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 committed and 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 works over elliptic curve prime order groups to commit and encrypt data to hide the values, 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. With the use of Rust, which has enormous support from the cryptography community, with it's efficiency in runtime, 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

List of Tables

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 [2] 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 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[1], 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.

1.2 Bayer-Groth Shuffle Argument

The Bayer-Groth shuffle argument[2], 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.

1.3 Special honest verifier zero-knowledge argument of knowledge

1.4 The Fiat-Shamir heuristic

1.5 The Schwartz-Zippel lemma

1.6 The Rust Programming Language

Rust[3] 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[3]. 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.

1.7 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, performance, or ease of concurrency. Rust's strict safety guarantees, combined with its ability to produce highly optimized binaries, make it an attractive choice for implementing cryptographic protocols.

Advantages of a Rust Implementation

A Rust implementation of the Bayer-Groth shuffle argument promises several advantages:

- **Memory Safety:** Rust's ownership and borrowing system ensures that the implementation is free from common memory related vulnerabilities.
- **Performance:** Rust's ability to produce low-level, high-performance code makes it suitable for resource-intensive cryptographic operations.

1.8 Objectives

The combination of Rust's safety and performance features with the efficiency and security of the Bayer-Groth shuffle argument suggests a Rust-based implementation would be beneficial for the state of the art.
Such an implementation would not only contribute to the academic and practical un-

derstanding of zero-knowledge proofs but also provide a robust and efficient tool for real-world cryptographic applications.

1.9 Thesis Structure

In the remaining parts of the paper we will first talk about the source material the Bayer-Groth shuffle argument, briefly talking about how they achieved their optimized shuffle argument. Then move onto the existing implementations of the algorithm and their prospects and differences, to ours.

In the Methodology we will start by examining important related concepts such as El Gamal Encryption, Pedersen Commitments and Homomorphic Properties and how we leverage them, continue with the research design, the choices of implementations for these properties.

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

Then 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 we have provided.

Chapter 2

Related Work

2.1 Efficient ZK Argument for Correctness of a Shuffle

In 2012, Bayer and Groth[2] 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 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.
Specifically the algorithm consists of 6 proof schemes, some used as components for others:

- **Shuffle Argument:** The overall argument proving a permutation's correctness on encrypted set of values,
  Constructs Multi-Exp and Product Arguments.

- **Multi-Exponentiation Argument:** The main improvement of the study, argument showing correctness of prime-order group exponentiations.
- **Product Argument:** The argument proving the product of certain commited vector values having a particular product,
  Constructs Hadamard Product and Single Value Product Arguments.
- **Hadamard Product Argument:** Argument that shows for vectors of certain commited values we have a particular result vector for their hadamard product.
  Constructs Zero Argument.
- **Zero Argument:** For two set of commited vectors and a bilinear mapping of two vectors to scalar, argument that shows the resulting operation equals 0. Important for showing equality via difference.
- **Single Value Product Argument:** a 3-move argument of knowledge of commited single values having a particular product.

2.2 Decaf: Eliminating cofactors through point compression

Decaf[4] 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

2.3.1 Mental Poker

Mental Poker[5] 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[6]. They also differ in their choice of cryptography primitives and go with arkworks curve points.[7].

2.3.2 Bayer-Groth Mixnet

A pure C++ implementation[8] 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.

2.3.3 Practical Ad-Hoc Implementations

While there are a few more implementations of the argument available online, they seem to either not be comprehensive enough for a comparison, or small code used as injection for other projects. Which we decided not to mention for brevity's sake.

Chapter 3

Methodology

3.1 Related Concepts

3.1.1 ElGamal Encryption

ElGamal encryption is a public-key cryptosystem based on the Diffie-Hellman key exchange[9]. 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 $y = g^x mod p$.

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:
  - $c_1 = g^k \, mod p$

- $c_2 = m.y^k \, mod p$

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 $s = c_1^x \, mod p$.
- Recovering the original message m by computing $m = c_2 / s \, mod p$.

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.

3.1.2 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:
  $C = g^m . h^r \, mod q$

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 $C' = g^m . h^r \, mod q$ 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.

