DICETHER: A SECURE DICE GAME 76F26CD - 2018-05-01

DICETHER

Abstract. This paper proposes a protocol for building a fair, secure and fast dice game. To achieve this the implementation is based on state channels. State channels allows to move state-altering operation normally performed on a blockchain off the blockchain. For conflict management and settlement Ethereum smart contracts are used but it should work on other blockchains providing smart contracts as well. The implementation can be tested at `dicether.com`

1. Introduction

Currently there are mainly two types of dice games implementations available. At the first one the player deposits his funds for playing at the casino. To verify the results of the bets a provably fair algorithm [2] is used. So, a user can verify the result of his bets, but the payout of the winnings is not guaranteed. The user needs to trust the operator to let him withdraw his funds and winnings.

The other one is using smart contracts [12] to implement the game logic. The main downside of implementing the whole game logic inside a smart contract is the slow gambling experience for the player. He has to wait a few minutes or even longer for every single dice roll. Furthermore, currently there is no reliable smart contract random number generating possible. So, third party services like e.g. *RANDOM.ORG* [10] and *ORACLIZE* [8] need to be used.

1.1. Driving Factors.

Dicether combines the advantages of both approaches without their disadvantages, to create a better and securer overall gambling experience. The key factors are fast gameplay and provable secure winnings payout and no third-party dependency for random number generation.

In this context fast means a bet should be resolved in under one second. Secure winnings payout implies it should always possible for Player (P) to get his winnings. In no circumstances it should be possible for the operator to withhold the winnings of P. The random number generation should only depend on Server (S) and P. Both should not know the result before the bet is finished.

1.2. Related Work.

Etheroll [5] is a full on chain dice implementation. For random number generation is uses *RANDOM.ORG* and *ORACLIZE*. To provide secure random numbers both need to be trusted. Under this assumption Etheroll is secure and provable fair. But *RANDOM.ORG* could change the random number generated to their advantage. Furthermore, they does not even need to change the random numbers, they could just buffer the random numbers and place bets according to the previous known results. The same applies to *ORACLIZE*. Additional, as they are implemented on the Ethereum platform, processing a bet can take up to multiple minutes.

Edgeless [4] provides a fast dice game. They do not provide any detailed documentation of the underling implementation in their whitepaper. The following details are extracted from reviewing their smart contract and their user interface. To start playing the player deposits his funds to their smart contract. For random number generation a server and player generated a random number (seed) is combined. The hash of the server seed is shown to the player before every bet. To check the fairness the player would have to check the server seed hash and combine his seed and the server seed for every single bet. So, checking fairness is more complicated as in traditional provable fair dice games like e.g. Primedice [9].

Funfair [7] is a company providing blockchain solutions for online casinos. They do not have a dice game, but they are working on a state channel implementation. As they are currently closed source and do not provide any detailed documentation, no detailed comparison can be provided.

2. Overview

The three main problems to solve are

- Provable payout of the player funds and winnings. It should not be possible for the casino to withhold the player funds or player winnings.
- The bet result generation should be fast. It should not take more than one second until the result is available to the player.
- The random number generation should not be dependent on third party services. The resulting random number for every bet should only be dependent on both S and P. Both should not know the result beforehand.

Providing fast bet resolution, which implies the result should be available to the player under one second, the implementation can currently not be completely smart contract based. Additionally, secure and

fast on chain random number generation is currently not possible on the ethereum platform [6].

In order to solve these problems, we use a state channel [3] approach. Only for creating a game session, which can comprise many bet, ending the game session and management and settlement of conflicts, smart contract interaction in necessary. For every bet the current game state is countersigned by both S and P. If a conflict arise the current state can be pushed to the smart contract, which handles processing the bet and ends the game session. The details are described in section 3

Creating a game session, the player funds used for placing bet are deposited in the smart contract. S does not have access to them so it can not withdraw or otherwise remove the player funds. The house stake used to pay out the winnings is also managed by the smart contract. So, the players can check if the house funds are sufficient.

