A Lightweight Implementation of a Shuffle Proof for Electronic Voting Systems

Philipp Locher[1,2] and Rolf Haenni[1]

[1] Research Institute for Security in the Information Society
Bern University of Applied Sciences, CH-2501 Biel, Switzerland
{philipp.locher, rolf.haenni}@bfh.ch
[2] Department of Informatics
University of Fribourg, CH-1700 Fribourg, Switzerland
philipp.locher@unifr.ch

Abstract: In the usual setting of a verifiable mix-net, an input batch of ciphertexts is shuffled through a series of mixers into an output batch of ciphertexts while hiding the mapping between input and output and preserving the plaintexts. Unlike shuffling, proving the correctness of a shuffle is relatively expensive and difficult to implement. In this paper, we present a new implementation of a shuffle proof based on the proof system proposed by Wikström and Terelius. The implementation offers a clean and intuitive application programming interface and can be used as a lightweight cryptographic component in applications of verifiable mix-nets. Verifiable electronic voting is the most prominent target application area.[1]

1 Introduction

Verifiable mix-nets are important building blocks in electronic voting protocols. They are used to provide vote secrecy by anonymizing the voting channel from the voter into the final tally. Some protocols use re-encryption mix-nets to shuffle the list of encrypted votes [BGP11, RBH$^+$09, RBH$^+$09], while other protocols require mix-nets to shuffle the voters' credentials [Nef01, JCJ05, HS11]. In both cases, the shuffling is performed through a series of mixers. To demonstrate the correctness of a mix-net shuffle, mixers provide individual zero-knowledge proofs—called *shuffle proofs*—to certify each step of the shuffling process. The link between the input and output of a mix-net remains hidden, as long as at least one trustworthy mixer is involved in the shuffling.

Shuffle proofs can be constructed in various ways. The first efficient shuffle proofs were proposed independently by Neff [Nef01] and Furukawa and Sako [FS01]. Neff's approach, which is based on the invariance of polynomials under the permutation of their roots, has later been improved by Groth, Ishai, and Bayer [Gro10, GI08, BG12]. The Furukawa and Sako approach is based on a commitment to a permutation matrix. Later, Wikström showed how to split the shuffle proof in an offline and online phase [Wik09]. Together with

[1] This work is supported by the Swiss National Science foundation, under the grant 200021L-140650/1.

Terelius, Wikström presented an improved and generalized proof, that allows choosing the permutation from a restricted subset of all permutations [TW10].

To the best of our knowledge, Wikström's *Verificatum* is currently the only off-the-shelf mix-net implementation that offers a complete shuffle proof [Wik13]. Verificatum has been used in the 2013 parliamentary election in Norway and for University elections in Israel. The complete Java source code is publicly available under a research license. Aside from Verificatum, multiple prototype implementations of shuffle proofs with corresponding performance tests have been mentioned in the literature [FMM+02, FMS10, BGP11, BG12], but none of them is available as a stand-alone library. The most recent performance analysis in [BG12] reports slightly better running times for Groth's approach compared to Verificatum, but this can be explained by the chosen programming languages (C++ vs. Java) and different levels of code optimization.

Contribution. We present a new implementation of a shuffle proof based on the proof system of Wikström and Terelius [Wik09, TW10]. Our implementation differs from Verificatum in multiple ways. First, we have embedded the shuffle proof in a cryptographic library with a clean and intuitive application programming interface. This greatly simplifies the integration of a shuffle proof in applications such as a mix-net. In other words, while Verificatum *is* a full-featured mix-net, we provide the necessary toolbox for building one. Second, as part of a cryptographic library, our implementation offers enhanced flexibility with respect to the homomorphic encryption system in use or the underlying algebraic group. It also supports proofs for shuffles that are not based on re-encryption, for example the one required in [HS11] for mixing voter credentials. Our shuffle proof implementation is therefore applicable in many different scenarios and is not even restricted to the context of verifiable mix-nets [Joa14]. The whole library comes as a lightweight Java component, which can be ported to any device—even to a notebook or smartphone—running a Java Virtual Machine. The source code is publicly available and free for non-commercial use.

