DezBet Yellow Paper

DezBet Team

April 30, 2022

1 Introduction

The gambling industry has been around for a long time. Those who have experienced it can vouch that at one point or another, thoughts about rigged games and unfair playing conditions were muddying up their thoughts. In situations such as this one, the virtues of on-chain implementations of monetary protocols cannot be overstated. Approaching this industry from a fresh, decentralized standpoint allows everyone to be equal. The house cannot always win, if there is no house. One could argue that we are all just addresses on the blockchain which spans across the globe. DezBet aims to provide a free-for-all, transparent and open source protocol, which will provide entertainment to anyone with a pair of private keys and an internet connection. Times are changing, and we must change with them.

2 Laying the foundation

In order to bring the platform to life, certain cornerstones had to be laid out. Those which will intertwine with the DezBet contracts to provide an immersive betting experience.

2.1 VRF Oracles

The main artery of every betting game is found within its unpredictability. Previous results must not define the future ones, and the source of randomness must act as an unbreakable backbone of the entire system. In order to achieve this, DezBet chose ChainLink on-chain oracles, a.k.a ChainLink VRF as the ultimate source of (random) truth.

2.2 Job automation

As different games enter the DezBet stage, some of them will bring about the concept of a limited time scope. As the blockchain itself does not offer event listeners or any similar mechanisms, but is rather user-interaction-based, we had to search for an external solution. One which would provide job scheduling in the form of custom time-based triggers for our own on-chain events. We found such a solution with Gelato.

3 Games

The base version of the platform will feature 3 games, ranging from short to long running. As the platform grows its user base, the game list will expand to cover anything one could come up with, from traditional Las-Vegas-like games, to new concepts, brainstormed natively on the blockchain. Three initial games are as follows: Coinflip, 1-versus-All and the Lottery.

3.1 Coinflip

Short game which will feature a 2-player battle. Based around an on-chain address pool, which can be thought of as a two slot array. Both competing addresses will queue the game by sending a designated amount of the gas token to the contract, as a stake. After the second address has joined the user pool, VRF will be queried for a random number, and a modulo of that number will determine the winner.

Game timeline:

- Address pool state: $[0x0, 0x0]$
- Address 0x123 joins the game with a designated stake
- Address pool state: $[0x123, 0x0]$
- Address 0x456 joins the game with a designated stake
- Address pool state: $[0x123, 0x456]$
- VRF is queried. It returns a random number X
- $modulo(X, 2) = Y$
- Y value determines the address pool index of the winner's address. In this case, 0 means 0x123 won, 1 means 0x456 won
- Address pool values get reset to default values
- Address pool state: $[0x0, 0x0]$

Note: Address which queries as the second one to play the game will pay for the gas costs of generating a random number using the VRF, in order to ensure that many games can be played consequently, without waiting for an external operator to trigger the winner-picking function.

3.2 1-versus-All a.k.a 1vN

Time-based game which allows any number of players to join. It is based around an unbounded address pool which stores the addresses of participants.

Each participant can enter the game by sending a designated amount of gas coins to the contract. After the timer expires, the job scheduler will call the winner-picking function, which in turn queries the VRF to get a random number. Modulo of this number will determine the winner, who takes all the collected funds within the contract, minus the platform fees.

Game timeline

- Address pool state: $[(empty)]$
- Address 0x123 joins the game with a designated stake
- Address pool state: $[0x123]$
- Many more addresses join the game ...
- Address pool state: $[0x123, 0x456, ..., 0xfff]$
- Timer expires
- Job scheduler calls $pickWinner()$
- VRF is queried. It returns a random number X
- $modulo(X, addressPool.size()) = Y$
- Y value determines the address pool index of the winner's address.
- Collected funds are send to the winner, minus the platform fees
- Address pool values get flushed
- Address pool state: $[(empty)]$

3.3 Lottery

Time-based game which allows any number of players to join. It is inspired by the real-life lottery/bingo games, where players pick a series of random numbers, which, if matched correctly against the lucky numbers, bring a grand prize to the contestant.

Game is based around picking a correct list of numbers, in the correct order. For example, if the winning numbers are $[1, 2, 3, 4]$, and an address has $[2, 1, 3, 4]$, this will not count as a win.

On-chain implementation will be based on hashing input values, using the EVM built-in k$eccak256$ hash function. This is the reason why different permutations of the same number combination count as different inputs. To increase chances of winning, the number pool will be greatly reduced in comparison to traditional bingo, as well as the lucky number array size.

Each entry hash will be stored within a mapping, which will be defined as m$apping(uint256, address[])$. If multiple addresses pick the same permutation of the same numbers, they will be stored under the same mapping key. When the winning hash has been calculated, all addresses under the corresponding key will be declared winners and split the jackpot equally.

This game does not have to have a winner. In this case, all the funds which have been collected will simply pour over into the next iteration, therefore increasing the total jackpot value.

Assuming that the amount of numbers per entry will be N, the input can be defined as

$$[x_1, x_2, ..., x_n], \text{ where each } x_k \text{ is an integer}$$

Value-to-be-stored on-chain is extracted from this by invoking

$$uint256(keccak256(abi.encodePacked(x_1, x_2, ..., x_n))).$$

This will yield identical hash values for every address with the same input array, as the array values will be bound by the size of an integer, therefore, appending multiple numbers will always yield a byte string of the same size.

After the VRF query has been performed, N random numbers will be generated, all within the number pool limits. Putting these values into a hash function will produce a winner. A simple key lookup will reveal all the addresses with a lucky combination.

Assuming that $[r_1, r_2, ..., r_n]$ is the VRF output, and defining the following:

$$uint256 hashedVRF = uint256(keccak256(abi.encodePacked(r_1, r_2, ..., r_n)))$$

Then m$apping[hashedVRF]$ will return a$ddress[]$ which contains the winning addresses

References