Pseudo-Code Algorithms for Verifiable Re-Encryption Mix-Nets

Rolf Haenni, Philipp Locher, Reto Koenig, and Eric Dubuis

Bern University of Applied Sciences, CH-2501 Biel, Switzerland
{philipp.locher,rolf.haenni, reto.koenig,eric.dubuis}@bfh.ch

Abstract. Implementing the shuffle proof of a verifiable mix-net is one of the most challenging tasks in the implementation of an electronic voting system. For non-specialists, even if they are experienced software developers, this task is nearly impossible to fulfill without spending an enormous amount of resources into studying the necessary cryptographic theory. In this paper, we present one of the existing shuffle proofs in a condensed form and explain all the necessary technical details in corresponding pseudo-code algorithms. The goal of presenting the shuffle proof in this form is to make it accessible to a broader audience and to facilitate its implementation by non-specialists.

1 Introduction

Various cryptographic techniques have been developed to guarantee vote privacy in verifiable electronic voting systems. In practice, processing the list of encrypted votes through a *verifiable re-encryption mix-net* has become the dominating approach in the last couple of years. Various systems developed by academics, practitioners, and vendors are based on this approach, which imitates the physical process of shaking a ballot box containing real votes on paper. While shuffling a list of encryptions is a simple process from a cryptographic point of view, proving that the shuffle has been preformed correctly is a much more difficult task.

For proving the correctness of a cryptographic shuffle, two provably secure proof techniques are dominant in the literature [2, 8] (other methods exist, but many of them have been proven insecure). Due to the complexity of the underlying cryptography, implementing these techniques is almost impossible for non-specialists. Given the manifold subtleties and pitfalls that need to be considered in a cryptographic implementation of such a complexity, even an experienced software developer with a broad cryptographic background may struggle in getting everything right.

Alternatively, system developers may try to delegate the shuffle proof to an existing software library, but such libraries are not available in large numbers. To the best of our knowledge, the only professionally maintained implementation is the *Verificatum Mix-Net* (VMN), which exists since 2008 [10, 11].[1] A few other shuffle proof implementations have been realized, for example as part of the

[1] See http://www.verificatum.com.

UniCrypt library [6], but their intended area of application is mainly academic. In the context of political elections, a practical problem of using third-party libraries in an actual implementation is the complicated system certification process, which gets more difficult with every additional dependency. Having the smallest possible number of dependencies to non-standard libraries may therefore be desirable a strategy for both system developers and election administrations.

In this paper, we focus on the shuffle proof proposed by Wikström and Terelius [8, 9, 11]. In their original publications, the proof is split into an offline and online part and covers options such as restrictions on the set of possible permutations. Their approach also supports various types of objects to shuffle. Such features are interesting from a theoretical point of view, but they are less interesting for practical applications in the area of electronic elections. Tailored pseudo-code algorithms for writing a verifier for VMN are given in [11], but algorithms for generating the proofs are not included in that document.

In this paper, we describe both parts of the shuffle proof in one compact form, while restricting ourselves to the most common use case of single ElGamal encryptions. We summarize the cryptographic theory necessary to understand the core proof mechanisms and provide detailed and comprehensive pseudo-code algorithms for generating and verifying such proofs. The goal of presenting the proof in this form is to make it accessible to a broader audience and to facilitate its implementation by non-specialists. In this way, we hope to facilitate the dissemination of shuffle proofs in electronic voting applications. Even without presenting new results, we think this is an important contribution to the community.

The organization of the paper is as follows. In Section 2, we review the cryptographic background that is necessary to understand the summary of Wikström's proof mechanisms given in Section 3. The pseudo-code algorithms for building a verifiable ElGamal re-encryption mix-net are presented in Section 4. Enough details are given for a software developer with little cryptographic background to implement the proof without accomplishing a profound understanding of the underlying mechanisms. Sections 2 and 3 may therefore be skipped by readers focused in implementing the proof. Section 5 concludes the paper.

2 Cryptographic Background

Let $\mathcal{G}$ be a cyclic group of prime order q, for which the decisional Diffie-Hellman (DDH) assumption is believed to hold. Since q is prime, every $x \in \mathcal{G} \setminus \{1\}$ is a generator. Any such group would be suitable for Wikström's shuffle proof, but here we restrict ourselves to the subgroup $\mathbb{G}_q = \{x^2 \bmod p : 1 \leq x \leq p-1\} \subset \mathbb{Z}_p^*$ of quadratic residues modulo a safe prime $p = 2q+1$. This is the most common choice in practice. When working with $\mathbb{G}_q$, the corresponding prime field $\mathbb{Z}_q = \{0, \ldots, q-1\}$ of integers modulo q plays an important role to perform computations in the exponent.

2.1 ElGamal Encryption

An *ElGamal enryption scheme* is a triple $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ of algorithms, which operate on groups such as $\mathbb{G}_q \subset \mathbb{Z}_p^*$, for which DDH holds [3]. The public parameters of an ElGamal encryption scheme over $\mathbb{G}_q$ are the primes p and q and a generator $g \in \mathbb{G}_q \setminus \{1\}$. A suitable generator can be found by squaring an arbitrary value $x \in \mathbb{Z}_p^* \setminus \{1, p-1\}$, for example $g = 2^2 = 4$ is always a generator of $\mathbb{G}_q$ (except for $p = 5$).

An ElGamal key pair is a tuple $(sk, pk) \leftarrow \mathsf{KeyGen}()$, where $sk \in_R \mathbb{Z}_q$ is the randomly chosen private decryption key and $pk = g^{sk} \in \mathbb{G}_q$ the corresponding public encryption key. If $m \in \mathbb{G}_q$ denotes the plaintext to encrypt, then

$$\mathsf{Enc}_{pk}(m, r) = (m \cdot pk^r, g^r) \in \mathbb{G}_q \times \mathbb{G}_q$$

denotes the ElGamal encryption of m with randomization $r \in_R \mathbb{Z}_q$. Note that the bit length of an encryption $e \leftarrow \mathsf{Enc}_{pk}(m, r)$ is twice the bit length of p. For a given encryption $e = (a, b)$, the plaintext m can be recovered by using the private decryption key sk to compute

$$m \leftarrow \mathsf{Dec}_{sk}(e) = a \cdot b^{-sk}.$$

For any given key pair $(sk, pk) \leftarrow \mathsf{KeyGen}()$, it is easy to demonstrate that $\mathsf{Dec}_{sk}(\mathsf{Enc}_{pk}(m, r)) = m$ holds for all $m \in \mathbb{G}_q$ and $r \in \mathbb{Z}_q$.

