Efficient ZK Argument for Shuffle Implementation in Rust

Master's Thesis

University of Freiburg

Ahmet Ercem Bulut
Student ID: 5362638
Supervisor: Prof. Christian Schindelhauer

August, 2024

Abstract

A shuffle operation in cryptography is an operation that takes a committed, anonymous series of values and returns the original serie modified with a permuted order. It is an important operation in many real world scenarios(e-voting, mental card games). Due to the plaintexts or data being encrypted for privacy, the correctness of a shuffle of commitments is not straight forward to verify. While there are algorithms to construct such arguments, we are providing the first comprehensive pure Rust library for the Correctness of a Shuffle Operation.

The library is built over elliptic curve prime order groups to commit and encrypt data for privacy, while exploiting the homomorphism of the elliptic curves for efficiency. This argument for correctness combines two separate arguments(Multi-exponentiation Argument, Product Argument) to produce a Shuffle Argument for correctness.

Utilizing the Rust programming language, which has enormous support from the cryptography community, with it's efficiency in runtime and security in memory, we aim to provide an extensive and easy to use zero-knowledge proof framework that can be seamlessly incorporated and used by other proof schemes, or used to construct complex arguments.

We give performance results of the tests we ran as well to show the implementation follows the proposed optimized performance.

Keywords: The Rust Programming Language, mix-net, zero-knowledge, shuffle, mental card games

Acknowledgements

acknowledgements here.

Contents

Chapter 1

Introduction

With the introduction of the Rust programming language to cryptography, there has been a sizable influx of cryptography research migrating to rust. The community has been at work creating tools and frameworks to exploit Rust's native advantages in performance and memory control.

We aim to contribute to this collection of tools and frameworks by implementing a secure and efficient version of a Shuffle Argument. The Shuffle argument provided by Bayer and Groth [1] prove that a deck of "values" has been shuffled, with an honest zero-knowledge verifier that makes use of two independent arguments.

The sublinear communication complexity the argument provides is an improvement which is essential to smaller communications in proofs, and the main reason we want to combine this efficiency with the efficiency of the Rust Programming Language.

We will describe a library of arguments which ultimately combine into a complete and secure Bayer-Groth Shuffle Argument, and also be used in tandem with other cryptographic proofs and schemes provided by Rust's cryptograhic community.

1.1 Bayer-Groth Shuffle Argument

The Bayer-Groth shuffle argument[1], is a specific type of zero-knowledge proof that allows for the verification of correctness for a shuffle without revealing the underlying permutation or the original values.

This protocol is particularly useful in scenarios where data must be anonymized or shuffled securely, such as in electronic voting systems, secure auctions, and mental card

games.

It enables a prover to demonstrate that a set of ciphertexts (encrypted values) is a valid permutation of another set of ciphertexts, with minimal computational and communication overhead. The protocol achieves this by leveraging homomorphic encryption and prime order groups, ensuring that the shuffle's correctness can be verified without revealing any sensitive information or needing interaction.

1.2 The Rust Programming Language

Rust[2] is a near low-level programming language known for its emphasis on safety, concurrency and performance. Designed to ensure properties that might be overlooked otherwise such as memory safety and reference efficiency, Rust has quickly gained popularity in software development. Particularly for applications where performance and security are critical.

Rust's ownership model, which enforces strict rules on how memory is managed, ensures that programs are both safe and efficient[2]. This is particularly important in cryptographic applications, where errors in memory management can lead to vulnerabilities and exploits.

Rust's zero-cost abstractions and fine-grained control over system resources make it an ideal choice for implementing complex cryptographic protocols, such as the Bayer-Groth shuffle argument.

1.3 Previous Implementations and the Need for Rust

While the Bayer-Groth shuffle argument has been implemented in various programming languages, there has been a growing interest in leveraging Rust's unique features for cryptographic applications.

Existing implementations in languages like C++ or Python often face challenges related to memory safety, runtime performance or communication size. Rust's strict safety guarantees, combined with its ability to produce highly optimized binaries, make it an attractive choice for implementing cryptographic protocols.

Chapter 2

Related Work

2.1 Shuffle Arguments

The shuffle operation has been introduced by Chaum [3], however a proof of correctness for the mix-nets has been troublesome to define over the years. Argument defined by Desmedt and Kurosawa [4] only accounted for a small scale corruption in the servers. Jakobson, Juels, and Rivest [5] proposed a relatively big number of servers to decrease the chance of leaks and tampered messages. Peng et al. [6] limited possible permutations and required a ratio of senders to be honest. These drawbacks were addressed by Wikström [7] and zero-knowledge arguments.

Zero-knowledge shuffles were flushed out by Sako and Killian [8] and Abe [9, 10, 11]. Furukawa and Sako [12] and Neff [13, 14] created the first shuffle algorithms for ElGamal encrypted values with linear card count complexity.

Neff [13] utilized the invraince of polynomials under permutation of roots to create the shuffle argument. Later Groth [15] laid out a perfect honest verifier zero-knowledge protocol. In 2008 Groth and Ishai [16] proposed the first sublinear communication complexity shuffle argument. This was later improved on by Bayer and Groth [1] by decreasing the $\mathcal{O}(N^{\frac{2}{3}})$ communication complexity to $\mathcal{O}(N^{\frac{1}{2}})$, which was the most optimal solution until this year where Abdolmaleki et al. [17] managed to achieve logarithmic communication complexity with bulletproofs.

2.2 Decaf: Eliminating cofactors through point compression

Decaf[18] proposes a point compression format for elliptic curves over large-characteristic fields. Which is an effective implementation of cofactor-4 elliptic curve points used by many state of the art cryptographic libraries.
Prime order groups created by these elliptic curves show additive homomorphism. Which is a significant improvement on performance when it comes to implementation of complex proofs.