For random number generation both P and S generate random seeds and generate, a chain of pseudo-random numbers by using Keccak hash function [1] for successive hashing the previous results. For detailed description see subsection 3.1. It is easy to recalculate the chain entries in one direction but as hash function are non-invertible it is impossible calculate entries in the other direction.

For every bet for random number generation the respectively previous entries of the hash chains are used as seeds for random number generation. The seed of S a P are combined and hence the random number generated. During a game session the random numbers for every bet can be easily verified by just rebuilding both hash chains starting with the last placed bet seeds.

3. Protocol Description

After giving a brief overview we will know describe the implementation in detail. Before placing bets, the player needs to initiate a game session. The number of bets possible for a game session is dependent on the hashchain size. For a hashchain with size n, $n-2$ bets can be placed. During one game session only three blockchain interactions are necessary. Two[1] for creating the session and one for ending the session.

All interactions with the smart contract or externally defined functions are formatted as typewriter text. E.g. the function called by the player to create a games session is written as `createGame`.

3.1. Creating a game session.

For starting the game session P generates a random value r_p and uses it as a seed for generating a hash chain by hashing iteratively the previous result with the Keccak hash function. Formally the generation of the hash chain is defined as:

$$\text{hc}(r,i) = \begin{cases} \texttt{KEC}(r), & i = 0 \\ \texttt{KEC}(\text{hc}(r, i-1)), & i > 0 \end{cases} \tag{1}$$

The number of entries n of the hash chain dictates maximum number of games playable. As described, currently we use $n = 1000$ iterations so at most $1000-2$ games can be played during one game session. To start a new game session P deposits his funds for the game session and publish his last hash chain entry $\text{hc}(r_p, n-1)$ to the smart contract:

$$\texttt{createGame}(\text{hc}(r_p, n-1), \{value : f_p\}) \tag{2}$$

The funds are securely stored in the contract and $hc(r_p, n-1)$ is published by the smart contract.

When he server detects a new game session which is identified by g (a simple counter incremented for every new game session) it accepts the game the S by generatin a random number r_s and and a hash chain of size n. Finally, S publishes the last entry of its hash chain to the contract:

$$\texttt{acceptGame}(g, \text{hc}(r_s n-1)) \tag{3}$$

3.2. Placing bets.

Placing bets happen by counter signing of current bet and previous results and revealing previous entry of hash chain for random number generation. The Game state $G = (B, S)$ consists of the bet B and the signature S. The bet B comprises the following fields:

- **roundId:** A scalar value equal to the number of placed bets for the game session; formally: B_r
- **gameType:** Game type of placed bet; formally: B_t
- **number:** Number chosen by player. E.g. between 1 and 98 for dice; formally: B_n
- **value:** Amount player wants to bet; formally: B_v
- **currentBalance:** Profit or Loss of previous bets. Initial 0; formally: B_b
- **serverHash:** Current server hash; formally: $B_{sh} = \text{hc}(p_s, n - B_r)$.
- **playerHash:** Current player hash; formally: $B_{ph} = \text{hc}(s_s, n - B_r)$.
- **gameId:** Game session identifier. Simple counter incremented for every game session created; formally: B_g
- **contractAddress:** Address of contract this bet is valid; formally: B_c

The signature comprises the fields:

- **playerSignature:** Player' s signature over bet data B; formally: S_p
- **serverSignature:** Server' s signature over bet data B; formally: S_s

[1]Creating a game session can be done in one transaction (Will be implemented in a future version).