3.1.3 Homomorphic Property

Homomorphic properties in cryptography refer to the ability of certain operations on encrypted data to produce a valid result that, when decrypted, matches the outcome of performing the same operations on the plaintext.
This characteristic is particularly valuable in scenarios where computations need to be performed on data without exposing it, preserving both privacy and security.
The Shuffle argument utilizes the homomorphic property of both the Elgamal encryption and the Pedersen commitment.

Homomorphism of Pedersen Commitments

Pedersen commitments exhibit a key homomorphic property: the product of two commitments is itself a commitment to the sum of the original messages. Mathematically, if

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

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

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.
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 $m_1 \,.\, m_2$:

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

with k_1 and k_2 being the respective ephemeral keys.
Most of the proof construction algorithms' verification and security is assured by using these properties for our arguments.

3.2 Ristretto Points

A Ristretto Point is a cryptographic construction designed to create a prime-order group from elliptic curves using Decaf 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.

3.2.1 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.

3.2.2 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.

3.2.3 Implementation of Choice

We have chosen Dalek Cryptography's crypto tools framework curve25519-dalek[10] as library of choice due to multiple reasons. As the library supports the homomorphic properties of such group points, it also reduces the dimension of operations via discarding unnecessary operations such as scalar and point multiplication(which would be cofactor multiplication).
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 4

Implementation

4.1 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[4].
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 reserved for "actual" multiplication of the points. 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 our implementation before we get deeper into our definitions:

- the term Scalar for our open integers which are not commited or hidden in anyway,
- the term Point when it comes to any elliptic curve point which can be achieved via:
  - commiting a scalar with a random blinding scalar results in one point,
  - multiplying a scalar with a point (raising the point to a power)
  - adding two points (multiplying two prime order points)
- the term Ciphertext which is a tuple of points over the ristretto elliptic curve, can be achieved via:

- encrypting a scalar which needs to be multiplied with the base point first,
- encrypting a point,

These properties are provided to us via the Ristretto and elgamal-rust[11] libraries.

4.2 System Design

We have decided to abstract each argument to:

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

4.2.1 Common Reference

To keep consistency within each of our arguments for our commitments and encryptions we decided to abstract the Pedersen commitment generators and ElGamal encryption keys in a seperate struct. This components associated member functions also shouldered the responsibility for:

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

Since Bayer-Groth defined their commitment scheme a bit differently, for completeness we also redefined the encryption scheme in the Common Reference.

4.2.2 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. One example we will show of the provers is the multi-exponentiation argument since it has two different prove-verify implementations. The "optimized" version is called as such due to it being the main contributor to the efficiency in message size, while also being the main computation spender.

As shown in table 4.1 the diagonalization over a factor of the total number of elements, can look pretty straight forward once the index arithmatics 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[12], accompanied with a comprehensive rust documentation.

4.3 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 4.1: Diagonalization over $n X m$ matrix

Chapter 5

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 5.1 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 5.2.
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 $\mathcal{O}(\sqrt{N})$[2] space improvement when $mu = 1$.

5.1 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 easy

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 5.1: 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 5.2: Multi-Exponentiation Trials, Space(bytes)/Time(ms) metrics

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.

5.2 Limitations

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 6

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.

6.1 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. 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

[3] 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/.

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

[5] 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.

[6] 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

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

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

[9] 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.

[10] 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/.

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

[12] 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 A.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 A.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 $y = g^x \mod p$.

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

A.1.3 A.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 $c_1 = g^k \mod p$.

3. Compute $c_2 = m \cdot y^k \mod p$.

The ciphertext is (c_1, c_2).

A.1.4 A.4 Decryption Process

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

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

The result m is the original plaintext message.

A.1.5 A.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 $m_1 \cdot m_2$:

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

A.2 Pedersen Commitments

A.2.1 A.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 A.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:

$$C = g^m \cdot h^r \mod q$$

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

A.2.3 A.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:

$$C' = g^m \cdot h^r \mod q$$

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

A.2.4 A.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:

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

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

A.2.5 A.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.