2.3 Existing Implementations

Mental Poker

Mental Poker[19] is a library aimed to implement a verifiable mental poker game for research purposes. While it is also purely in Rust, the library focuses more on the implementation and the efficiency of Barnett Smart Card Protocol[20]. They also differ in their choice of cryptography primitives and go with arkworks curve points.[21].

Bayer-Groth Mixnet

A pure C++ implementation[22] of the protocol, for use in a messaging system. Useful for us as well since they have in detail performance metrics we can compare to. They use the same curve 25519 as ours and also give hardware specifications for the performances. They have certain drawbacks such as: for some values of the parameter m the verification fails, the row size m should always be larger than the column size n etc.. Our library works without these limitations as well.

groth-shuffle

A simple implementation by Dalskov [23] in c++, using Rellic library for elliptic curve operations. Kecchak algorithm for compressing the points, but no metrics or further elaborations are made.

Chapter 3

Objective and Structure

Implementing the mathematical operations that construct a Shuffle argument is a trivial task with enough experience in programming. The main objective of this thesis is to utilize Rust's unique properties of memory safety and runtime efficiency to create a safe and verifiable library of arguments that is easy to use and incorporate.

By the end of the paper we hope to have given a convincing argument to why our library is secure and efficient, and also why Rust is a perfect match for such arguments of proof.

Thesis Structure

In the remaining parts of the paper we will first talk about importand preliminary concepts. First we will talk about Pedersen commitment, ElGamal encryption, and their respective schemes. We will also explain their homomorphism and how it could be exploited for arguments of knowledge.

We will go over Zero-Knowledge proofs and the schemes used in the shuffle argument. Setting up for Bayer and Groth's shuffle argument, we will discuss how they achieved their optimized shuffle argument. Finally we will describe Ristretto Points and how we incorporated it to our scheme.

In Implementation, we will explain our structure and try to show how we utilized, Rust's efficiency and safety for certain important parts of the protocol.

We will end our discussion with performance metrics we have collected and their significance, followed by talking about what could be future additions to the library.

Chapter 4

Methodology

4.1 Pedersen Commitment

Pedersen commitments are a cryptographic technique used to commit to a value while keeping it hidden, with the ability to later reveal the committed value. They are widely used in cryptographic protocols, including zero-knowledge proofs, due to their strong security properties and the ability to support homomorphic operations.

Key Components

- **Group Parameters:** Pedersen commitments rely on a cyclic group G of prime order q, where the discrete logarithm problem is hard. Two generators g and h of the group are publicly known.
- **Commitment Value:** A value C that commits to a message m using a random blinding factor r.

Commitment Process

To commit a message m (which is typically an integer modulo q), the commiter:

- Chooses a random blinding factor r uniformly from $\mathbb{Z}_q$.
- Computes the commitment C as:
  $C = g^m \cdot h^r \pmod{q}$

The commitment C is then published, while m and r are kept secret.

Revealing the Commitment

To reveal the commitment, committer discloses the values m and r. The verifier checks the valididty by computing $C' = g^m \cdot h^r \pmod{q}$ and ensuring that $C' = C$.

Security

- **Hiding:** The commitment C does not reveal any information about the message m due to the random blinding factor r. This ensures privacy until the committer chooses to reveal the value.
- **Binding:** Once C is published, the committer cannot change the value of m without being detected, as doing so would require finding a different pair (m, r) that produces the same commitment, which is infeasible due to the hardness of the discrete logarithm problem.

4.2 ElGamal Encryption

ElGamal encryption is a public-key cryptosystem based on the Diffie-Hellman key exchange[24]. The security of the encryption relies on the difficulty of solving the discrete logarithm problem in large cyclic groups. Without having the private key x, it is computationally infeasible to derive the plaintext from the ciphertext.

Key Components

- **Private Key:** A randomly chosen integer x, which is used to generate the public key. Needs to be secret.
- **Public Key:** Consists of a large prime number p, a genarator g of a multiplicative group modulo p, and a public key y, where $y = g^x \pmod{p}$.

Encryption Process

To encrypt a message m(represented as a number), the sender:

- Chooses a random integer k, which serves as the ephemeral key.
- Computes the ciphertext pair (c_1, c_2) as follows:

- $c_1 = g^k \pmod{p}$
- $c_2 = m \cdot y^k \pmod{p}$

The ciphertext (c_1, c_2) is then sent to the recipient.

Decryption Process

The recipient, who knows the private key x, decrypts the ciphertext by:

- Computing $s = c_1^x \pmod{p}$.
- Recovering the original message m by computing $m = c_2/s \pmod{p}$.

Applications

ElGamal encryption is used in various cryptographic protocols, including digital signatures and encryption schemes. Its ability to support homomorphic operations makes it particularly useful in privacy-preserving applications, such as secure voting systems and zero-knowledge proofs.

4.3 Homomorphism

Homomorphism refers to the ability of certain operations to respect the structure of the underlying group order. For example Pedersen commitments have additive homomorphism, which means any addition operation applied to the values before committing, would give the same result if the operation would have been applied to the commited versions of the same values.

This characteristic is particularly valuable in scenarios where computations need to be performed on commited data without revealing it, preserving both privacy and security.

The Shuffle argument utilizes the homomorphism of both the Elgamal encryption and the Pedersen commitment.