The ElGamal encryption scheme is IND-CPA secure under the DDH assumption and homomorphic with respect to multiplication. Therefore, component-wise multiplication of two ciphertexts yields an encryption of the product of respective plaintexts:

$$\mathsf{Enc}_{pk}(m_1, r_1) \cdot \mathsf{Enc}_{pk}(m_2, r_2) = \mathsf{Enc}_{pk}(m_1 m_2, r_1 + r_2).$$

In a homomorphic encryption scheme like ElGamal, a given encryption $e \leftarrow \mathsf{Enc}_{pk}(m, r)$ can be *re-encrypted* by multiplying e with an encryption of the neutral element 1. The resulting re-encryption of e,

$$\mathsf{ReEnc}_{pk}(e, r') = e \cdot \mathsf{Enc}_{pk}(1, r') = \mathsf{Enc}_{pk}(m, r + r'),$$

is clearly an encryption of m with a fresh randomization $r + r'$.

2.2 Pedersen Commitments

The (extended) *Pedersen commitment scheme* is based on a cyclic group for which the discrete logarithm (DL) assumption holds. In this document, we use the same subgroup $\mathbb{G}_q \subset \mathbb{Z}_p^*$ of integers modulo $p = 2q + 1$ as in the ElGamal encryption scheme. Let $g, h_1, \ldots, h_N \in \mathbb{G}_q \setminus \{1\}$ be independent generators of $\mathbb{G}_q$, which means that their relative logarithms are provably not known to anyone.

The Pedersen commitment scheme consists of two deterministic algorithms, one for computing a commitment

$$\mathsf{Com}(\mathbf{m}, r) = g^r \prod_{i=1}^{N} h_i^{m_i} \in \mathbb{G}_q$$

to N messages $\mathbf{m} = (m_1, \ldots, m_N) \in \mathbb{Z}_q^n$ with randomization $r \in_R \mathbb{Z}_q$, and one for checking the validity of $c \leftarrow \mathsf{Com}(\mathbf{m}, r)$ when $\mathbf{m}$ and r are revealed (which we do not require in this paper). In the special case of a single message m, we write $\mathsf{Com}(m, r) = g^r h^m$ using a second generator h independent from g. The Pedersen commitment scheme is perfectly hiding and computationally binding under the DL assumption.

In Wikström's shuffle proof, we also require commitments to permutations $\psi : \{1, \ldots, N\} \rightarrow \{1, \ldots, N\}$. Let $\mathbf{B}_\psi = (b_{ij})_{N \times N}$ be the *permutation matrix* of ψ, which consists of bits

$$b_{ij} = \begin{cases} 1, & \text{if } \psi(i) = j, \\ 0, & \text{otherwise.} \end{cases}$$

Note that in each row and each column in $\mathbf{B}_\psi$, exactly one bit is set to 1. If $\mathbf{b}_j = (b_{1,j}, \ldots, b_{N,j})$ denotes the j-th column of $\mathbf{B}_\psi$, then

$$\mathsf{Com}(\mathbf{b}_j, r_j) = g^{r_j} \prod_{i=1}^{N} h_i^{b_{ij}} = g^{r_j} h_i, \text{ for } i = \psi^{-1}(j),$$

is a commitment to $\mathbf{b}_j$ with randomization r_j. By computing such commitments to all columns, we obtain a *permutation commitment*

$$\mathsf{Com}(\psi, \mathbf{r}) = (\mathsf{Com}(\mathbf{b}_1, r_1), \ldots, \mathsf{Com}(\mathbf{b}_N, r_N))$$

to ψ with randomizations $\mathbf{r} = (r_1, \ldots, r_N)$. Note that the size of such a $\mathbf{c} \leftarrow \mathsf{Com}(\psi, \mathbf{r})$ is $O(N)$.

2.3 Non-Interactive Preimage Proofs

Non-interactive zero-knowledge proofs of knowledge are important building blocks in cryptographic protocol design. In a non-interactive *preimage proof*

$$\mathit{NIZKP}[(x) : y = \phi(x)]$$

for a one-way group homomorphism $\phi : X \rightarrow Y$, the prover proves knowledge of a secret preimage $x = \phi^{-1}(y) \in X$ for a public value $y \in Y$ without revealing anything about x [7].

The most common construction of a non-interactive preimage proof results from combining the so-called Σ-protocol with the Fiat-Shamir heuristic [4]. Generating a preimage proof $(t, s) \leftarrow \mathsf{GenProof}_\phi(x, y)$ for ϕ consists of picking a random value $w \in_R X$ and computing a commitment $t = \phi(w) \in Y$, a challenge $c = \mathsf{Hash}(y, t)$, and a response $s = w + c \cdot x \in X$. Verifying a proof includes computing $c = \mathsf{Hash}(y, t)$ and checking $t = y^{-c} \cdot \phi(s)$. For a given proof $\pi = (t, s)$, this process is denoted by $b \leftarrow \mathsf{CheckProof}_\phi(\pi, y)$, where $b \in \{0, 1\}$ indicates if the proof is valid or not. Clearly, we have

$$\mathsf{CheckProof}_\phi(\mathsf{GenProof}_\phi(x, y), y) = 1$$

for all $x \in X$ and $y = \phi(x) \in Y$. Proofs constructed in this way are perfect zero-knowledge in the random oracle model, which in practice is approximated with the use of a collision-resistant hash function.

3 Summary of Wikström's Shuffle Proof

A *cryptographic shuffle* of a list $\mathbf{e} = (e_1, \dots, e_N)$ of ElGamal encryptions $e_i \leftarrow \mathsf{Enc}_{pk}(m_i, r_i)$ is another list of ElGamal encryptions $\mathbf{e}' = (e'_1, \dots, e'_N)$, which contains the same plaintexts m_i in permuted order. Such a shuffle can be generated by selecting a random permutation $\psi : \{1, \dots, N\} \to \{1, \dots, N\}$ from the set Ψ_N of all such permutations (e.g., using Knuth's shuffle algorithm [5]) and by computing re-encryptions $e'_i \leftarrow \mathsf{ReEnc}_{pk}(e_j, r'_j)$ for $j = \psi(i)$. We write

$$\mathbf{e}' \leftarrow \mathsf{Shuffle}_{pk}(\mathbf{e}, \mathbf{r}', \psi)$$

for an algorithm performing this task, where $\mathbf{r}' = (r'_1, \dots, r'_N)$ denotes the randomization used to re-encrypt the input ciphertexts.

Proving the correctness of a cryptographic shuffle can be realized by proving knowledge of ψ and $\mathbf{r}'$, which generate $\mathbf{e}'$ from $\mathbf{e}$ in a cryptographic shuffle:

$$NIZKP[(\psi, \mathbf{r}') : \mathbf{e}' = \mathsf{Shuffle}_{pk}(\mathbf{e}, \mathbf{r}', \psi)].$$

Unfortunately, since $\mathsf{Shuffle}_{pk}$ does not define a homomorphism, we can not apply the standard technique for preimage proofs. Therefore, the strategy of what follows is to find an equivalent formulation using a homomorphism.

