Random Number Generation in CoinPoker

Mike A. Segal, CoinPoker Advisor

2018 March 29
Last Updated: 2018 July 11

1 Introduction

The purpose of this document is to explicate a protocol for the provision of secure and provably fair poker games on the CoinPoker platform, and is based on the work done with the CoinPoker team on 2018 March 27-29.

There exist two classes of solutions: those which rely upon a trusted third-party, and those which operate in a peer-to-peer or serverless capacity or which place limited trust value in the server.

This document focuses on a proposal for the first type of solution which was ultimately selected for implementation in CoinPoker. We provide a high-level specification for the protocol and implementation notes where applicable.

2 Verifiable RNG with Trusted Third-Party

If there exists a third-party who is trusted to maintain the secrecy of the RNG seed, then we can enable users to verify the fairness of the RNG via a simple commitment protocol.

2.1 Protocol Description

Let $\mathcal{O}$ denote the trusted third-party, a.k.a. the **Operator**.

Before each hand of poker begins, let N be the number of players who will participate in the upcoming hand and denote the players $\mathcal{P}_1, \ldots, \mathcal{P}_N$.

Let $\mathcal{H}$ be a cryptographic hash function. For the purpose of this implementation, it is recommended that $\mathcal{H}$ be set to the **KECCAK-256**[1] Sponge-based hash function, which is the same secure cryptographic hash function used throughout the Ethereum protocol.[2]

The participants proceed as follows:

1. $\mathcal{O}$ instantiates a pseudorandom number generator (PRNG)[3] seeded with a secret seed r^{init}.[4] [5]

[1] `https://keccak.team/keccak.html`

[2] It should be noted that the KECCAK-256 implementation used by Ethereum differs slightly from the SHA-3 hash function standarized by NIST in FIPS-202 2015. They have identical security properties, but for compatibility with Ethereum, it is recommended that this implementation use the original KECCAK-256 proposal.

[3] The SIMD-oriented Fast Mersenne Twister (SFMT) specified in the CoinPoker whitepaper is a suitable choice of PRNG.
See: `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/`

[4] The seed need not actually be random, but it must be unpredictable to players. It is common to use the server clock as a PRNG seed, which is sufficient for this use case, but we must ensure that it uses millisecond (or smaller) time resolution—second resolution is not good enough!

[5] The value of r^{init} is ephemeral. Once the PRNG has been instantiated, it should be forgotten by $\mathcal{O}$.

2. For each card value[6] d_i, the operator uses the PRNG from Step 1 to generate a random salt value[7] a_i, and computes card commitments $e_i \coloneqq \mathcal{H}(a_i \| d_i)$.[8]

3. Beginning with a deterministically ordered (i.e. unshuffled) deck, $\mathcal{O}$ uses the same PRNG from Step 1 to run the Fisher-Yates-Durstenfeld shuffling algorithm[9] to derive a shuffled deck $\mathcal{D}^{init}$, which consists of the card values d_i in shuffled order. We define p^{init} to be the permutation on the indices i representing the shuffled order of $\mathcal{D}^{init}$.

4. $\mathcal{O}$ constructs the initial deck commitment vector $\mathbf{e}^{init} \coloneqq \{e_{p^{init}(i)} \mid 0 \leq i < 52\}$ consisting of the card commitments e_i in the same order as $\mathcal{D}^{init}$. $\mathcal{O}$ sends the card commitment vector $\mathbf{e}^{init}$ to all players $\mathcal{P}_1, \ldots, \mathcal{P}_N$.

5. Each player $\mathcal{P}_i$ generates a 256-bit random seed s_i.[10]

6. Each player $\mathcal{P}_i$ computes $c_i \coloneqq \mathcal{H}(s_i)$ and sends the value c_i to $\mathcal{O}$.

---

[6]The card values can be represented using the first 52 unsigned integers [0..52).

[7]Salts should be fixed-length values of at least 32-bits in length.

[8]The plaintext card value d_i must be included as an input to this hash commitment to ensure that $\mathcal{O}$ cannot switch the cards later. The salt a_i is also needed to prevent players from reversing the hash operation by brute force.

[9]The Durstenfeld optimization to the Fisher-Yates shuffling algorithm is described in pseudo-code here: `https://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle#The_modern_algorithm`

[10]In this and all subsequent sections, we prescribe that client-side random numbers should be generated by the client operating system's non-blocking random number generation faculty. In particular, never rely on libraries that generate random numbers computationally.