Homomorphism of Pedersen Commitments

Pedersen commitments' homomorphism as we mentioned is additive: the addition of two commitments is itself the commitment to the sum of the original messages. Mathemati-

cally, if $C_1 = g^{m_1} . h^{r_1}$ and $C_2 = g^{m_2} . h^{r_2}$, then:

$$C_1 + C_2 = g^{m_1+m_2} \cdot h^{r_1+r_2}$$

This allows for multiple commited values to be combined in a way that reflects their arithmetic sum without revealing the individual values themselves.

Homomorphims of ElGamal Ciphertexts

The shuffle argument involves showing that a set of ciphertexts (encrypted values) is a valid permutation of another set without revealing the permutation or the underlying plaintexts.

The usual definition of ElGamal encryption is over a finite field, where the homomorphism is multiplicative. Homomorphims helps us achieve this via providing committed values that we can operate over rather than needing to reveal them in any way. Mathmetically:
The ciphertext (c_1, c_2) of plaintext $m_1 \cdot m_2$:

$$(c_1, c_2) = \left(g^{k_1 \cdot k_2} \pmod p,\ (m_1 \cdot m_2) \cdot y^{k_1 \cdot k_2} \pmod p\right)$$

with k_1 and k_2 being the respective ephemeral keys.

However, we are defining our ElGamal encryption over elliptic curves which turns their homomorphism towards addition like Pedersen commitments. This allows as tu treat encryption and commmitment primitives similarly and make sure they are in the same prime order.

Most of the proof construction algorithms' verification, efficiency and security is assured by using these properties for our arguments.

4.4 Zero-Knowledge Proofs

Zero-knowledge proofs (ZKPs) are a fundamental concept in cryptography, enabling one party (the prover) to demonstrate knowledge of a particular piece of information to another party (the verifier) without revealing the information itself.

This concept, first introduced by Goldwasser, Micali, and Rackoff[25], has since become an important part of privacy-preserving cryptographic protocols. ZKPs are widely used in applications where privacy and security are paramount, such as digital signatures, identity verification, and secure voting systems.

Special honest verifier zero-knowledge argument of knowledge

The zero-knowledge scheme that we are going to be using for our arguments(defined by Bayer and Groth) is defined over a common reference string σ that is the result of a probabilistic polynomial time setup algorithm $\mathcal{G}$ dependent on a arbitrary security parameter. Our common reference string is a tuple $\sigma = (pk, ck)$ where pk and ck are public key for the ElGamal encryption and commitment key for the Pedersen commitment.

The languages of statements x are defined as the set of statements that have a witness w for the relation R under the common reference key. Meaning the shuffle is valid in the case of a shuffle argument, where x is the original and shuffled deck, w is the proposed permutation and the base hiding factor of the base ElGamal encryption.

$$L_\sigma := \{x | \exists w : (\sigma, x, w) \in R\}$$

The commitments revealed by communication create a public transcript, which is the result of algorithms such as the prover $\mathcal{P}$ and verifier $\mathcal{V}$ interacting with inputs. We will denote the transcript entries as $(\sigma, hist) \leftarrow \mathcal{G}(\lambda)$, where λ is the security parameter for example. One can think of the transcript as values needed for verification while remembering it is public and should not share open values. The last entry in this public transcript is the acceptance or rejection of the prover verifier duo.

An *argument* is defined as a triple of the setup algorithm, prover and verifier: $(\mathcal{G}, \mathcal{P}, \mathcal{V})$. This *argument* when faced with an adversary $(x, w) \leftarrow \mathcal{A}(\sigma, hist)$, either accepts the argument via $\mathcal{P}(\sigma, x, w), \mathcal{V}(\sigma, x)$ accepting, or claims the statement and witness provided by the adversary is not in the relation R and rejects. This implies perfect completeness.

The chance of an adversary fooling the verifier with an $x \notin L_\sigma$ and no witness, is computationally hard due to keeping a history of transcript and the common reference key for the language. It would need to solve the discrete log problem for elliptic curves.

An argument is called a public coin if the prover's messages need to adhere to random and independent messages of the verifier. These are called *challenges* made by the

verifier. A public coin's randomness provides honesty to the prover's actions, otherwise random challenges made by the verifier couldn't be verified in the final transcript.

The Fiat-Shamir heuristic

Fiat-Shamir heuristic proposed in 1987 [26] aims to turn an interactive argument, prover and verifier communicate as two entities to create a healthy argument, into a non-interactive one. This means the prover wouldn't have to wait for externally communicated challenges during the construction of the proof.

Not being dependent on verifier for the construction of the proof also means the same proof could be verified by all verifiers, otherwise interactions with every seperate verifier which is a lot more complexity.

Non-interactivity is achieved by a secure random oracle creating challenges instead of the verifier. If the oracle returns a uniformly random answer to brand-new inputs this satisfies the randomness and independence property normally enforced by the verifier. To achieve this the prover requests random challenges from the oracle and sends this challenge to individual verifiers with the proof.

4.5 Efficient ZK Argument for Correctness of a Shuffle

In 2012, Bayer and Groth[1] presented an algorithm with sublinear communication complexity for shuffling a deck of homomorphically encrypted values. According to their findings, operations for an efficient sublinear size argument show linearity in group elements when they are "in the exponent".

Using this adaptation, they constructed an efficient multi-exponentiation argument that shows a ciphertext C is the product of a set of known ciphertexts $C_1, ..., C_N$ raised to a set of hidden committed values.

By reducing this bottleneck sublinearly, the argument gains significant improvement in performance ($\mathcal{O}(\sqrt{N})$). They also provide other optimization and minor improvements over the prover computations.

The algorithms used in the Bayer-Groth paper construct the backbone of our library,

with optimizations that are products of using native Rust and it's efficiency.