The shuffle proof according to Wikström and Terelius consists of two parts, an offline and an online proof. In the offline proof, the prover computes a commitment $c \leftarrow \mathsf{Com}(\psi, \mathbf{r})$ and proves that c is a commitment to a permutation matrix. In the online proof, the prover demonstrates that the committed permutation matrix has been used in the shuffle to obtain $\mathbf{e}'$ from $\mathbf{e}$. The two proofs can be kept separate, but combining them into a single proof results in a slightly more efficient method. Here, we only present the combined version of the two proofs and we restrict ourselves to the case of shuffling ElGamal ciphertexts.

From a top-down perspective, Wikström's shuffle proof can be seen as a two-layer proof consisting of a top layer responsible for preparatory work such as computing the commitment $\mathbf{c} \leftarrow \mathsf{Com}(\psi, \mathbf{r})$ and a bottom layer computing a standard preimage proof.

3.1 Preparatory Work

There are two fundamental ideas behind Wikström's shuffle proof. The first idea is based on a simple theorem that states that if $\mathbf{B}_\psi = (b_{ij})_{N \times N}$ is an N-by-N-matrix over $\mathbb{Z}_q$ and $(x_1, ..., x_N)$ a vector of N independent variables, then $\mathbf{B}_\psi$ is a permutation matrix if and only if $\sum_{j=1}^{N} b_{ij} = 1$, for all $i \in \{1, \dots, N\}$, and $\prod_{i=1}^{N} \sum_{j=1}^{N} b_{ij} x_i = \prod_{i=1}^{N} x_i$. The first condition means that the elements of each row of $\mathbf{B}_\psi$ must sum up to one, while the second condition requires that $\mathbf{B}_\psi$ has exactly one non-zero element in each row.

Based on this theorem, the general proof strategy is to compute a permutation commitment $\mathbf{c} \leftarrow \mathsf{Com}(\psi, \mathbf{r})$ and to construct a zero-knowledge argument that the two conditions of the theorem hold for $\mathbf{B}_\psi$. This implies then that $\mathbf{c}$ is a commitment to a permutation matrix without revealing ψ or $\mathbf{B}_\psi$.

For $\mathbf{c} = (c_1, \dots, c_N)$, $\mathbf{r} = (r_1, \dots, r_N)$, and $\bar{r} = \sum_{j=1}^{N} r_j$, the first condition leads to the following equality:

$$\prod_{j=1}^{N} c_j = \prod_{j=1}^{N} g^{r_j} \prod_{i=1}^{N} h_i^{b_{ij}} = g^{\sum_{j=1}^{N} r_j} \prod_{i=1}^{N} h_i^{\sum_{j=1}^{N} b_{ij}} = g^{\bar{r}} \prod_{i=1}^{N} h_i = \mathsf{Com}(\mathbf{1}, \bar{r}). \quad (1)$$

Similarly, for arbitrary values $\mathbf{u} = (u_1, \dots, u_N) \in \mathbb{Z}_q^N$, $\mathbf{u}' = (u'_1, \dots, u'_N) \in \mathbb{Z}_q^N$, with $u'_i = \sum_{j=1}^{N} b_{ij} u_j = u_j$ for $j = \psi(i)$, and $\tilde{r} = \sum_{j=1}^{N} r_j u_j$, the second condition leads to two equalities:

$$\prod_{i=1}^{N} u'_i = \prod_{j=1}^{N} u_j, \quad (2)$$

$$\begin{aligned}\prod_{j=1}^{N} c_j^{u_j} &= \prod_{j=1}^{N} (g^{r_j} \prod_{i=1}^{N} h_i^{b_{ij}})^{u_j} = g^{\sum_{j=1}^{N} r_j u_j} \prod_{i=1}^{N} h_i^{\sum_{j=1}^{N} b_{ij} u_j} = g^{\tilde{r}} \prod_{i=1}^{N} h_i^{u'_i} \\ &= \mathsf{Com}(\mathbf{u}', \tilde{r}), \end{aligned} \quad (3)$$

By proving that (1), (2), and (3) hold, and from the independence of the generators, it follows that both conditions of the theorem are true and finally that $\mathbf{c}$ is a commitment to a permutation matrix. In the interactive version of Wikström's proof, the prover obtains $\mathbf{u} = (u_1, \dots, u_N) \in \mathbb{Z}_q^N$ in an initial message from the verifier, but in the non-interactive version we derive these values from the public inputs, for example by computing $u_i \leftarrow \mathsf{Hash}((\mathbf{e}, \mathbf{e}', \mathbf{c}), i)$.

The second fundamental idea of Wikström's proof is based on the homomorphic property of the ElGamal encryption scheme and the following observation for values $\mathbf{u}$ and $\mathbf{u}'$ defined in the same way as above:

$$\begin{aligned}\prod_{i=1}^{N} (e'_i)^{u'_i} &= \prod_{j=1}^{N} \mathsf{ReEnc}_{pk}(e_j, r'_j)^{u_j} = \prod_{j=1}^{N} \mathsf{ReEnc}_{pk}(e_j^{u_j}, r'_j u_j) \\ &= \mathsf{ReEnc}_{pk}(\prod_{j=1}^{N} e_j^{u_j}, \sum_{j=1}^{N} r'_j u_j) = \mathsf{Enc}_{pk}(1, r') \cdot \prod_{j=1}^{N} e_j^{u_j}, \end{aligned} \quad (4)$$

for $r' = \sum_{j=1}^{N} r'_j u_j$. By proving (4), it follows that every e'_i is a re-encryption of e_j for $j = \psi(i)$. This is the desired property of the cryptographic shuffle. By putting (1) to (4) together, the shuffle proof can therefore be rewritten as follows:

$$NIZKP \left[(\bar{r}, \tilde{r}, r', \mathbf{u}') : \begin{array}{c} \prod_{j=1}^{N} c_j = \mathsf{Com}(\mathbf{1}, \bar{r}) \\ \wedge \prod_{i=1}^{N} u'_i = \prod_{j=1}^{N} u_j \\ \wedge \prod_{j=1}^{N} c_j^{u_j} = \mathsf{Com}(\mathbf{u}', \tilde{r}) \\ \wedge \prod_{i=1}^{N} (e'_i)^{u'_i} = \mathsf{Enc}_{pk}(1, r') \cdot \prod_{j=1}^{N} e_j^{u_j} \end{array} \right]. \quad (5)$$