Paper Overview. This paper gives an introduction and overview of our shuffle proof implementation. Section 2 presents a summary of the necessary technical background to understand Wikström's shuffle proof as implemented. Section 3 first gives some details about the design of the whole library and its components, and then presents a complete example of usage from a programmer's perspective. Section 4 concludes the paper with an outlook and an overview of ongoing work.

2 Shuffle Proof

We give a short introduction to Wikström's shuffle proof as presented in [Wik09, TW10]. To accentuate its essence, we fade out some technical details in our summary of the proof. In particular, we describe the proof in terms of two homomorphic one-way functions, from which respective preimage proofs are derived. We believe that the compactness of this representation is very instructive and simplifies the understanding of the proof.

2.1 Cryptographic Preliminaries

We denote by G_q a cyclic group of prime order q, for which the decisional Diffie-Hellman assumption is believed to hold. For simplicity, we write G_q always multiplicatively and assume that independent generators $g, h_1, \ldots, h_N \in G_q$ are publicly known (for $N = 1$ we write $h = h_1$). For an arbitrary group H and any pair of vectors $\bar{u} \in H^N$ and $\bar{e} \in \mathbb{Z}^N$, we use the notation $\langle \bar{u}, \bar{e} \rangle$ for both $\sum_{i=1}^{N} e_i u_i$ and $\prod_{i=1}^{N} u_i^{e_i}$, depending on whether H is written additively or multiplicatively. For an arbitrary finite set S, we write $r \in_R S$ for picking the value r uniformly at random from S.

Generalized Pedersen Commitments. We use Pedersen commitments $Com(m, r) = g^r h^m$ over G_q to commit to an integer $m \in \mathbb{Z}_q$ with randomization $r \in_R \mathbb{Z}_q$. To commit to a vector $\bar{m} = (m_1, \ldots, m_N) \in \mathbb{Z}_q^N$ of integers, we use generalized Pedersen commitments $Com(\bar{m}, r) = g^r \prod_{i=1}^{N} h_i^{m_i}$. To commit to an $N \times N$-matrix $M \in \mathbb{Z}_q^{N \times N}$, we compute generalized Pedersen commitments column-wise by

$$Com(M, \bar{r}) = \big(Com(\bar{m}_1, r_1), \ldots, Com(\bar{m}_N, r_N)\big),$$

where $\bar{m}_j$ denotes the j-th column of M and $\bar{r} = (r_1, \ldots, r_N) \in_R \mathbb{Z}_q^N$ the corresponding randomization vector. In case M is a permutation matrix relative to a permutation π of size N, then committing to M in this way allows computing a commitment to a permuted vector based on the matrix commitment (i.e., without knowing M or π). This is a consequence of the fact that generalized Pedersen commitments are additively homomorphic. More precisely, if $\bar{e}' \in \mathbb{Z}^N$ denotes the vector of integers obtained from $\bar{e} \in \mathbb{Z}^N$ by permuting its values according to π, then

$$\langle Com(M, \bar{r}), \bar{e} \rangle = Com(M\bar{e}, \langle \bar{r}, \bar{e} \rangle) = Com(\bar{e}', r)$$

is a commitment of $\bar{e}'$ with randomization $r = \langle \bar{r}, \bar{e} \rangle$.

