ZKShuffle: Mental Poker on SNARK for Ethereum

Shumo Chu[†], Boyuan Feng[†], Rahul Maganti[♢] *
[†]p0x labs, [♢]Jump Crypto

1 Background and Motivation

ZKShuffle is p0x labs' implementation of Barnett and Smart's Mental Poker scheme [1], which is dervied from the seminar work of RSA [2]. ZKShuffle is a framework for playing card/board games without physical cards and without a trusted third party. For example, if you build a mental poker framework for a deck of 52 cards, you can almost write any poker game such as Texas Hold'em using only Solidity. For a comprehensive overview, we direct the reader to the series of articles published by Geometry [3, 4].

However, Geometry's implementation [5] requires verifier to run a linear-sized multi-exponentiation, which is expensive on Ethereum . In this work, we propose a new mental poker design focusing on reducing gas costs on Ethereum. Technically this design still follows the design from Barnett and Smart (also the one Geometry uses), the difference is that we implements the new shuffle argument using Groth16 [6]. This reduces the cost of each shuffle and decrypt to constant cost of verification on chain (thus also scales to more players). Now, everyone can play Texas Hold'em and Hearth Stone on Ethereum!

2 ZKShuffle Scheme Overview

At a high level, ZKShuffle can distribute and shuffle a deck of cards privately to each individual player. And a player can reveal a single card (or a subset of cards) on her hand when she plays her hand.

Our construction uses Groth16 on a pairing-friendly curve. Thus, we use the notation $\mathbb{G}$ for the group element in the embedded curve (i.e. Baby Jubjub), F_q for the base field of $\mathbb{G}$, and F_r for the scalar field of $\mathbb{G}$.

Suppose there are k players ($P = \{p_1, \cdots, p_k\}$) and a deck of n cards ($C = [c_1, \cdots, c_n]$). Our zkShuffle scheme consists of 4 functions:

- Setup: The mental poker scheme provides a generator $g : \mathbb{G}$. Each player p_i generates a random secret key $sk_i : \mathbb{F}_r$ and uses the generator to produce her public key $pk_i : \mathbb{G}$ where:
$$pk_i := sk_i \cdot g$$
We also get an aggregated public key:
$$pk := (sk_1 + \cdots + sk_k) \cdot g$$

- shuffle_encrypt: To shuffle a deck of cards, every player needs to take her turn to call shuffle_encrypt. A player p_i takes a deck of cards C_i from previous player, and

*Ordered alphabetically.

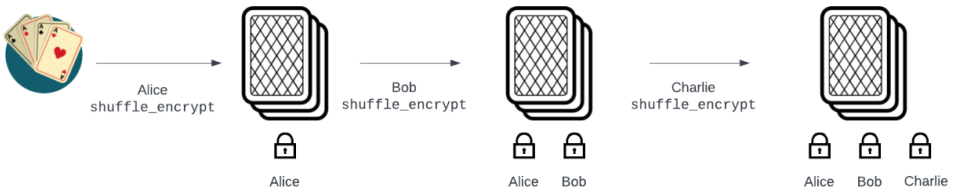

Figure 1: Shuffle the Deck Intuition

then shuffles and encrypts to produce a new deck of card C_{i+1}. First, let A be a randomly sampled permutation matrix. p_i can produce a deck of shuffled cards:

$$A \cdot C_i$$

Then, p_i randomly sampled a vector $R_i : \mathbb{F}_r^n$, and applied a homomorphic encryption scheme (ElGamal):

$$C_{i+1} = ElGamal(g,\ pk, A \cdot C_i, R_i)$$

By the homomorphic property of ElGamal, one nice property of shuffle_encrypt is, the order of encryption is irrelevant to the result! After the shuffle and encryption of the deck, p_i posts the produced deck as well as a zero-knowledge proof of the validity of the shuffle and encryption on chain.

- decrypt: A player p_i takes a set of encrypted cards $\mathcal{C}$ to make *one round* of decryption. Let's ignore the detail now.
- decrypt_post: A player p_i takes a set of encrypted cards and calls decrypt. After decryption, the player posts the decrypted card as well as the validity proof of decryption on chain.

3 Intuition Behind ZKShuffle and How to Use It

Let's say Alice, Bob, and Charlie want to play a poker game together. The first step is that they need to shuffle a deck of encrypted cards *together*. We require each player to join the shuffle so that **no subset** of players can control the sequence of shuffled cards or encryption. Here, we need to use a homomorphic encryption scheme [7], to ensure the sequence of encryption and decryption would not affect the final result.