3.2.1. *Initiating a bet.* To place a bet the player increments the current round id, sets the game type g, chooses a number n, a value v, sets the hashes of S and P and the balance b. b is the balance of the previous bet state added to the player profit of the last bet. Formally with B^* being the new bet state and B being the old state:

$$B^*_r = B_r + 1 \tag{4}$$

$$B^*_t = g \tag{5}$$

$$B^*_n = n \tag{6}$$

$$B^*_v = v \tag{7}$$

$$B^*_b = b \tag{8}$$

$$B^*_{ph} = \mathrm{hc}(p_s, n - B^*_r) \tag{9}$$

$$B^*_{sh} = \mathrm{hc}(s_s, n - B^*_r) \tag{10}$$

The new bet state $B*$ is signed by P. The signature and B^* is send to S. S verifies the signature the bet state B^*. If verification passes S signs the bet B^* and returns the signature to P Subsequent P verifies the signature of S.

3.2.2. *Revealing the seed.* If the signature S is valid P sends his seed $p_s = hc(r_s, n - B^*_r - 1)$, which is nothing more than the previous entry in the hash chain, to S. S verifies the seed. Formally:

$$\vdash \texttt{KEC}(p_s) = B^*_{ph} \tag{11}$$

If verification succeeds S returns its seed $s_s = hc(r_p, n - B^*_r - 1)$ to P. P verifies the seed:

$$\vdash \texttt{KEC}(s_s) = B^*_{sh} \tag{12}$$

3.2.3. *Result calculation.* To finally generate the random number r both seeds are concatenated, and the Keccak hash is calculated:

$$r = \texttt{KEC}(s_s \parallel p_s) \tag{13}$$

The bet result calculation is dependent of the game type, e.g. for dice, the result number is calculated by r mod 100. Modulo bias [11] is negligible as $r \ggg 100$. With the result number the new balance can be calculated (game type dependent) and the next round can be started.

3.3. **Ending the game session.** To end the current game session P signs a special bet state with game type $B_t = 0$. As above B describes previous bet state B^* is the new one:

$$B^*_r = B_r + 1 \tag{14}$$

$$B^*_t = 0 \tag{15}$$

$$B^*_n = 0 \tag{16}$$

$$B^*_v = 0 \tag{17}$$

$$B^*_b = B_b \tag{18}$$

$$B^*_{ph} = \mathrm{hc}(p_s, n - B^*_r) \tag{19}$$

$$B^*_{sh} = \mathrm{hc}(s_s, n - B^*_r) \tag{20}$$

As when placing bets $B*$ and the signature is send to S where the signature and the state is verified. If both are valid S signs B^* and sends the the current bet state B^* and the signature of P to the smart contract:

$$\texttt{serverEndGame}(B^*, S^*_p) \tag{21}$$

The smart contract validates the signature and sends the balance added to funds initial deposited by the player back to him.

3.4. **Conflict Resolution.** In the above chapter we have described how a regular game session, when player and server comply with the rules, is handled. Know will examine which conflicts between player and server can arise and describe how they can be resolved using the conflict resolution logic implemented in the smart contract.

If the opponent does not respond the conflict initiator can force the termination of the game session. To give the opponent enough time to respond to a conflict all functions forcing game session termination are time locked! Additionally, to offer an incentive to regular end game session a fee needs to paid the opponent when he does not respond in time and the game session termination is forced.

3.4.1. *S does not respond after game session creation.* P starts game session g and the server stops responding. With `playerCancelActiveGame`(g) the player can initiated the termination of the game session. After specific time t_s has passed to give the server time to respond the player can force the termination of the game session and gets his deposited funds send back.

3.4.2. *S does not allow to place new bets or to end the game session.* With the current game state $G = (B, S)$ P can push the current bet state B, his seed $p_s = \mathrm{hc}(p_s, n - B_r - 1)$ and the signature of S to the smart contract:

$$\texttt{playerEndGameConflict}(B, p_s, S_s) \tag{22}$$

S has time t_s to push a newer bet state B^* or confirm state of P. If the server does not respond in time t_s P can force the game session termination. The bet state B is used by the smart contract to calculate players earnings. The player earnings added to the initial deposited funds are send back to P.