The last step of the preparatory work results from replacing in the above expression the equality of products, $\prod_{i=1}^{N} u'_i = \prod_{j=1}^{N} u_j$, by an equivalent expression based on a chained list $\hat{\mathbf{c}} = \{\hat{c}_1, \dots, \hat{c}_N\}$ of Pedersen commitments with different generators. For $\hat{c}_0 = h$ and random values $\hat{\mathbf{r}} = (\hat{r}_1, \dots, \hat{r}_N) \in \mathbb{Z}_q^N$, we define $\hat{c}_i = g^{\hat{r}_i} \hat{c}_{i-1}^{u'_i}$, which leads to $\hat{c}_N = \mathsf{Com}(u, \hat{r})$ for $u = \prod_{i=1}^N u_i$ and

$$\hat{r} = \sum_{i=1}^{N} \hat{r}_i \prod_{j=i+1}^{N} u'_j.$$

Applying this replacement leads to the following final result, on which the proof construction is based:

$$NIZKP \left[(\bar{r}, \hat{r}, \tilde{r}, r', \hat{\mathbf{r}}, \mathbf{u}') : \begin{array}{c} \prod_{j=1}^N c_j = \mathsf{Com}(\mathbf{1}, \bar{r}) \\ \wedge\, \hat{c}_N = \mathsf{Com}(u, \hat{r}) \wedge \left[\bigwedge_{i=1}^N (\hat{c}_i = g^{\hat{r}_i} \hat{c}_{i-1}^{u'_i}) \right] \\ \wedge \prod_{j=1}^N c_j^{u_j} = \mathsf{Com}(\mathbf{u}', \tilde{r}) \\ \wedge \prod_{i=1}^N (e'_i)^{u'_i} = \mathsf{Enc}_{pk}(1, r') \cdot \prod_{j=1}^N e_j^{u_j} \end{array} \right]. \qquad (6)$$

To summarize the preparatory work for the proof generation, we give a list of all necessary computations:

- Pick $\mathbf{r} = (r_1, \dots, r_N) \in_R \mathbb{Z}_q^N$ and compute $\mathbf{c} \leftarrow \mathsf{Com}(\psi, \mathbf{r})$.
- For $i = 1, \dots, N$, compute $u_i \leftarrow \mathsf{Hash}((\mathbf{e}, \mathbf{e}', \mathbf{c}), i)$, let $u'_i = u_{\psi(i)}$, pick $\hat{r}_i \in_R \mathbb{Z}_q$, and compute $\hat{c}_i = g^{\hat{r}_i} \hat{c}_{i-1}^{u'_i}$.
- Let $\hat{\mathbf{r}} = (\hat{r}_1, \dots, \hat{r}_N)$ and $\hat{\mathbf{c}} = (\hat{c}_1, \dots, \hat{c}_N)$.
- Compute $\bar{r} = \sum_{j=1}^N r_j$, $\hat{r} = \sum_{i=1}^N \hat{r}_i \prod_{j=i+1}^N u'_j$, $\tilde{r} = \sum_{j=1}^N r_j u_j$, and $r' = \sum_{j=1}^N r'_j u_j$.

Note that $\hat{r}$ can be computed in linear time by generating the values $\prod_{j=i+1}^N u'_j$ in an incremental manner by looping backwards over $j = N, \dots, 1$.

3.2 Preimage Proof

By rearranging all public values to the left-hand side and all secret values to the right-hand side of each equation, we can derive a homomorphic one-way function from the final expression of the previous subsection. In this way, we obtain the homomorphic function

$$\begin{aligned} &\phi(x_1, x_2, x_3, x_4, \hat{\mathbf{x}}, \mathbf{x}') \\ &\quad = (g^{x_1}, g^{x_2}, \mathsf{Com}(\mathbf{x}', x_3), \mathsf{ReEnc}_{pk}(\prod_{i=1}^{N} (e'_i)^{x'_i}, -x_4), (g^{\hat{x}_1} \hat{c}_0^{x'_1}, \dots, g^{\hat{x}_N} \hat{c}_{N-1}^{x'_N})), \qquad (7) \end{aligned}$$

which maps inputs $(x_1, x_2, x_3, x_4, \hat{\mathbf{x}}, \mathbf{x}') \in X$ of length $2N + 4$ into outputs

$$(y_1, y_2, y_3, y_4, \hat{\mathbf{y}}) = \phi(x_1, x_2, x_3, x_4, \hat{\mathbf{x}}, \mathbf{x}') \in Y$$

of length $N + 5$, i.e., $X = \mathbb{Z}_q^4 \times \mathbb{Z}_q^N \times \mathbb{Z}_q^N$ is the domain and $Y = \mathbb{G}_q^3 \times \mathbb{G}_q^2 \times \mathbb{G}_q^N$ the co-domain of ϕ. Note that we slightly modified the order of the five sub-functions of ϕ for better readability. By applying this function to the secret values

$(\bar{r}, \hat{r}, \tilde{r}, r', \hat{\mathbf{r}}, \mathbf{u}')$, we get a tuple of public values,

$$(\bar{c}, \hat{c}, \tilde{c}, e', \hat{\mathbf{c}}) = (\frac{\prod_{j=1}^{N} c_j}{\prod_{j=1}^{N} h_j}, \frac{\hat{c}_N}{h^u}, \prod_{j=1}^{N} c_j^{u_j}, \prod_{j=1}^{N} e_j^{u_j}, (\hat{c}_1, \dots, \hat{c}_N)), \tag{8}$$

which can be derived from the public values $\mathbf{e}$, $\mathbf{e}'$, $\mathbf{c}$, $\hat{\mathbf{c}}$, and pk (and from $\mathbf{u}$, which is derived from $\mathbf{e}$, $\mathbf{e}'$, and $\mathbf{c}$).

To summarize, we have a homomorphic one-way function $\phi : X \to Y$, secret values $x = (\bar{r}, \hat{r}, \tilde{r}, r', \hat{\mathbf{r}}, \mathbf{u}') \in X$, and public values $y = (\bar{c}, \hat{c}, \tilde{c}, e', \hat{\mathbf{c}}) = \phi(x) \in Y$. We can therefore generate a non-interactive preimage proof

$$NIZKP \left[(\bar{r}, \hat{r}, \tilde{r}, r', \hat{\mathbf{r}}, \mathbf{u}') : \begin{array}{r} \bar{c} = g^{\bar{r}} \wedge \hat{c} = g^{\hat{r}} \wedge \tilde{c} = \mathsf{Com}(\mathbf{u}', \tilde{r}) \\ \wedge\ e' = \mathsf{ReEnc}_{pk}(\prod_{i=1}^{N} (e_i')^{u_i'}, -r') \\ \wedge \left[\bigwedge_{i=1}^{N} (\hat{c}_i = g^{\hat{r}_i} \hat{c}_{i-1}^{u_i'}) \right] \end{array} \right], \tag{9}$$

using the standard procedure from Section 2.3. The result of such a proof generation, $(t, s) \leftarrow \mathsf{GenProof}_\phi(x, y)$, consists of two values $t = \phi(w) \in Y$ of length $N + 5$ and $s = \omega + c \cdot x \in X$ of length $2N + 4$, which we obtain from picking $w \in_R X$ (of length $2N + 4$) and computing $c = \mathsf{Hash}(y, t)$. Alternatively, a different $c = \mathsf{Hash}(y', t)$ could be derived directly from the public values $y' = (\mathbf{e}, \mathbf{e}', \mathbf{c}, \hat{\mathbf{c}}, pk)$, which has the advantage that $y = (\bar{c}, \hat{c}, \tilde{c}, e', \hat{\mathbf{c}})$ needs not to be computed explicitly during the proof generation.

