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. Which is important in many real world scenarios(e-voting, mental card games). Due to the plaintexts or data being encrypted, the correctness of a shuffle of commitments is not straight forward to verify. To overcome this problem an honest zero-knowledge verifier has been proposed to verify the correctness of a shuffle of homomorphic encryptions by Bayer and Groth(2012). This argument for correctness combines two separate arguments(Multi-exponentiation Argument, Product Argument) to produce a Shuffle Argument for correctness. The implementation of these arguments are lacking in today's literature of tools. With the use of Rust, which has enormous support from the cryptography community, and with it's efficiency in runtime, we aim to provide an extensive and easy to use zero-knowledge proof system.

Acknowledgements

acknowledgements here.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

1.1.1 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.

1.1.2 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 com-

munication 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.

1.1.3 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.

Safety and Performance

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.

1.1.4 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.

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

1.3 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

Former Literature

2.1 Source Material

2.1.1 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 base proofs 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, argu-

ment showing correctness of prime-order group exponentiations.

- **Product Argument:** The argument proving the product of certain commited values have 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 Existing Solutions

2.2.1 Mental Poker

Mental Poker[4] 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[5]. They also differ in their choice of cryptography primitives and go with arkworks curve points.[6].

2.2.2 Bayer-Groth Mixnet

A pure C++ implementation[7] 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.

2.2.3 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 Important Related Concepts

3.1.1 El Gamal Encryption

3.1.2 Pedersen Commitment

3.1.3 Homomorphic Property

3.2 Research Design

3.2.1 Ristretto Points

A Ristretto Point is a cryptographic construction designed to create a prime-order group from elliptic 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 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 sys-

tem.

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[8] 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 System Design

system design here.

4.2 Implementation Details

implementation details here.

Chapter 5

Results

results here.

Chapter 6

Discussion

6.1 Analysis of Results

analysis of the results here.

6.2 Implications

discussion on implications here.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

conclusion here.

7.2 Future Work

suggestions for future work here.

Bibliography

[1] 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

[2] 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.

[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] 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.

[5] 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

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

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

[8] 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/.

Appendix A

Appendix A

Your appendix content here.