3.4.3. *S does not respond to the reveal seed request.* P placed a bet B^*, S signed the bet and returned its signature to P. In the next step P sends his seed to S but it does not return its seed. P can now publish the current bet state B^* and his seed $p_s = \mathrm{hc}(r_s, n - B_r - 1)$ to the smart contract:

$$\texttt{playerEndGameConflict}(B, p_s, S_s) \tag{23}$$

If S does not reveal its seed int time t_s to the smart contract, the player can force the termination of the game session and the bet is processed as the player has won.

3.4.4. *P creates a game session but does not play.* S can initiate the termination of the game session:

(24) `serverCancelActiveGame`()

P has time t_p to push newer bet state. If he does not, S can force the termination of the game session. The funds deposited by P are send to back to him.

3.4.5. *P stops playing without ending the game session.* With the game state $G = (B, S)$ S can publish last bet state B, his seed $s_s = \mathrm{hc}(r_s, n - B_r - 1)$ and the player seed $p_s = \mathrm{hc}(r_s, n - B_r - 1)$ to the contract:

(25) `serverEndGameConflict`(B, s_s, p_s, S_p)

If P does not publish a newer one or confirms the current state in time t_p, S can force the termination of the game session. The current balance added to the initial deposited funds are send back to P.

3.4.6. *The player does not reveal his seed.* The bet B^* is signed by P and S and in the next step P would need to reveal his seed. Without P revealing his seed both P and S can not calculate the random number needed to get the bet result. So, this case can be handled as the bet B^* was never placed and the player stopped playing, see subsubsection 3.4.4.

This is the only real difference in player and server side conflict handling. If S does not reveal his seed the bet can bet ignored as both S and P do not know the bet outcome. But the case the server does not reveal its seed special logic is needed. As in the previous step P send his seed to S, so S can calculate the result. If it does not like it, it could just not return its seed. So in this case the bet can not be ignored and this conflict needs special handling as described in subsubsection 3.4.3.

4. Conclusions

We have introduced and formally defined the protocol of a state channel based dice game. It provides fast game play and yet it is provable fair and secure. The complete game session can be verified by the player. The payout of the winnings and funds of the player cannot be withhold by the operator. The implementation can be tested under `https://dicether.com`. The source code can be found at `https://github.com/dicether`.

References

[1] Guido Bertoni et al. *KECCAK*. 2017. URL: `https://keccak.team/keccak.html`.

[2] *BitZino And The Dawn Of 'Provably Fair' Casino Gaming*. Aug. 2012. URL: `https://web.archive.org/web/20121005100002/http://www.forbes.com/sites/jonmatonis/2012/08/31/bitzino-and-the-dawn-of-provably-fair-casino-gaming`.

[3] Jeff Coleman, Denton Liu, and Anna Wang. *Fully Abstracted State Channels*. 2017. URL: `https://github.com/ledgerlabs/state-channels/wiki/Fully-Abstracted-State-Channels`.

[4] Edgeless. "EDGELESS CASINO CROWDSALE WHITEPAPER". In: (Dec. 2016).

[5] Etheroll. "ETHEROLL DICE GAME WHITEPAPER". In: (2016).

[6] *HPOC_2015*. 2015. URL: `https://github.com/ethereum/wiki/wiki/HPOC%5C_2015`.

[7] Longley and Oliver HoptonWOOD. "FunFair Technology Roadmap and Discussion". In: (June 2017).

[8] *ORACLIZE LIMITED*. URL: `https://oraclize.it`.

[9] *Primedice*. URL: `https://primdice.com`.

[10] *RANDOM.ORG*. URL: `https://random.org`.

[11] Wikipedia contributors. *Fisher–Yates_shuffle — Wikipedia, The Free Encyclopedia*. [Online; accessed 10-Apr-2018]. 2018. URL: `https://en.wikipedia.org/wiki/Fisher%E2%80%93Yates%5C_shuffle%5Cbibtex%20escape%20#Modulo_bias`.

[12] DR. GAVIN WOOD. "ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER BYZANTIUM VERSION c0c3b5d - 2018-04-19". In: (Apr. 2018).