While describing the algorithms we will follow the multiplicative homomorphism as to keep notation consistent with Bayer-Groth's paper, however in the implementation multiplication and exponentiation turns into addition and multiplication. Specifically the algorithm consists of 6 proof schemes, some used as components for others. We will describe these arguments in terms of special honest zero-knowledge arguments we described previously for brevity, since the mathematical justifications of the algorithms are beyond the scope of this paper.

We will denote the prime order group used by commitment as $\mathbb{G}$ and write $\mathbb{H} = \mathbb{G} \times \mathbb{G}$ for ciphertexts. $c_a = com_{ck}(a; r)$ indicates a commitment for the value a with a random value of r. $\mathcal{E}(a; b)$ represents the ElGamal encryption of the message a with randomness b. x, y, z will be reserved for challenges of the Fiat-Shamir heuristic from now on. Finally we show a permutation π of $\vec{C}$ as $\vec{C}_\pi$.

Shuffle Argument

The main argument for proof of shuffle.

- **Common Reference:** pk, ck.
- **Statement:** $\vec{C}, \vec{C'} \in \mathbb{H}^N$ with $N = mn$.
- **Witness:** $\pi \in \Sigma_N$ and $\vec{\rho} \in \mathbb{Z}_q^N$ so that $\vec{C'} = \mathcal{E}(\vec{1}; \vec{\rho})\vec{C}_\pi$

Where N is the total card count.

The argument aims to exploit the sum of all ciphertexts being unchanged by a shuffle operation. After three layers of homomorphic challenges to π, the results and the openings are used in a Product Argument, which would only hold if π would be a correct shuffle.

These challenges are also used to prove that a challenged sum of $\vec{C}^{\vec{x}}$ would equal to a sum of the permuted chipertexts with challenged π exponents and challenged ρ randomness. This is the main improvement of Bayer-Groth argument and is the Multi-Exponentation Argument

Multi-Exponentiation Argument

An argument that shows correctness of prime-order group exponentiations. The argument gets rid of overhead matrix elements that are not needed for the argument by only focusing on the main diagonal product(as that mimics the permutation as it contains every ciphertext), instead of all of the diagonals.

- **Common Reference:** pk, ck.
- **Statement:** $\vec{C}_1, \ldots, \vec{C}_m \in \mathbb{H}^n$ and $C \in \mathbb{H}$ and $c_{A_1}, \ldots, c_{A_m} \in \mathbb{G}$ where $m = \mu m'$.
- **Witness:** $A \in \mathbb{Z}_q^{n \times m}, \vec{r} \in \mathbb{Z}_q^m$ and $\rho \in \mathbb{Z}_q$ such that

$$C = \mathcal{E}_p k(1; \rho) \prod_{i=1}^{m} \vec{C}_i^{\vec{a_i}} \text{ and } \vec{c}_A = com_{ck}(A; \vec{r}).$$

With μ being the optimization factor.

Product Argument

An argument proving that the product of certain commited vector values have a particular product.

- **Common Reference:** pk, ck.
- **Statement:** $\vec{c}_A \in \mathbb{G}^m$ and $b \in \mathbb{Z}_q$.
- **Witness:** $A \in \mathbb{Z}^{n \times m}, \vec{r} \in \mathbb{Z}_q^m$ such that

$$\vec{c}_A = com_{ck}(A; \vec{r}) \text{ and } \prod_{i=1}^{n} \prod_{j=1}^{m} a_{ij} = b$$

Product argument engages in a Hadamard Product argument for column-wise commitments to the witness matrix and a Single Value Product argument for the element-wise commitments.

Hadamard Product Argument

An Hadamard product between two vectors of same dimensions is a vector which the elements of are element-wise products of the argument vectors.

The argument shows that, vectors of commited values have a particular result vector for their element-wise(hadamard) product.

- **Common Reference:** pk, ck.
- **Statement:** $\vec{c_A}, c_b$.
- **Witness:** $\vec{a}_1, \ldots, \vec{a}_m, \vec{r}$, and $\vec{b}, s$ such that

$$\vec{c}_A = com_{ck}(A; \vec{r}) \qquad c_b = com_{ck}(\vec{b}; s) \qquad \vec{b} = \prod_{i=1}^{m} \vec{a}_i.$$

Finally engages in a Zero Argument to assert equality between the products of challenged commitments and the challenged product b.

Zero Argument

For two set of commited vectors and a bilinear mapping of two vectors to scalar, Zero argument shows the resulting operation equals 0. Important for showing equality via difference.

- **Common Reference:** pk, ck.
- **Statement:** $\vec{c}_A, \vec{c}_B$ and a bilinear map $* : \mathbb{Z}_q^n \times \mathbb{Z}_q^n \to \mathbb{Z}_q$.
- **Witness:** $A = \{\vec{a}_i\}_{i=1}^m \in \mathbb{Z}_q^{n \times m}, \vec{r} \in \mathbb{Z}_q^m$, and $B = \{\vec{b}_i\}_{i=0}^{m-1}, \vec{s} = (s_0, \ldots, s_{m-1}) \in \mathbb{Z}_q^m$ such that

$$\vec{c}_A = com_{ck}(A; \vec{r}) \qquad c_b = com_{ck}(\vec{B}; \vec{s}) \qquad 0 = \sum_{i=1}^{m} \vec{a}_i * \vec{b}_{i-1}.$$

The $*$ mapping incorporates one of the 3 challenges to project the column vector values of A and B into a singular value and the argument only accepts when this value equals zero.

Single Value Product Argument

a 3-move argument of knowledge [15], which shows that a vector of committed values have a particular single product.

- **Common Reference:** pk, ck.
- **Statement:** $c_a \in \mathbb{G}$ and $b \in \mathbb{Z}_q$.