Homomorphic Encryptions. For a randomized asymmetric encryption scheme, such as for example ElGamal or Paillier, we write $u = Enc_{pk}(m, r)$ for encrypting a plaintext $m \in \mathcal{M}$ with randomization $r \in \mathcal{R}$ and public key pk into a ciphertext $u \in \mathcal{C}$. Let $\odot$, $\oplus$, and $\otimes$ be respective group operations on $\mathcal{M}$, $\mathcal{R}$, and $\mathcal{C}$. An encryption scheme is homomorphic, if $Enc_{pk}(m_1, r_1) \otimes Enc_{pk}(m_2, r_2) = Enc_{pk}(m_1 \odot m_2, r_1 \oplus r_2)$ for all $m_1, m_2 \in \mathcal{M}$ and $r_1, r_2 \in \mathcal{R}$. A homomorphic encryption scheme allows a ciphertext $u = Enc_{pk}(m, r)$ to be re-encrypted with a new randomization r'. We write $ReEnc_{pk}(u, r') = u \otimes Enc_{pk}(1, r') = Enc_{pk}(m, r \oplus r')$, where $1 \in \mathcal{M}$ denotes the identity element of the plaintext space.

Zero-Knowledge Proofs. A zero-knowledge proof of knowledge is an interactive protocol in which a prover P convinces a verifier V that P knows a value (private input) satisfying a certain predicate (public input) without revealing any information about the value. Σ-proofs are zero-knowledge proofs of knowledge based on a three-message protocol: P passes a commitment t to V, V replies with a randomly chosen challenge c, and

P sends a response s back to V. The triple (t, c, s) is called proof transcript, which V either accepts or rejects. A large class of Σ-proofs results from any group homomorphism $\phi : G \to H$. Let $\odot$ and $\otimes$ be respective operators of G and H. If $x \in G$ is the private input known to P and $y = \phi(x) \in H$ the public input known to P and V, we write

$$\Sigma\text{-proof}\Big[\ x \mid\ y = \phi(x)\Big]$$

for the Σ-proof of knowing the preimage x. It can be constructed by the following standard procedure: P picks $r \in_R G$ and sends $t = \phi(r)$ to V, V replies with $c \in_R C \subseteq \mathbb{Z}$, and P sends the response $s = r \odot x^c$ back to V. V accepts the proof, if and only if $\phi(s) = t \otimes y^c$. This general construction of preimage Σ-proofs covers many known proofs of knowledge as special cases [Mau09]. It can be turned into a non-interactive proof by obtaining c from a random oracle using (t, y) as query [FS86].[2]

2.2 Proof of a Shuffle

The shuffle proof according to Wikström and Terelius consists of an offline and an online phase [Wik09, TW10]. In the offline phase, the mixer commits to a permutation matrix and proves under zero knowledge that the commitment contains indeed a permutation matrix. An upper bound for the size of the shuffle is required to conduct this phase prior to the actual shuffling. Later, in the online phase, the mixer performs the shuffle according to the committed permutation matrix and proves the correctness of the shuffle under zero knowledge.

Offline Phase. Let π be a permutation of size N and $\bar{c}_\pi = Com(M, \bar{s}) \in G_q^N$ a commitment to the corresponding permutation matrix M. If $\bar{x} = (x_1, \ldots, x_N)$ is a vector of N independent variables, then an $N \times N$-matrix M over $\mathbb{Z}_q$ is a permutation matrix if and only if $\prod_{i=1}^N \langle \bar{m}_i, \bar{x} \rangle = \prod_{i=1}^N x_i$ and $M\bar{1} = \bar{1}$. These properties allow to prove that $\bar{c}_\pi$ is a commitment to a permutation matrix [TW10]:

$$\Sigma\text{-proof}\left[\begin{array}{c} v, w \in \mathbb{Z}_q \\ \bar{e}' \in \mathbb{Z}_q^N \end{array} \middle| Com(\bar{1}, v) = \langle \bar{c}_\pi, \bar{1} \rangle \wedge Com(\bar{e}', w) = \langle \bar{c}_\pi, \bar{e} \rangle \wedge \prod_{i=1}^N e'_i = \prod_{i=1}^N e_i \right],$$

where $v = \langle \bar{s}, \bar{1} \rangle$, $w = \langle \bar{s}, \bar{e} \rangle$, and $\bar{e}' = (e'_1, \ldots, e'_N) = (e_{\pi(1)}, \ldots, e_{\pi(N)})$ are the private inputs. The vector $\bar{e} = (e_1, \ldots, e_N) \in \mathbb{Z}_q^N$ is a public input selected and communicated beforehand by the verifier.[3] The last part of the proof, which consists in showing the equality $\prod_{i=1}^N e'_i = \prod_{i=1}^N e_i$, can be achieved using a recursive commitment structure $c_1, \ldots, c_N$ with base case $c_0 = h$ [Wik12]. This leads to a slightly different representation

[2] To establish a binding between prover and proof, the prover's identity is sometimes adjoined to the random oracle query.