This preimage proof, together with the two lists of commitments $\mathbf{c}$ and $\hat{\mathbf{c}}$, leads to the desired non-interactive shuffle proof $NIZKP[(\psi, \mathbf{r}') : \mathbf{e}' = \mathsf{Shuffle}_{pk}(\mathbf{e}, \mathbf{r}', \psi)]$. We denote the generation and verification of a such proof $\pi = (t, s, \mathbf{c}, \hat{\mathbf{c}})$ by

$$\pi \leftarrow \mathsf{GenProof}_{pk}(\mathbf{e}, \mathbf{e}', \mathbf{r}', \psi)$$
$$b \leftarrow \mathsf{CheckProof}_{pk}(\pi, \mathbf{e}, \mathbf{e}').$$

respectively. Corresponding algorithms are depicted in Algorithm 4.3 and Algorithm 4.6. Note that generating the proof requires $7N + 4$ and verifying the proof $9N + 11$ modular exponentiations in $\mathbb{G}_q$. The proof itself consists of $5N + 9$ elements ($2N + 4$ elements from $\mathbb{Z}_q$ and $3N + 5$ elements from $\mathbb{G}_q$).

4 Pseudo-Code Algorithms

Based on the background information given in the previous two sections, we will now transform the mathematical description of the proof into detailed pseudo-code algorithms. This will give us an even closer look at how the shuffle proof works. Algorithms 4.1, 4.3 and 4.6 are the three main algorithms for performing the shuffle, generating the proof, and checking the validity of a proof, respectively. We decided to give almost monolithic descriptions for each of these algorithms with little dependencies to sub-routines.

There are some public parameters, which we do not pass explicitly as arguments to each algorithm: the prime modulo p of the group $\mathbb{G}_q \subset \mathbb{Z}_p^*$, the group order $q = (p-1)/2$, the main independent group generators g and h, and N other independent generators $h_1, \dots, h_N$. We do not give algorithms for finding suitable group parameters or give recommendations about their sizes, we simply assume that they are publicly known.[2] For a deterministic algorithm that generates an arbitrary number of independent generators, we refer to the NIST standard FIPS PUB 186-4 [1, Appendix A.2.3]. The deterministic nature of this algorithm enables the independence of the generators to be publicly verified.

Most numeric calculations in the given algorithms are either performed modulo p or modulo q. For maximal clarity, we indicate the modulus in each individual case. We suppose that efficient algorithms are available for computing modular exponentiations $x^y \bmod p$ and modular inverses $x^{-1} \bmod p$. Divisions $x/y \bmod p$ are handled as $xy^{-1} \bmod p$ and exponentiations $x^{-y} \bmod p$ with negative exponents as $(x^{-1})^y \bmod p$ or $(x^y)^{-1} \bmod p$. We also assume that readers are familiar with mathematical notations for sums and products, such that implementing expressions like $\sum_{i=1}^N x_i$ or $\prod_{i=1}^N x_i$ is straightforward.

An important precondition for every algorithm is the validity of the input parameters, for example that an ElGamal encryption $e = (a, b)$ is an element of $\mathbb{G}_q \times \mathbb{G}_q$ or that given input lists are of equal length. We specify all preconditions for every algorithm, but we do not give explicit code to perform corresponding checks. However, as many attacks on mix-nets are based on infiltrating invalid parameters, we stress the importance of conducting such checks in an actual implementation. For testing group membership $x \in \mathbb{G}_q$ of quadratic residues modulo p, we refer to algorithms for computing the Jacobi symbol $\left(\frac{x}{p}\right)$, for example in [1, pp. 76–77].

Finally, we assume that efficient and secure algorithms are available for computing cryptographic hash values $h \leftarrow \mathsf{Hash}(x)$ of arbitrary mathematical objects and for picking uniform elements $r \in_R \mathbb{Z}_q$ (or more generally $r \in_R [a, b]$). Writing such algorithms is a difficult problem on its own, which we cannot address here. However, such algorithms are usually available in standard cryptographic libraries of modern programming languages.

4.1 Generating the Shuffle

The input of a cryptographic shuffle $\mathbf{e}' \leftarrow \mathsf{Shuffle}_{pk}(\mathbf{e}, \mathbf{r}', \psi)$ is a list of $\mathbf{e} = (e_1, \dots, e_N)$ encryptions e_i, in our case ElGamal encryptions $e_i = (a_i, b_i) \in \mathbb{G}_q^2$, which need to be re-encrypted and permuted. In Algorithm 4.1, we describe this procedure, which includes picking a random permutation $\psi = (j_1, \dots, j_N) \in \mathit{\Psi}_N$ (line 2) and a list $\mathbf{r}' = (r'_1, \dots, r'_N)$ of re-encryption randomizations (line 4). The re-encryptions are computed in a loop over all input encryptions (lines 3–7) and permuted by re-arranging them according to ψ (line 8). The random values ψ and $\mathbf{r}'$ are returned together with $\mathbf{e}'$, because they are required as secret inputs to the proof generation.

The above shuffling algorithm calls one sub-routine for generating a random permutation $\psi \in \mathit{\Psi}_N$. We present a procedure for this problem in Algo-

[2] See https://www.keylength.com for current recommendations.

```
Algorithm: GenShuffle($\mathbf{e}, pk$)
Input: ElGamal encryptions $\mathbf{e} = (e_1, \dots, e_N)$, $e_i = (a_i, b_i) \in \mathbb{G}_q^2$
       Encryption key $pk \in \mathbb{G}_q$
$\psi \leftarrow$ GenPermutation($N$)                // $\psi = (j_1, \dots, j_N)$, see Algorithm 4.2
for $i = 1, \dots, N$ do
    $r'_i \in_R \mathbb{Z}_q$
    $a'_i \leftarrow a_i \, pk^{r'_i} \bmod p$
    $b'_i \leftarrow b_i \, g^{r'_i} \bmod p$
    $e'_i \leftarrow (a'_i, b'_i)$
$\mathbf{e}' \leftarrow (e'_{j_1}, \dots, e'_{j_N})$
$\mathbf{r}' \leftarrow (r'_1, \dots, r'_N)$
return $(\mathbf{e}', \mathbf{r}', \psi)$                // $\mathbf{e}' \in (\mathbb{G}_q^2)^N$, $\mathbf{r}' \in \mathbb{Z}_q^N$, $\psi \in \Psi_N$
```