- **Witness:** $\vec{a} \in \mathbb{Z}_q^n, r \in \mathbb{Z}_q$ such that

$$c_a = com_{ck}(\vec{a}; r) \qquad \text{and} \qquad b = \prod_{i=1}^{n} a_i.$$

4.6 Ristretto Points

A Ristretto Point is a cryptographic construction designed to create a prime-order group from elliptic curves using Decaf[18] on Edwards Curves, enhancing the security and efficiency of cryptographic operations. In the context of mental card games, Ristretto Points enable players to prove knowledge or possession of certain cards without revealing the cards themselves.

Prime-Order Group Properties

Ristretto Points provide a prime-order group that simplifies mathematical operations and ensures predictable, secure behavior in cryptographic protocols. A prime-order group over elliptic curves eliminates issues related to cofactor multiplication, which can complicate the implementation of secure protocols. In mental card games, this property ensures that each card, represented as a Ristretto Point, interacts securely and predictably within the proof system.

Unique Encoding and Decoding

One of the key features of Ristretto Points is their ability to encode and decode points on an elliptic curve in a way that eliminates ambiguities. Each encoded point uniquely corresponds to a single group element, which is critical for maintaining the integrity of the cryptographic proofs. This property ensures that each card in the mental card game, when encoded as a Ristretto Point, has a unique representation, preventing issues such as duplication or misidentification.

Implementation of Choice

We have chosen Dalek Cryptography's crypto tools framework curve25519-dalek[27] as library of choice due to multiple reasons. As the library supports the homomorphic

properties of such group points, and since it is based on elliptic curves it shows additive homomorphism.

The library is also used prominently by the Rust Cryptography community in implementations of all kinds of cryptographic proofs and arguments.(bulletproofs, r1cs etc...). This gives us the advantage of being easily implementable into already existing Ristretto Point systems.

Chapter 5

Implementation

5.1 Commitments and Encryptions

Since Ristretto points are defined in elliptic curve groups, it differs from the usual uses of ElGamal over a finite field where the homomorphism is multiplicative. In elliptic curve groups homomorphism becomes additive[18].

This means when we talk about multiplication of a scalar and a group point we are not talking about cofactor multiplication, we are talking about raising the group point to a scalar power. Because of the same reason we don't have addition between a scalar value and a group point, since group point addition is an automorphism. By having both of our Pedersen commitments and ElGamal encrpytions over the same prime order group we can have seamless interaction between them. Here are some of the terms we will use according to Ristretto implementation before we go into design.

Primitives and Operations

- A Scalar is an open integer which is not commited or hidden in anyway, shown with a lower-case letter.
- A Point is any elliptic curve point, achieved via commitment or encryption, shown with a upper-case letter.
- A Ciphertext is a tuple of points over the ristretto elliptic curve, achieved by ElGamal encrpytion.

These primitives are provided to us via the Ristretto and elgamal-rust[28] libraries.

As for the operations, we have implemented mathematically notated operations such as, vector multiplication, matrix exponentiation as generic traits over data types. More complex operations such as commitment and encryption are abstracted to the Common Reference. Here are the operations we have implemented over these primitives.

Vectors

- $\vec{x}\vec{y} = (x_1 y_1, \ldots, x_n y_n)$ and corresponding exponentiation
- $\vec{x} \cdot \vec{y} = \sum_{i=1}^{n} x_i y_i$

Ciphertexts

- For $\vec{M} = (M_1, \ldots, M_n)$ and $\vec{\rho} = (\rho_1, \ldots, \rho_n)$
  $\mathcal{E}_{pk}(\vec{M}; \vec{\rho}) = (\mathcal{E}_{pk}(M_1; \rho_1), \ldots, \mathcal{E}_{pk}(M_m; \rho_m))$
- For ciphertexts C, $\vec{C}^{\vec{a}} = (C_1, \ldots, C_n)^{a_1, \ldots, a_n)^T} = \prod_{i=1}^{n} C_i^{a_i}$.
- For matrix $A \in \mathbb{Z}_q^{n \times m}$ with column vectors $\vec{a}_1, \ldots \vec{a}_m$
  $\vec{C}^A = (\vec{C}^{\vec{a}_1}, \ldots, \vec{C}^{\vec{a}_m})$.

Commitments

Our Pedersen commitments have their commitment key ck as $n + 1$ random generators and their base prime order group. This allows us to commit to maximum n sized vectors, we can also commit to less then n elements by setting the rest of the entries to zero.

- For matrix $A \in \mathbb{Z}_q^{n \times m}$ with column vectors $\vec{a}_1, \ldots \vec{a}_m$
  $com_{ck}(A; \vec{r}) = (com_{ck}(\vec{a}_1; r_1), \ldots, com_{ck}(\vec{a}_m; r_m))$
- For vector $\vec{a} \in \mathbb{Z}_q^N$ and $\vec{r} \in \mathbb{Z}_q^m$ where $N = mn$
  $com_{ck}(\vec{a}; \vec{r})$ operation allows vectors of values to be treated as matrices if the parameter sizes dont match,an operation over column vectors.

5.2 System Design

Common Reference

First we take a look at the common reference string key needed for commitments and encryptions. The construction of this key is in polynomial time respect to card count since we need $n+1$ random generator points from our group order q for Pedersen commitments. The ElGamal public key is much less of a problem since it's public key setup can be done in constant time. To keep consistency with our group order for our commitments and encryptions we made sure to use the same Ristretto curve.

We also decided to abstract the common reference keys in a seperate struct. This component's associated member functions also shouldered the responsibility for:

- returning random scalars from a crpytographically safe rng,
- commiting scalars, vectors and matrices of scalars,
- encryption of scalar or point values.