[3] Usually, the values e_i are selected from a subset $[0, 2^{k_e} - 1]^N \subseteq \mathbb{Z}_q$, where k_e is a security parameter.

of the above proof:

$$\Sigma\text{-proof}\left[\begin{array}{c|l} v,w,d\in\mathbb{Z}_q & Com(\bar{1},v)=\langle\bar{c}_\pi,\bar{1}\rangle\ \wedge\ Com(\bar{e}',w)=\langle\bar{c}_\pi,\bar{e}\rangle\ \wedge \\ \bar{t},\bar{e}'\in\mathbb{Z}_q^N & \bigwedge_{i=1}^N(c_i=g^{t_i}c_{i-1}^{e'_i})\ \wedge\ Com(0,d)=c_N/h^{\prod_{i=1}^N e_i} \end{array}\right],$$

where $\bar{t}=(t_1,\ldots,t_N)\in_R\mathbb{Z}_q^N$ and $d=d_N$ are additional private inputs for $d_0=0$ and $d_i=t_i+e'_i\,d_{i-1}$ for $i>0$. This leads directly a homomorphic one-way function,

$$\phi_{offline}(v,w,\bar{t},d,\bar{e}')=\Big(Com(\bar{1},v),Com(\bar{e}',w),g^{t_1}c_0^{e'_1},\ldots,g^{t_N}c_{N-1}^{e'_N},Com(0,d)\Big),$$

which can be used for constructing a preimage proof by the standard procedure.

Online Phase. Let $\mathcal{C}$ denote the ciphertext space of the given homomorphic encryption scheme. We assume that the largest cyclic subgroup of $\mathcal{C}$ is of the same order q as the cyclic group used for the Pedersen commitments.[4] The input list of ciphertexts is denoted by $\bar{u}=(u_1,\ldots,u_N)\in\mathcal{C}^N$ and the corresponding shuffled list of permuted and re-encrypted ciphertexts by $\bar{u}'=(u'_1,\ldots,u'_N)\in\mathcal{C}^N$. Again, a public vector $\bar{e}\in\mathbb{Z}_q^N$ is selected and communicated beforehand by the verifier. A proof that $\bar{u}'$ has been formed correctly by shuffling $\bar{u}$ according to the committed permutation matrix $\bar{c}_\pi=Com(M,\bar{s})$ can be constructed as follows [Wik09]:

$$\Sigma\text{-proof}\left[\begin{array}{c|l} r,w\in\mathbb{Z}_q & Com(\bar{e}',w)=\langle\bar{c}_\pi,\bar{e}\rangle\ \wedge\ \prod_{i=1}^N(u'_i)^{e'_i}=\prod_{i=1}^N(u_i)^{e_i}Enc(1,r) \\ \bar{e}'\in\mathbb{Z}_q^N & \end{array}\right],$$

where $r=\langle\bar{r},\bar{e}\rangle$, $w=\langle\bar{s},\bar{e}\rangle$, and $\bar{e}'=(e'_1,\ldots,e'_N)=(e_{\pi(1)},\ldots,e_{\pi(N)})$ are the private inputs for $\bar{r}=(r_1,\ldots,r_N)\in_R\mathbb{Z}_q^N$ and $u'_i=ReEnc(u_{\pi(i)},r_{\pi(i)})$. Again, this implies a homomorphic one-way function,

$$\phi_{online}(r,w,\bar{e}')=\Big(Com(\bar{e}',w),\textstyle\prod_{i=1}^N(u'_i)^{e'_i}Enc(1,-r)\Big),$$

for which a preimage proof can be constructed by the standard procedure.