Algorithm 4.1: Generates a random permutation $\psi \in \Psi_N$ and uses it to shuffle a given list $\mathbf{e}$ of ElGamal encryptions into a shuffled list $\mathbf{e}'$.

rithm 4.2, which is essentially Knuth's shuffle algorithm [5, pp. 139–140]. The auxiliary variable I is an integer array of size N, which is addressed with indices $i, k \in \{1, \dots, N\}$. After initializing the array with integers $1, \dots, N$ (line 2), N swap operations are performed with indices chosen at random (lines 3–6). Knuth's algorithm is proven to implement a uniform distribution over all possible permutations.

```
1 Algorithm: GenPermutation($N$)
  Input: Permutation size $N \in \mathbb{N}$
2 $I \leftarrow \langle 1, \dots, N \rangle$
3 for $i = 1, \dots, N$ do
4     $k \in_R \{i, \dots, N\}$
5     $j_i \leftarrow I[k]$
6     $I[k] \leftarrow I[i]$
7 $\psi \leftarrow (j_1, \dots, j_N)$
8 return $\psi$                                   // $\psi \in \Psi_N$
```

Algorithm 4.2: Generates a random permutation $\psi \in \Psi_N$ following Knuth's shuffle algorithm.

4.2 Generating the Shuffle Proof

The mathematical description of the shuffle proof in Section 3 is the basis for procedure shown in Algorithm 4.3. The core of the algorithm is the preimage proof specified in (9), which requires some preparatory work. The first preparatory step is the generation of the permutation commitment $\mathbf{c}$ (line 2), which we delegate to a separate subroutine. The second preparatory step is the computation of values $\mathbf{u}$, which are derived from the public inputs $\mathbf{e}$ and $\mathbf{e}'$ and the permutation commitment $\mathbf{c}$, and which are permuted according to ψ into $\mathbf{u}'$ (lines 3–6). The next preparatory step is the computation of the commitment chain $\hat{\mathbf{c}}$ in a separate subroutine with $c_0 = h$ as initial value (line 7). Finally, the last preparatory step consists in computing the secret inputs $\bar{r}$, $\hat{r}$, $\tilde{r}$, and r' for the preimage proof (lines 8–14).

The implementation of the preimage proof starts on line 15, where $2N + 4$ values $w_i, \hat{w}_i, w'_i \in \mathbb{Z}_q$ are selected at random (lines 15–18). They are needed for the computation of the $N + 5$ commitments $t_i, \hat{t}_i \in \mathbb{G}_q$ (lines 19–25), which follows the definition of the homomorphic one-way function ϕ as specified in (7). The commitments and all public values are then used to compute the challenge c (lines 26–27), which determines to $2N + 4$ responses $s_i, \hat{s}_i, s'_i \in \mathbb{Z}_q$ (lines 28–33). The algorithm ends with returning the tuples t and s of all commitments and responses, respectively, together with the permutation commitment $\mathbf{c}$ and the commitment chain $\hat{\mathbf{c}}$.

Each of the two auxiliary algorithms called during the proof generation returns a list of Pedersen commitments. In the case of Algorithm 4.4, the return value is actually a commitment to the permutation ψ. The procedure for computing such a permutation commitment is described in Section 2.2. The return value of Algorithm 4.5 consists of Pedersen commitments that are linked over one of the two generators. The role of this commitment chain has been discussed in Section 3 and does not require further explanations.

4.3 Verifying the Shuffle Proof

A shuffle proof $\pi = (t, c, \mathbf{c}, \hat{\mathbf{c}})$ generated by Algorithm 4.3 consists of the result (t, s) of the preimage proof and the two lists of commitments $\mathbf{c}$ and $\hat{\mathbf{c}}$ obtained as a result of several preparatory steps. Algorithm 4.6 shows the necessary steps of checking the validity of such a proof. The additional input values of this algorithm are two lists of encryptions $\mathbf{e}$ and $\mathbf{e}'$ and the public key pk.

The first preparatory step in the algorithm is the derivation of the values $\mathbf{u}$ from the inputs $\mathbf{e}$, $\mathbf{e}'$, and $\mathbf{c}$ (lines 2–3). The second preparatory step is the computation of the public values $\bar{c}$, $\hat{c}$, $\tilde{c}$, and $e' = (a', b')$ (lines 5–9) according to their definition given in (8). Nothing else is needed to perform the verification of the preimage proof (t, s) according to the standard procedure described in Section 2.3. That is, the challenge c can be derived from the public values $\mathbf{e}$, $\mathbf{e}'$, $\mathbf{c}$, $\hat{\mathbf{c}}$, and pk (line 11), which then leads to $N + 5$ values $t'_i, \hat{t}'_i \in \mathbb{G}_q$ by applying the one-way function ϕ to s (lines 12–17). The resulting values are compared to respective values $t_i, \hat{t}_i \in \mathbb{G}_q$ included in the proof, and if all values match, the proof is valid (line 18).