Arguments

We have decided to abstract each argument to:

- a struct containing the initial public statements and witnesses,
- member prove function for the construction of the proof arguments,
- member verify function that verifies the initial statement with the given proof.

As such we have 6 "Provers" for each argument defined in Bayer-Groth, for the arguments injected into others the parent argument's constructed proof also contains the child argument's proof. This injection is paralleled in the verification step of the arguments, where the child prover is needed for the child verification so the overall verification becomes complete and secure.

The Random Oracle

For our non-interactive random oracle we chose a cryptographically safe RNG based on the ChaCha stream cipher [29]. Which is an improved variant of Salsa20 [30] cipher

family. The Pseudo-RNG based on this cipher uses the underlying cipher generators to expand a key into a stream of pseudo random bytes.

Which satisfies our definition of a random oracle for the Fiat-Shamir heuristic. During the construction of the proofs, the challenges x, y, z are requested as random scalars from this random oracle.

Provers

While being straight forward mathmetical implementations of the Bayer-Groth algorithms, Rust provides us with security by keeping the initial arguments explicitly immutable, and fast batch operations due to it's efficient iterator and collection access implementations. This property is especially exploited when it comes to matrix operations.

As shown in table 5.1 the diagonalization over a factor of the total number of elements can look pretty straight forward once the index arithmetics are manipulated for efficiency. Since the matrix exponentiation here is purely defined via iterators in $pow()$, the iterative recalling of this function is devoid of indexed memory access overhead.

This type of improvements provided by native Rust is repeated in many places across the library. All of the codebase is open on github[31], accompanied with a comprehensive rust documentation.

Public Transcript

5.3 Main Challenges

The proof constructions become straight forward whence the common references are set. Since they are non-interactive as well the verifier part can be simulated by random challenges. However due to multiple proof compositions, an y challenge we have recieved in the parent proof, should also be the same challenge in the child proof. We have decided that the challenges should be passed down in the proving step since composite proofs are still sequential.

```
let m = //factor of card count(effective column count)
let mu = //Efficiency hyperparameter
let m_ = m / mu;//(Square root of N for optimal)
//Base Ciphertexts for diagonal products
//b_ and tau_ are random value scalar vectors
let mut randomCT: Vec<Ciphertext> = b_.iter()
        .zip(tau_.iter())
        .map(|(_b, _tau)| {
            self.com_ref.encrypt(&EGInp::Scal(*_b)
                                        ,_tau)
        }).collect();

//Diagonal Operations over mu
for i in 1..=mu {
    for j in 1..=mu {
        let k = j + mu - i - 1 ;
        randomCT[k] = (0..=m_ - 1).fold(randomCT[k],
                    |acc, l|
                        acc + self.C_mat[mu * l + i - 1].pow(self.A[mu * l
                    );
            }
        }
```

Table 5.1: Diagonalization over $n X m$ matrix

Chapter 6

Results

We have measured the time it takes for any of the arguments' proof and verification, while also recording the resulting proofs' memory size. Even at first with low card count of $8x8 = 64$ in 6.1 we can see how the optimized version of multi-exponentiation is much more efficient in both size and time performance.

Since the main improvement is the difference between this optimized and base versions we will focus on more test results with different parameters in 6.2.

As we can see from the table the proof size is dependent on the column size(m) of a matrix of cards, rather than N the total card count. This is due to the efficiency of the shuffle argument, performing expensive operations over column vectors instead over single values.

While the speed of the base argument still is affected by the total card count, we can see that is not true for the optimized argument. The optimized argument is not even dependent on the column size but only on the efficency parameter mu. We can see the performance metrics follow the proposed speed up and proof size in Bayer-Groth. A $\mathcal{O}(\sqrt{N})$[1] space improvement when $mu = 1$.

6.1 Implications

We have achieved a safe and verifiable Rust library that constructs shuffle proofs. Since we have used a common scheme for security, Ristretto Points, our arguments and proof construction algorithms could be implemented seamlessly to any Rust proof scheme. Theoretically inspired by Bayer-Groth and practically useful with Rust we provide this

Setup Time	41
Permutation Time	40
m	8
n	8
mu	1
Proof Type	Time(ms) or Size(bytes)
Shuffle Proof Size	10304
Shuffle Proof Time	702
Shuffle Verify Time	93
Opt Mexp Proof Size	1184
Opt Mexp Proof Time	102
Opt Mexp Verify Time	17
Base Mexp Proof Size	8224
Base Mexp Proof Time	551
Base Mexp Verify Time	88
Product Proof Size	1648
Product Proof Time	327
Product Verify Time	75
SV Product Proof Size	1088
SV Product Proof Time	31
SV Product Verify Time	18
Hadamard Proof Size	5280
Hadamard Proof Time	283
Hadamard Verify Time	57
Zero Proof Size	3680
Zero Proof Time	205
Zero Verify Time	47

Table 6.1: Performance Metrics, Time in ms, Size as bytes

Proof Type / Parameters	Base Mexp	Optimized Mexp
N=64; m=8; mu=1	8224/639	1184/119
N=64; m=8; mu=4	8224/631	4064/322
N=128; m=8; mu=1	8480/1220	1184/211
N=128; m=16; mu=1	15904/2298	1184/221
N=128; m=16; mu=4	15904/2198	4064/600
N=1024; m=16; mu=1	17696/17544	1184/1779

Table 6.2: Multi-Exponentiation Trials, Space(bytes)/Time(ms) metrics

easy to use shuffle argument library.

Providing modular implementations of every sub proof, that can be used to create more complex or simpler proofs than shuffling. We are hoping this would provide the cryptography community in Rust with some inspiration.