3 Design and Implementation

Some selected details of our shuffle proof implementation are the focus of this section. The goal is to make the reader familiar with the design and some basic concepts of our implementation. For this, we give first some background information about *UniCrypt*, the cryptographic library into which our shuffle proof is embedded. Then we discuss a complete example of usage to illustrate the provided interface from a programmer's perspective. The chosen example covers the full process of generating and mixing some ElGamal encryptions and constructing and verifying the offline and online proofs.

[4] For simplicity, we expect the same group order for the encryptions and commitments, but this is not a necessary requirement for the proof.

3.1 UniCrypt

UniCrypt is a Java library developed for the purpose of simplifying the implementation of cryptographic voting protocols.[5] It consists of two layers, one for the mathematical fundament and one for the cryptographic primitives. The mathematical layer deals with all sorts of algebraic structures, corresponding elements, and functions. Its purpose is to provide strict type safety on a mathematical level: elements always "know" the algebraic structure to which they belong, i.e., applying a group operator is only allowed for elements of the group and evaluating a function is is only allowed for elements of the domain.

The cryptographic layer provides interfaces and implementations of various cryptographic schemes: symmetric and asymmetric encryption, secret sharing, commitments, digital signatures, zero-knowledge proofs, mix-nets, and more. It also contains a package for generating pseudo-random numbers, common reference strings, or random oracles. In the remaining paragraphs, we provide additional information to some of the cryptographic components, which appear in the example of the following subsection. Corresponding top-level Java interfaces exist in the UniCrypt library.

Mixer. This interface specifies the functionality of a pure cryptographic shuffle: an input list of values is shuffled into an output list of values without proving its correctness. Currently, two implementations of this interface are available: a *re-encryption mixer*, which permutes and re-encrypts a list of ciphertexts of a homomorphic encryption scheme, and a so-called *identity mixer*, which shuffles credentials according to the method described in [HS11].

Proof System. This is the general interface for many different types of zero-knowledge proofs. It specifies two principal methods, one for generating a proof for given private and public inputs, and one for verifying such proofs. Multiple standard zero-knowledge proofs of knowledge are implemented in a generic way. There are implementations for basic preimage proofs, for conjunctive or disjunctive compositions of preimage proofs, and for equality and inequality proofs. They can all be applied to any homomorphic function. There are also implementations for validity proofs, which can deal with ElGamal encryptions and Pedersen commitments. The online part of the shuffle proof is implemented in two different ways, one for a re-encryption mixer and one for an identity mixer.

Challenge Generator. The proof system implementation in UniCrypt is very flexible about constructing proofs in an interactive or non-interactive manner. The process of creating the challenge is abstracted in our concept of a *challenge generator*. During the construction or verification of a proof, the proof system simply calls the challenge generator to get a suitable challenge. The details of selecting the challenge in a concrete implementation are hidden behind the interface. The default implementations are non-interactive, which obtain the challenge from calling a random oracle.

[5] The source code of the UniCrypt library is publicly available on GitHub under a dual AGPLv3/commercial licence, see https://github.com/bfh-evg/unicrypt.

3.2 Example of Usage

To present our shuffle proof implementation from a programmer's point of view, we present a complete example of a re-encryption shuffle including proving and verifying the correctness of the shuffle. To keep the code as tight as possible, the example operates often in a default manner. For example, the independent generators used in the generalized Pedersen commitment are implicitly derived from the default common reference string, and non-interactive challenge generators are created automatically by the proof systems.