5 Conclusion

In this paper, we have given a compact summary of Wikström's shuffle proof and a detailed description of the proof generation and verification processes in form

```
Algorithm: $\mathsf{GenProof}(\mathbf{e}, \mathbf{e}', \mathbf{r}', \psi, pk)$
Input: ElGamal encryptions $\mathbf{e} = (e_1, \dots, e_N)$, $e_i = (a_i, b_i) \in \mathbb{G}_q^2$
       Shuffled ElGamal encryptions $\mathbf{e}' = (e'_1, \dots, e'_N)$, $e'_i = (a'_i, b'_i) \in \mathbb{G}_q^2$
       Re-encryption randomizations $\mathbf{r}' = (r'_1, \dots, r'_N)$, $r'_i \in \mathbb{Z}_q$
       Permutation $\psi = (j_1, \dots, j_N) \in \Psi_N$
       Encryption key $pk \in \mathbb{G}_q$
$(\mathbf{c}, \mathbf{r}) \leftarrow \mathsf{GenCommitment}(\psi)$          // $\mathbf{c} = (c_1, \dots, c_N)$, $\mathbf{r} = (r_1, \dots, r_N)$
for $i = 1, \dots, N$ do
    $u_i \leftarrow \mathsf{Hash}((\mathbf{e}, \mathbf{e}', \mathbf{c}), i)$
    $u'_i \leftarrow u_{j_i}$
$\mathbf{u} \leftarrow (u_1, \dots, u_N)$, $\mathbf{u}' \leftarrow (u'_1, \dots, u'_N)$
$(\hat{\mathbf{c}}, \hat{\mathbf{r}}) \leftarrow \mathsf{GenCommitmentChain}(h, \mathbf{u}')$          // $\hat{\mathbf{c}} = (\hat{c}_1, \dots, \hat{c}_N)$, $\hat{\mathbf{r}} = (\hat{r}_1, \dots, \hat{r}_N)$
$\bar{r} \leftarrow \sum_{i=1}^{N} r_i \bmod q$
$v_N \leftarrow 1$
for $i = N-1, \dots, 1$ do
    $v_i \leftarrow u'_{i+1} v_{i+1} \bmod q$
$\hat{r} \leftarrow \sum_{i=1}^{N} \hat{r}_i v_i \bmod q$
$\tilde{r} \leftarrow \sum_{i=1}^{N} r_i u_i \bmod q$
$r' \leftarrow \sum_{i=1}^{N} r'_i u_i \bmod q$
for $i = 1, \dots, 4$ do
    $\omega_i \in_R \mathbb{Z}_q$
for $i = 1, \dots, N$ do
    $\hat{\omega}_i \in_R \mathbb{Z}_q$, $\omega'_i \in_R \mathbb{Z}_q$
$t_1 \leftarrow g^{\omega_1} \bmod p$
$t_2 \leftarrow g^{\omega_2} \bmod p$
$t_3 \leftarrow g^{\omega_3} \prod_{i=1}^{N} h_i^{\omega'_i} \bmod p$
$(t_{4,1}, t_{4,2}) \leftarrow (pk^{-\omega_4} \prod_{i=1}^{N} (a'_i)^{\omega'_i} \bmod p, g^{-\omega_4} \prod_{i=1}^{N} (b'_i)^{\omega'_i} \bmod p)$
$\hat{c}_0 \leftarrow h$
for $i = 1, \dots, N$ do
    $\hat{t}_i \leftarrow g^{\hat{\omega}_i} \hat{c}_{i-1}^{\omega'_i} \bmod p$
$y \leftarrow (\mathbf{e}, \mathbf{e}', \mathbf{c}, \hat{\mathbf{c}}, pk)$, $t \leftarrow (t_1, t_2, t_3, (t_{4,1}, t_{4,2}), (\hat{t}_1, \dots, \hat{t}_N))$
$c \leftarrow \mathsf{Hash}(y, t)$
$s_1 \leftarrow \omega_1 + c \cdot \bar{r} \bmod q$
$s_2 \leftarrow \omega_2 + c \cdot \hat{r} \bmod q$
$s_3 \leftarrow \omega_3 + c \cdot \tilde{r} \bmod q$
$s_4 \leftarrow \omega_4 + c \cdot r' \bmod q$
for $i = 1, \dots, N$ do
    $\hat{s}_i \leftarrow \hat{\omega}_i + c \cdot \hat{r}_i \bmod q$, $s'_i \leftarrow \omega'_i + c \cdot u'_i \bmod q$
$s \leftarrow (s_1, s_2, s_3, s_4, (\hat{s}_1, \dots, \hat{s}_N), (s'_1, \dots, s'_N))$
$\pi \leftarrow (t, s, \mathbf{c}, \hat{\mathbf{c}})$
return $\pi$          // $\pi \in (\mathbb{G}_q^3 \times \mathbb{G}_q^2 \times \mathbb{G}_q^N) \times (\mathbb{Z}_q^4 \times \mathbb{Z}_q^N \times \mathbb{Z}_q^N) \times \mathbb{G}_q^N \times \mathbb{G}_q^N$
```

Algorithm 4.3: Generates a proof of shuffle for given ElGamal encryptions $\mathbf{e}$ and $\mathbf{e}'$ according to Wikström's method.

```
Algorithm: $\mathsf{GenCommitment}(\psi)$
Input: Permutation $\psi = (j_1, \dots, j_N) \in \Psi_N$
for $i = 1, \dots, N$ do
    $r_{j_i} \in_R \mathbb{Z}_q$
    $c_{j_i} \leftarrow g^{r_{j_i}} h_i \bmod p$
$\mathbf{c} \leftarrow (c_1, \dots, c_N)$
$\mathbf{r} \leftarrow (r_1, \dots, r_N)$
return $(\mathbf{c}, \mathbf{r})$                    // $\mathbf{c} \in \mathbb{G}_q^N$, $\mathbf{r} \in \mathbb{Z}_q^N$
```

Algorithm 4.4: Generates a commitment $\mathbf{c} = \mathsf{Com}(\psi, \mathbf{r})$ to a permutation ψ by committing to the columns of the corresponding permutation matrix.

```
Algorithm: $\mathsf{GenCommitmentChain}(c_0, \mathbf{u})$
Input: Initial commitment $c_0 \in \mathbb{G}_q$
       Public challenges $\mathbf{u} = (u_1, \dots, u_N)$, $u_i \in \mathbb{Z}_q$
for $i = 1, \dots, N$ do
    $r_i \in_R \mathbb{Z}_q$
    $c_i \leftarrow g^{r_i} c_{i-1}^{u_i} \bmod p$
$\mathbf{c} \leftarrow (c_1, \dots, c_N)$
$\mathbf{r} \leftarrow (r_1, \dots, r_N)$
return $(\mathbf{c}, \mathbf{r})$                    // $\mathbf{c} \in \mathbb{G}_q^N$, $\mathbf{r} \in \mathbb{Z}_q^N$
```

Algorithm 4.5: Generates a commitment chain $c_0 \rightarrow c_1 \rightarrow \cdots \rightarrow c_N$ relative to a list of public challenges $\mathbf{u}$ and starting with a given commitment c_0.

of pseudo-code algorithms. The level of detail of these algorithms is such that even developers with little background in cryptography can implement to proof by coding carefully line after line. This solves an important problem of system developers in charge of such an implementation. In the past, many of them have struggled when facing the complexity of the underlying cryptography. With this paper at hand, they have now a detailed guideline which they can follow without fully understanding the theory. We expect that a robust implementation for someone that starts from scratch will now be possible within a matter of weeks (instead of months at least).