7. If a player $\mathcal{P}_i$ is not present (i.e. offline), $\mathcal{P}_i$ delivers an invalid value to $\mathcal{O}$[11], or $\mathcal{P}_i$ fails to deliver any value to $\mathcal{O}$ within a specified time interval, then $\mathcal{O}$ sets $s_i \coloneqq i$ and computes $c_i \coloneqq \mathcal{H}(s_i)$.[12]

8. $\mathcal{O}$ then generates a 256-bit random seed s_0 and computes $c_0 \coloneqq \mathcal{H}(s_0)$.

9. $\mathcal{O}$ sends all players $\mathcal{P}_1, \ldots, \mathcal{P}_N$ the commitment vector $\mathbf{c} \coloneqq \{c_i \mid 0 \leq i \leq N\}$.

10. Upon receiving $\mathbf{c}$, each player $\mathcal{P}_i$ sends s_i to $\mathcal{O}$.

11. If a player $\mathcal{P}_i$ who participated in Step 5 is not present (i.e. offline) or fails to deliver a valid value to $\mathcal{O}$ within a specified time interval, then the Operator must deliver an **Abort** message to all players and restart the protocol.[13] [14]

12. $\mathcal{O}$ computes the aggregate seed $s^* \coloneqq \mathcal{H}(s_0 \| \ldots \| s_N)$.

13. $\mathcal{O}$ instantiates a new pseudorandom number generator (PRNG) seeded with s^*.

14. Beginning with the initial deck $\mathcal{D}^{init}$, $\mathcal{O}$ feeds the values generated by the PRNG from Step 13 to the Fisher-Yates-Durstenfeld shuffling

[11] As we are using a 256-bit hash function, a valid commitment value consists of any 256-bit string.

[12] $\mathcal{O}$ can set any fixed value for s_i, but should not select a random or nondeterministic value so as to provide transparency about the lack of entropy contributed by $\mathcal{P}_i$.

[13] To prevent denial of service attacks, the Operator may choose to exclude a player who repeatedly aborts the protocol at this stage from contributing entropy to future iterations of the protocol. In such cases, the player should be treated as if they failed to deliver the commitment value as in Step 7.

[14] Note that it is not sufficient for $\mathcal{O}$ to generate a new s_i and continue the protocol at this stage. In addition to causing the verification to fail, this would permit $\mathcal{O}$ to compute a seed value that generates any possible card sequence, since $\mathcal{O}$ is already privy to the players' seeds.

algorithm to derive the final shuffled deck $\mathcal{D}^{final}$. We define p^{final} to be the permutation representing the transformation of $\mathcal{D}^{init}$ into $\mathcal{D}^{final}$.

15. The hand is then played to completion using the shuffled deck $\mathcal{D}^{final}$ from the previous step.

16. Upon completion of the hand, $\mathcal{O}$ sends all players $\mathcal{P}_1, \ldots, \mathcal{P}_N$ the seed vector $\mathbf{s} \coloneqq \{s_i \mid 0 \leq i \leq N\}$.

17. Privately, each player uses $\mathbf{s}$ to compute s^* and p^{final}. Each player then applies p^{final} to the card commitment vector e^{init} from Step 4 to derive the final vector of card commitments $\mathbf{e}^{final} \coloneqq \{e_{p^{final}(p^{init}(i))} \mid 0 \leq i < 52\}$ consisting of the card commitments e_i in the same order as $\mathcal{D}^{final}$.[15]

18. Upon completion of the hand, $\mathcal{O}$ also sends each player $\mathcal{P}_k$ the salts a_i corresponding only to those cards which $\mathcal{P}_k$ has seen in the course of the hand.

19. Each player $\mathcal{P}_k$ then independently verifies:

    (a) That the value of s_k in the received seed vector $\mathbf{s}$ matches the value of s_k generated in Step 5.

    (b) That $c_i = \mathcal{H}(s_i)$, for all $0 \leq i \leq N$.

    (c) By applying the salts from Step 18 to card values and comparing with $\mathbf{e}^{final}$: That all cards seen by $\mathcal{P}_k$ in the course of the hand[16] match the corresponding positions in $\mathcal{D}^{final}$.

[15] To be clear, the players are not privy to p^{init}, but they can apply p^{final} to the initially-ordered $\mathbf{e}^{init}$ to derive card commitments in the final ordering.

[16] To be as comprehensive as possible, this should include private cards, community cards, and any cards revealed during the hand, such as mucked cards, face-up burnt cards, and any cards revealed in the showdown.

2.2 Consideration

To aid in the verifiability by end-users of the design and implementation of the protocol, it is recommended that the parts of the client responsible for executing the above protocol be encapsulated in an open source module that is capable of being independently compiled and linked to the client.[17]

[17] In this scenario, the closed source portion of the client software is considered to belong to the Operator $\mathcal{O}$. The open source module responsible for random number generation and verification should not share any data with the closed source client unless and until it is ready to share that data with the server.