it hard to successfully implement any one of them. I spent the most time attempting to implement a solution with Oraclize [4], which was attractive for its ease of implementation. Developing with Oraclize became difficult, as it is a paid service and requires a connection to the main network (I was only working on my test network). Should this implementation had worked, the implementation would have relied on a third party service (and a centralised component).
Decentralised poker providers, Virtue [8], achieve randomness using generators on the players local system, through a desktop client. Since I was not building a front-end, I did not have the luxury of this.
The result of failure to provide randomness (and difficulties with looping over the deck to shuffle it, a failure in (understanding) Solidity [2]) resulted in the game being played with an unshuffled deck, leaving the game predictable and therefore not suitable for play with real stakes.

2.3.2 Secrecy

In the time frame, and given the complexity of the project, I failed to implement any form of secrecy. This would have been important for encrypting values such as the deck and the players hands (and could have been solved by encrypting each card individually). However, given the lack of randomness in shuffling the deck, I did not reach a point where implementing secrecy was important. Successfully implemented features, such as betting and round iterations, required no form of secrecy.

2.3.3 Stack Limitations

Due to stack limitations in Solidity [5], implementing the algorithms to determine the winner required a number of refactors [3]. The refactors involved removing the number of local variables, to reduce the amount of information stored on the stack.
However I was unable, for unknown reasons, to implement the `hasTwoPair` function, which determines whether the player holds two distinct pairs between his hand and the table. I attempted to implement this function in a number of different ways, but at each attempt creating the contract would fail with an infinite gas cost warning.

3 Threat Model

There are a number of ways an adversary could attack our implementation of poker. Some ways are a result of failures discussed in §2.3, others are a result of the general form of implementation.

3.1 Open Information

The lack of randomness, discussed in §2.3.1, leaves our deck unshuffled. Any adversary could track the number of players, and work out what cards they hold in their hand. Furthermore the adversary could predict the river based on the number of players.
Should randomness be implemented and the deck shuffled, the lack of secrecy, discussed in §2.3.2, means an adversary could read the deck and player hands directly from the blockchain. They could then determine whether it is worth them partaking in betting. An innocent player stands to lose a lot against this kind of adversary. The adversary stands to lose, at maximum, the cost of the big blind.

3.2 Collusion

Our implementation reveals the other players in the game once the game has started. The thought behind this was to stop players attempting to dodge games with specific players they may dislike playing against. Furthermore, if a service was to be built where the player was entered into a random game, instead of joining a contract by being passed the address, it would stop players leaving games before finding a game in which their partner, another adversary, is also playing. [3]
However in our implementation, two players can discuss, outside the scope of the game, their cards. Counting cards [**?**] is a common problem in the gambling industry. A player can be asked to leave and barred from a casino if the casino suspects them of counting cards. Counting cards involves predicting odds of river cards and other players cards by tracking which cards have been played. Knowing another player's (the adversary's partner's) cards, greatly increases chances of predicting other cards yet to be shown in the game. An adversary can then make more informed betting decisions based on this information, giving him a better chance of getting involved in a winning pot, or folding from a losing pot.

3.3 Denial of Service

In current implementations of online poker contain a timeout while waiting for each players bet. When a player does not make their bet in time, if the player is the highest bidder, they simply check (and pass play to the next player). If the player is not the highest bidder, their hand is folded and their stake in the pot lost.
My implementation does not contain any timeout functionality. This means that any player can choose to not make a bid. This results in the game being frozen, as the current player is never passed on.
An adversary may choose to freeze the game if he has a losing hand, or simply wants to ensure no player receives any amount of the pot.
A timeout functionality would be necessary for an implementation which would be deployed on the main network.

[3] A cost paid on joining games would discourage adversaries iterating through multiple games until finding their desired partners - in a matchmaking environment.

References

[1] Blockchain week 2016: Virtue poker - ryan gittleson. https://www.youtube.com/watch?v=8lClBD62Mys.

[2] Commit: Add non-working pseudo random shuffle alg. https://github.com/louisdeb/dapp-poker/commit/1f6d02410a2c9f94581da96c0da833f825304988.

[3] Commit: Fix stack size issues when determining winner. https://github.com/louisdeb/dapp-poker/commit/b9c6885a1d4b45876de6c49d646cfc3568f3a4f3.

[4] Oraclize. http://www.oraclize.it/.

[5] Solidity issues: Stack too deep 267. https://github.com/ethereum/solidity/issues/267.

[6] Texas hold 'em. https://en.wikipedia.org/wiki/Texas_hold_%27em.

[7] Ub & absolute poker to file bankruptcy.

[8] Virtue Poker. Faq about virtue poker.

[9] Arseny Reutov. Predicting random numbers in ethereum smart contracts.