To verify the above claim, we have given a draft of this paper to an experienced software developer with no special education or experience in implementing cryptographic algorithms. Within approximately four weeks of part-time work, he managed to produce high-quality Java 8 code of everything that is needed for building a verifiable mix-net. Based on the precise description of our paper, he even managed—for a very small group $\mathbb{G}_q$—to calculate a shuffle proof entirely by hand, and his test suite reaches nearly 100% code coverage. Using the native

```
Algorithm: $\mathsf{CheckProof}(\pi, \mathbf{e}, \mathbf{e}', pk)$
Input: Shuffle proof $\pi = (t, s, \mathbf{c}, \hat{\mathbf{c}})$, $t = (t_1, t_2, t_3, (t_{4,1}, t_{4,2}), (\hat{t}_1, \dots, \hat{t}_N))$,
       $s = (s_1, s_2, s_3, s_4, (\hat{s}_1, \dots, \hat{s}_N), (s'_1, \dots, s'_N))$, $\mathbf{c} = (c_1, \dots, c_N)$,
       $\hat{\mathbf{c}} = (\hat{c}_1, \dots, \hat{c}_N)$
       ElGamal encryptions $\mathbf{e} = (e_1, \dots, e_N)$, $e_i = (a_i, b_i) \in \mathbb{G}_q^2$
       Shuffled ElGamal encryptions $\mathbf{e}' = (e'_1, \dots, e'_N)$, $e'_i = (a'_i, b'_i) \in \mathbb{G}_q^2$
       Encryption key $pk \in \mathbb{G}_q$
for $i = 1, \dots, N$ do
    $u_i \leftarrow \mathsf{Hash}((\mathbf{e}, \mathbf{e}', \mathbf{c}), i)$
$\mathbf{u} \leftarrow (u_1, \dots, u_N)$
$\bar{c} \leftarrow \prod_{i=1}^N c_i / \prod_{i=1}^N h_i \bmod p$
$u \leftarrow \prod_{i=1}^N u_i \bmod q$
$\hat{c} \leftarrow \hat{c}_N / h^u \bmod p$
$\tilde{c} \leftarrow \prod_{i=1}^N c_i^{u_i} \bmod p$
$(a', b') \leftarrow (\prod_{i=1}^N a_i^{u_i} \bmod p, \prod_{i=1}^N b_i^{u_i} \bmod p)$
$y \leftarrow (\mathbf{e}, \mathbf{e}', \mathbf{c}, \hat{\mathbf{c}}, pk)$
$c \leftarrow \mathsf{Hash}(y, t)$
$t'_1 \leftarrow \bar{c}^{-c} g^{s_1} \bmod p$
$t'_2 \leftarrow \hat{c}^{-c} g^{s_2} \bmod p$
$t'_3 \leftarrow \tilde{c}^{-c} g^{s_3} \prod_{i=1}^N h_i^{s'_i} \bmod p$
$(t'_{4,1}, t'_{4,2}) \leftarrow ((a')^{-c} pk^{-s_4} \prod_{i=1}^N (a'_i)^{s'_i} \bmod p, (b')^{-c} g^{-s_4} \prod_{i=1}^N (b'_i)^{s'_i} \bmod p)$
for $i = 1, \dots, N$ do
    $\hat{t}'_i \leftarrow \hat{c}_i^{-c} g^{\hat{s}_i} \hat{c}_{i-1}^{s'_i} \bmod p$
return
    $(t_1 = t'_1) \wedge (t_2 = t'_2) \wedge (t_3 = t'_3) \wedge (t_{4,1} = t'_{4,1}) \wedge (t_{4,2} = t'_{4,1}) \wedge \left[\bigwedge_{i=1}^N (\hat{t}_i = \hat{t}'_i)\right]$
```

Algorithm 4.6: Checks the correctness of a shuffle proof π generated by Algorithm 4.3. The public values are the ElGamal encryptions $\mathbf{e}$ and $\mathbf{e}'$ and the public encryption key pk.

GMP library for fast big integer calculations and parallel streams from Java 8 to exploit the power of all available cores, the performance of his code is already well-optimized. This work has been conducted in the context of the CHVote Internet voting project of the State of Geneva in Switzerland. The complete source code will be available soon on a public server. We have also assigned this implementation task to a group of students with little background knowledge in cryptography and security. This is an ongoing project, we expect the results in a couple of months in form of a Bachelor thesis. A goal of this work is to obtain a second independent implementation and to use it for mutual tests.

By compiling the theory of Wikström's shuffle proof into a single paper and by facilitating the implementation of verifiable mix-nets with pseudo-code algorithms, we hope that this paper will help to disperse this technology even further.

References

[1] Digital signature standard (DSS). FIPS PUB 186-4, National Institute of Standards and Technology (NIST) (2013)

[2] Bayer, S., Groth, J.: Efficient zero-knowledge argument for correctness of a shuffle. In: Pointcheval, D., Johansson, T. (eds.) EUROCRYPT'12, 31st Annual International Conference on Theory and Applications of Cryptographic Techniques. pp. 263–280. LNCS 7237, Cambridge, UK (2012)

[3] El Gamal, T.: A public key cryptosystem and a signature scheme based on discrete logarithms. In: Blakley, G.R., Chaum, D. (eds.) CRYPTO'84, Advances in Cryptology. pp. 10–18. LNCS 196, Springer, Santa Barbara, USA (1984)

[4] Fiat, A., Shamir, A.: How to prove yourself: Practical solutions to identification and signature problems. In: Odlyzko, A.M. (ed.) CRYPTO'86, 6th Annual International Cryptology Conference on Advances in Cryptology. pp. 186–194. LNCS 263, Santa Barbara, USA (1986)

[5] Knuth, D.E.: The Art of Computer Programming, vol. 2, Seminumerical Algorithms. Addison Wesley, 3rd edn. (1997)

[6] Locher, P., Haenni, R.: A lightweight implementation of a shuffle proof for electronic voting systems. In: Plödereder, E., Grunske, L., Schneider, E., Ull, D. (eds.) INFORMATIK 2014, 44. Jahrestagung der Gesellschaft für Informatik. pp. 1391–1400. No. P-232 in Lecture Notes in Informatics, Stuttgart, Germany (2014)

[7] Maurer, U.: Unifying zero-knowledge proofs of knowledge. In: Preneel, B. (ed.) AFRICACRYPT'09, 2nd International Conference on Cryptology in Africa. pp. 272–286. LNCS 5580, Gammarth, Tunisia (2009)

[8] Terelius, B., Wikström, D.: Proofs of restricted shuffles. In: Bernstein, D.J., Lange, T. (eds.) AFRICACRYPT'10, 3rd International Conference on Cryptology in Africa. pp. 100–113. LNCS 6055, Stellenbosch, South Africa (2010)

[9] Wikström, D.: A commitment-consistent proof of a shuffle. In: Boyd, C., González Nieto, J. (eds.) ACISP'09, 14th Australasian Conference on Information Security and Privacy. pp. 407–421. LNCS 5594, Brisbane, Australia (2009)

[10] Wikström, D.: User Manual for the Verificatum Mix-Net Version 1.4.0. Verificatum AB, Stockholm, Sweden (2014)

[11] Wikström, D.: How to Implement a Stand-alone Verifier for the Verificatum Mix-Net: VMN Version 3.0.2. Verificatum AB, Stockholm, Sweden (2016)