6.2 Limitations

Assumes ciphertexts are valid. While the Bayer-Groth algorithm suggests it works with any homomorphic property, in our case we have only covered ElGamal encryption and Pedersen commitments over elliptic curves.

Some of the arguments don't have mutual dependencies so the overall Shuffle Argument performance could be better if concurrency were added. However for a proof of concept we have not decided to implement it.

Chapter 7

Conclusion

In conclusion we believe we have provided a library that replicates the results of Bayer-Groth Shuffle Arguments in a pure Rust environment. While Rust's community is growing everyday there is an increasing demand in practical implementations of all kind, which we believe we have contributed to the cryptography discourse with this library.

Chapter 8

Future Work

The library will be improved going forward with certain improvements already underway such as:

- Generalized prove-verify schemes with Rust Traits
- Concurrency between Product Argument and Mexp Argument for a faster Shuffle Argument
- Generalized homomorphism, so any explicitly defined property is supported

Bibliography

[1] S. Bayer and J. Groth, "Efficient zero-knowledge argument for correctness of a shuffle," in *Advances in Cryptology – EUROCRYPT 2012*, D. Pointcheval and T. Johansson, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 263–280.

[2] S. Klabnik and w. c. f. t. R. C. Nichols, Carol. (2024) *The Rust Programming Language Book* [Online]. Avaliable: https://doc.rust-lang.org/book/.

[3] D. L. Chaum, "Untraceable electronic mail, return addresses, and digital pseudonyms," *Commun. ACM*, vol. 24, no. 2, p. 84–90, feb 1981. [Online]. Available: https://doi.org/10.1145/358549.358563

[4] Y. Desmedt and K. Kurosawa, "How to break a practical mix and design a new one," in *Advances in Cryptology — EUROCRYPT 2000*, B. Preneel, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 557–572.

[5] M. Jakobsson, A. Juels, and R. L. Rivest, "Making mix nets robust for electronic voting by randomized partial checking," in *11th USENIX Security Symposium (USENIX Security 02)*. San Francisco, CA: USENIX Association, Aug. 2002. [Online]. Available: https://www.usenix.org/conference/11th-usenix-security-symposium/making-mix-nets-robust-electronic-voting-randomized

[6] K. Peng, C. Boyd, E. Dawson, and K. Viswanathan, "A correct, private, and efficient mix network," in *Public Key Cryptography – PKC 2004*, F. Bao, R. Deng, and J. Zhou, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 439–454.

[7] D. Wikström, "The security of a mix-center based on a semantically secure cryptosystem," in *Progress in Cryptology — INDOCRYPT 2002*, A. Menezes and P. Sarkar, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 368–381.

[8] K. Sako and J. Kilian, "Receipt-free mix-type voting scheme," in *Advances in Cryptology — EUROCRYPT '95*, L. C. Guillou and J.-J. Quisquater, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 393–403.

[9] M. Abe, "Universally verifiable mix-net with verification work independent of the number of mix-servers," in *Advances in Cryptology — EUROCRYPT'98*, K. Nyberg, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 437–447.

[10] ——, "Mix-networks on permutation networks," in *Advances in Cryptology - ASIACRYPT'99*, K.-Y. Lam, E. Okamoto, and C. Xing, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 258–273.

[11] M. Abe and F. Hoshino, "Remarks on mix-network based on permutation networks," in *Public Key Cryptography, 4th International Workshop on Practice and Theory in Public Key Cryptography, PKC 2001, Cheju Island, Korea, February 13-15, 2001, Proceedings*, ser. Lecture Notes in Computer Science, vol. 1992. Springer, 2001, pp. 317–324.

[12] J. Furukawa, K. Mori, and K. Sako, *An Implementation of a Mix-Net Based Network Voting Scheme and Its Use in a Private Organization.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 141–154. [Online]. Available: https://doi.org/10.1007/978-3-642-12980-3_8

[13] C. A. Neff, "A verifiable secret shuffle and its application to e-voting," in *Proceedings of the 8th ACM Conference on Computer and Communications Security*, ser. CCS '01. New York, NY, USA: Association for Computing Machinery, 2001, p. 116–125. [Online]. Available: https://doi.org/10.1145/501983.502000

[14] C. Neff, "Verifiable mixing (shuffling) of elgamal pairs," 01 2004.

[15] J. Groth, "A verifiable secret shuffle of homomorphic encryptions," *Journal of Cryptology*, vol. 23, no. 4, pp. 546–579, Oct 2010. [Online]. Available: https://doi.org/10.1007/s00145-010-9067-9

[16] J. Groth and Y. Ishai, "Sub-linear zero-knowledge argument for correctness of a shuffle," in *Advances in Cryptology – EUROCRYPT 2008*, N. Smart, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 379–396.

[17] B. Abdolmaleki, P. Fauzi, T. Krips, and J. Siim, "Shuffle arguments based on subset-checking," Cryptology ePrint Archive, Paper 2024/1056, 2024. [Online]. Available: https://eprint.iacr.org/2024/1056

[18] M. Hamburg, "Decaf: Eliminating cofactors through point compression," Cryptology ePrint Archive, Paper 2015/673, 2015. [Online]. Available: https://eprint.iacr.org/2015/673

[19] N. Mohnblatt, K. Gurkan, A. Novakovic, and C. Chen. (2022) *Mental Poker. A library for mental poker (and other card games). Based on the Barnett-Smart protocol and the Bayer-Groth argument of correct shuffle.* [Online]. Available: https://github.com/geometryxyz/mental-poker.

[20] A. Barnett and N. P. Smart, "Mental poker revisited," in *IMA Conference on Cryptography and Coding*, 2003. [Online]. Available: https://api.semanticscholar.org/CorpusID:27461759

[21] arkworks contributors. (2022) *arkworks zkSNARK ecosystem.* Available: https://arkworks.rs.