3.1 Shuffle the Deck

To shuffle the deck, we start from a deck of open cards. Then, each player takes turns calling shuffle_encrypt to randomly shuffle the deck of cards she received and apply the homomorphic encryption scheme ("add a lock" as shown in Fig. 1). At the end of the round, we achieve:

1. a deck of shuffled and encrypted cards on chain
2. every player has shuffled the deck once and encrypted each card once

Let's ask two questions:

Q1. Why every player needs to join the shuffle?

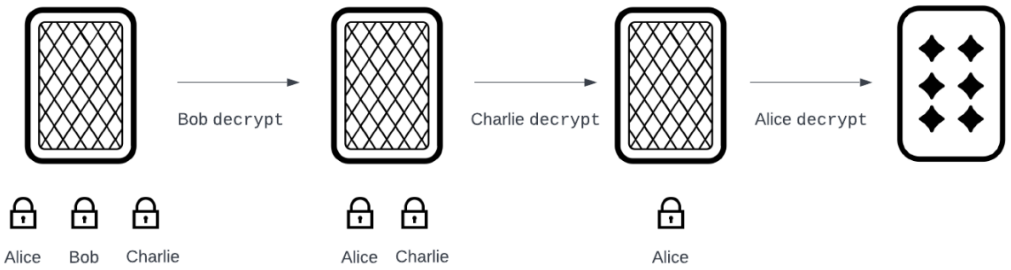

Figure 2: Card Dealing Intuition

We don't want to trust anyone. Notice, after the first player, each player only shuffles the encrypted card deck. Essentially every player contributes to the randomness of the final deck and *unless all players collude*, the shuffle is fair.

Q2. Why do we need homomorphic encryption?

Homomorphic encryption is needed since the order of encryption and decryption is *irrelavent* to the end encrypt/decrypt result. We need to leverage this property in the card dealing (subsection 3.2) part.

3.2 Card Dealing

Card dealing handles 3 cases in the poker game:

- Deal a card from the deck to a single player (say Alice).
- A single player plays one card from her hand.
- Deal a community card

Let's first look at how to deal with a Card to Alice. Suppose we already have a shuffled and encrypted deck on chain. And now we are dealing the first card on the deck to Charlie. Below is the sequence of actions, as illustrated in Fig. 2:

1. Bob calls `decrypt_post` on the first card, which calls `decrypt` the first card and writes it back on chain
2. Charlie calls `decrypt_post` on the card (notice, now this is the card Bob already calls `decrypt_post`), and write it back on chain.
3. Alice fetch the card on chain, and calls `decrypt` (without writing it back on chain, otherwise everyone knows). Notice, after 2, only Alice's encryption (a.k.a. lock) is left on the card. So Alice can decrypt the card privately in her hand.

Now, suppose Alice wants to play the card just dealt to her from her hand. She reveals the card and uses zero-knowledge proof to show that this is indeed the card she decrypted from the encrypted card given to her.

And also dealing community cards is simply a full round of `decrypt` and the last person needs to reveal the card on chain!

Let's also ask three questions about dealing:

Q1. When dealing a card to Alice, what prevents other players or anyone else from seeing Alice's card?

For the entire dealing sequence, only Alice can see the plain text after her final decrypt. Through the process, neither Bob, Charlie, nor blockchain validators can see her card.

Q2. Does the decrypt sequence have to be Bob →Charlie→Alice?

No. Thanks to the homomorphic encryption scheme we use, as long as Alice is the last one to decrypt the card, we are good!

Q3. When Alice plays her hand, what prevents her from cheating (a.k.a. playing a card that doesn't belong to her)?

That is why when Alice plays her hand, she cannot just reveal her card, she needs to submit a zero-knowledge proof to show the validity of her decryption on chain as well. We will get into details about when and where SNARK is needed in section 4.

4 Where and Why is a SNARK needed?

Both shuffle_encrypt and decrypt_post require the player to generate a zero-knowledge proof and submit the zero-knowledge proof together with the result.

In shuffle_encrypt, a zero-knowledge proof is needed to prove the validity of the shuffle and encryption. This is required to guarantee the soundness of the shuffle, for example, people don't put random content in the encrypted cards so that nothing can be decrypted. For not leaking shuffle order on chain, the proof generated in shuffle_encrypt needs to be strictly zero-knowledge.

In decrypt_post, a zero-knowledge proof is needed to prove the validity of the decryption. This prevents from the player posting the wrong decryption results intentionally to gain an advantage in the game. The proof generated in decrypt_post needs to be zero-knowledge for not leaking the player's secret key.