[22] M. Fragkoulis and N. Tyagi. (2018) *Bayer Groth Mixnet Implementation in C++* [Online]. Available: https://github.com/grnet/bg-mixnet.

[23] A. Dalskov. (2021) *Cryptographic Shuffle ala. Bayer and Groth.* Available: https://github.com/anderspkd/groth-shuffle.

[24] T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," in *Advances in Cryptology*, G. R. Blakley and D. Chaum, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1985, pp. 10–18.

[25] S. Goldwasser, S. Micali, and C. Rackoff, "The knowledge complexity of interactive proof systems," *SIAM Journal on Computing*, vol. 18, no. 1, pp. 186–208, 1989. [Online]. Available: https://doi.org/10.1137/0218012

[26] A. Fiat and A. Shamir, "How to prove yourself: Practical solutions to identification and signature problems," in *Advances in Cryptology — CRYPTO' 86*, A. M. Odlyzko, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, pp. 186–194.

[27] D. Cryptography. (2024) *curve25519-dalek A pure-Rust implementation of group operations on Ristretto and Curve25519.* [Online]. Available: https://docs.rs/curve25519-dalek/latest/curve25519_dalek/.

[28] E. McMurtry. (2020) *ElGamal Rust* [Online]. Avaliable: https://docs.rs/rust-elgamal/latest/rust_elgamal/.

[29] D. Bernstein, "Chacha, a variant of salsa20," 01 2008.

[30] D. J. Bernstein, *The Salsa20 Family of Stream Ciphers.* Berlin, Heidelberg: Springer-Verlag, 2008, p. 84–97. [Online]. Available: https://doi.org/10.1007/978-3-540-68351-3_8

[31] A. E. Bulut. (2024) *ElGamal Shuffle Codebase* [Online]. Avaliable: https://github.com/ercembu/elgamal-shuffle.

Appendix A

Appendix A

A.1 ElGamal Encryption

A.1.1 Overview of ElGamal Encryption

ElGamal encryption is a public-key cryptosystem based on the Diffie-Hellman key exchange. It ensures confidentiality and security under the assumption that the discrete logarithm problem is hard.

A.1.2 Key Generation

- **Select Parameters:** A large prime p and a generator g of the multiplicative group $\mathbb{Z}_p^*$.
- **Private Key:** A random integer x such that $1 \leq x < p - 1$.
- **Public Key:** Calculate $y = g^x \pmod{p}$.

The public key is (p, g, y), and the private key is x.

A.1.3 Encryption Process

To encrypt a message m (where $m < p$):

1. Choose a random ephemeral key k such that $1 \leq k < p - 1$.
2. Compute $c_1 = g^k \pmod{p}$.

3. Compute $c_2 = m \cdot y^k \pmod{p}$.

The ciphertext is (c_1, c_2).

A.1.4 Decryption Process

To decrypt the ciphertext (c_1, c_2) using the private key x:

1. Compute $s = c_1^x \pmod{p}$.
2. Compute $m = c_2 \cdot s^{-1} \pmod{p}$, where s^{-1} is the multiplicative inverse of s modulo p.

The result m is the original plaintext message.

A.1.5 Homomorphic Properties

ElGamal encryption has a multiplicative homomorphism property. Given two ciphertexts (c_{11}, c_{12}) for message m_1 and (c_{21}, c_{22}) for message m_2, the product of these ciphertexts is a valid encryption of the product $m_1 \cdot m_2$:

$$(c_{11} \cdot c_{21} \pmod{p},\ c_{12} \cdot c_{22} \pmod{p}) = \left(g^{k_1+k_2} \pmod{p},\ (m_1 \cdot m_2) \cdot y^{k_1+k_2} \pmod{p}\right)$$

A.2 Pedersen Commitments

A.2.1 Overview of Pedersen Commitments

Pedersen commitments are a cryptographic primitive that provides a way to commit to a value while keeping it hidden and allowing the commitment to be opened later. They are homomorphic, additively hiding, and computationally binding, making them useful in various cryptographic protocols.

A.2.2 Commitment Scheme

Given a cyclic group G of prime order q with a generator g and another element h where the discrete logarithm of h with respect to g is unknown:

- **Commitment:** To commit to a value m, choose a random $r \in \mathbb{Z}_q$ and compute the commitment as:

$$C = g^m \cdot h^r \pmod{q}$$

  Here, C is the commitment, m is the message, and r is the random blinding factor.

A.2.3 Opening the Commitment

To open the commitment C, reveal both the value m and the blinding factor r. The verifier can check the validity of the commitment by recomputing C as:

$$C' = g^m \cdot h^r \pmod{q}$$

If $C' = C$, the commitment is valid.

A.2.4 Homomorphic Properties

Pedersen commitments are homomorphic, meaning the product of two commitments is a commitment to the sum of the corresponding values. For two commitments $C_1 = g^{m_1} \cdot h^{r_1}$ and $C_2 = g^{m_2} \cdot h^{r_2}$, the combined commitment is:

$$C_1 \cdot C_2 = g^{m_1+m_2} \cdot h^{r_1+r_2} \pmod{q}$$

This property allows for operations on committed values without revealing them.

A.2.5 Security Properties

Pedersen commitments are:

- **Hiding:** The commitment does not reveal the value m due to the random blinding factor r.
- **Binding:** It is computationally infeasible to find different pairs (m, r) and (m', r') such that $g^m \cdot h^r = g^{m'} \cdot h^{r'}$.

These properties make Pedersen commitments highly secure for use in zero-knowledge proofs and other cryptographic protocols.