5 Detailed Construction

5.1 El Gamal Encryption

Setup: Given a randomly selected generator $g : \mathbb{G}$ and each player's randomly selected secret key $sk_i : F_r$, we can produce an aggregated public key $pk : \mathbb{G}$

$$pk = (sk_1 + \ldots + sk_k) \cdot g$$

Encrypt. This step has the following signature:

$$\mathsf{Encrypt}(g : \mathbb{G},\ pk : \mathbb{G},\ m : \mathbb{G} \times \mathbb{G},\ r : F_r) \to \mathbb{G} \times \mathbb{G}$$

The output is ciphertext $(c_1 : \mathbb{G}, c_2 : \mathbb{G})$ where

$$c_1 = m[1] + r \cdot g$$
$$c_2 = m[2] + r \cdot pk$$

Decrypt. This step has the following signature:

$$\mathsf{decrypt}(sk : F_r,\ C : \mathbb{G} \times \mathbb{G}) \to \mathbb{G}$$

The output is message m where

$$m = c_2 - sk \cdot c_1$$

5.2 Shuffle and Remask Argument on SNARK

Permutation Matrix A matrix M is a permutation matrix if

- It is a square matrix. More specifically, supposing there are m rows and n columns, we have $m = n$.
- Each element $m_{i,j}$ is either 0 or 1.
- The sum of each row and each column is exactly 1. In other words, we have

$$\sum_i m_{i,j} = 1, \forall j \in \{1, 2, \cdots, n\}$$

$$\sum_j m_{i,j} = 1, \forall i \in \{1, 2, \cdots, n\}$$

shuffle_encrypt circuit For a poker game, there is a pre-defined number of cards n.

Given

- Public generator $g : \mathbb{G}$ and an aggregated public key $pk : \mathbb{G}$
- Public matrices of masked cards

$$\mathcal{X} = [(x_{0,0}, x_{0,1}), (x_{1,0}, x_{1,1}), \cdots, (x_{n-1,0}, x_{n-1,1})]$$

$$\mathcal{Y} = [(y_{0,0}, y_{0,1}), (y_{1,0}, y_{1,1}), \cdots, (y_{n-1,0}, y_{n-1,1})]$$

the prover proves the knowledge of

- Private matrix $\mathcal{A}$ of shape $n \times n$
- Private vector of randomness

$$\mathcal{R} = [r_0, r_1, \cdots, r_{n-1}]$$

such that

- $\mathcal{A}$ is a permutation matrix
- $\mathcal{B} = \mathcal{A} \times \mathcal{X}$
- $\mathcal{Y} = \mathsf{ElGamal.Encryption}(g, pk, \mathcal{B}, \mathcal{R})$

Here, $ElGamal.Encrypt(g, pk, \mathcal{B}, \mathcal{R})$ means n ElGamal encryption with individual r_i.

Cost Analysis We implement the circuit in Circom, the total R1CS constraint count is about $170,000$.

decrypt_post circuit Suppose a prover wants to decrypt the i^{th} card value.

Given

- Public generator $g : \mathbb{G}$
- Public masked card $y_i = (y_{i,0}, y_{i,1})$
- Current player's public key $pk_i = sk_i \cdot g$

the prover proves the knowledge of

- Current player's secret key sk_i

such that

- $pk_i = sk_i \cdot g$
- $out = y_{i,1} - sk_i \cdot y_{i,0}$

Cost Analysis We program the circuit in Circom, the decrypt_post circuit consists of $3,500$ R1CS constraints.

6 Acknowledgement

We would like to thank Nicolas Mohnblatt from Geometry for his thorough feedback on the work and manuscript.

References

[1] A. Barnett and N. P. Smart, "Mental poker revisited," in *IMACC*, vol. 2898 of *Lecture Notes in Computer Science*, pp. 370–383, Springer, 2003.

[2] A. Shamir, R. L. Rivest, and L. M. Adleman, *Mental Poker*, pp. 37–43. Boston, MA: Springer US, 1981.

[3] N. Mohnblatt, A. Novakovic, and K. Gurkan, "Mental poker in the age of snarks - part 1." https://geometry.xyz/notebook/mental-poker-in-the-age-of-snarks-part-1.

[4] N. Mohnblatt, A. Novakovic, and K. Gurkan, "Mental poker in the age of snarks - part 2." https://geometry.xyz/notebook/mental-poker-in-the-age-of-snarks-part-2.

[5] N. Mohnblatt, A. Novakovic, and K. Gurkan, "Geometry research's mental poker implementation." https://github.com/geometryresearch/mental-poker.

[6] J. Groth, "On the size of pairing-based non-interactive arguments," in *EUROCRYPT (2)*, vol. 9666 of *Lecture Notes in Computer Science*, pp. 305–326, Springer, 2016.

[7] Wikipedia, "Homomorphic encryption." https://en.wikipedia.org/wiki/Homomorphic_encryption.