Setup. We first create a list of ElGamal ciphertexts, which in a normal application is given by the context. For this, we randomly select a cyclic group $G_q \subseteq \mathbb{Z}_p^*$ such that $p = 2q + 1$ is a safe prime of a specified bit length. Then the ElGamal encryption scheme is instantiated based on the default generator of the selected cyclic group, and a public key pk is chosen at random (we don't decrypt in this example, so no private key is needed). Finally, the list $\bar{u}$ of input ciphertexts is created based on random messages.

```
// Create cyclic group for random safe prime (1024 bits)
CyclicGroup group = GStarModSafePrime.getRandomInstance(1024);

// Create ElGamal encryption scheme and select random public key pk
ElGamalEncryptionScheme elGamal =
        ElGamalEncryptionScheme.getInstance(group.getDefaultGenerator());
Element pk = group.getRandomElement();

// Set shuffle size and create random ElGamal ciphertexts
int n = 100;
Tuple ciphertexts = Tuple.getInstance();
for (int i = 0; i < n; i++) {
    Pair c = elGamal.encrypt(pk, group.getRandomElement());
    ciphertexts = ciphertexts.add(c);
}
```

Listing 1: Setup

Shuffle. To shuffle the list of input ciphertexts $\bar{u}$, a re-encryption mixer is instantiated based on the ElGamal encryption scheme, the public key pk, and the shuffle size N. Then permutation π and the re-encryption randomizations $\bar{r}$ are selected at random. To be able to proof the correctness of the shuffle later, it is important to create these values explicitly. Finally, calling the shuffle method of the mixer outputs a list of ciphertexts $\bar{u}'$.

```
// Create mixer, a random permutation pi, and randomizations r
ReEncryptionMixer mixer = ReEncryptionMixer.getInstance(elGamal, pk, n);
PermutationElement pi = mixer.getPermutationGroup().getRandomElement();
Tuple r = mixer.generateRandomizations();

// Shuffle ciphertexts using pi and r
Tuple shuffledCiphertexts = mixer.shuffle(ciphertexts, pi, r);
```

Listing 2: Shuffle

Offline Phase. The first step in the offline phase of the shuffle proof is to generate the permutation matrix commitment $\bar{c}_\pi$ relative to π. For this, we instantiate a permutation commitment scheme and select the commitment randomizations $\bar{s}$ explicitly at random. To prove that $\bar{c}_\pi$ is a commitment to a permutation, we instantiate a permutation commitment proof system, which allows creating the proof using $(\pi, \bar{s})$ as private and $\bar{c}_\pi$ as public input. Note that the code of Listing 3 could be executed before the ciphertexts are shuffled in the last line of Listing 2.

```
// Create permutation commitment c_pi based on pi and randomizations s
PermutationCommitmentScheme pcs =
        PermutationCommitmentScheme.getInstance(group, n);
Tuple s = pcs.getRandomizationSpace().getRandomElement();
Tuple c_pi = pcs.commit(pi, s);

// Create permutation commitment proof system
PermutationCommitmentProofSystem pcps =
        PermutationCommitmentProofSystem.getInstance(group, n);

// Define private and public inputs
Pair offlinePrivateInput = Pair.getInstance(pi, s);
Element offlinePublicInput = c_pi;

// Generate permutation commitment proof
Pair offlineProof =
        pcps.generate(offlinePrivateInput, offlinePublicInput);
```

Listing 3: Online Phase (Proof of Knowledge of Permutation Matrix)

Online Phase. Finally, the shuffle proof can be generated with the help of a re-encryption shuffle proof system. The triples $(\pi, \bar{s}, \bar{r})$ and $(\bar{c}_\pi, \bar{u}, \bar{u}')$ are the private and public inputs, respectively.

```
// Create shuffle proof system
ReEncryptionShuffleProofSystem rsps =
        ReEncryptionShuffleProofSystem.getInstance(group, n, elGamal, pk);

// Define private and public inputs
Triple onlinePrivateInput = Triple.getInstance(pi, s, r);
Triple onlinePublicInput =
        Triple.getInstance(c_pi, ciphertexts, shuffledCiphertexts);

// Generate shuffle proof
Triple onlineProof = rsps.generate(onlinePrivateInput, onlinePublicInput);
```

Listing 4: Online Phase (Commitment Consistent Proof of a Shuffle)

Verification. The verification of the overall proof is straightforward: just call the verification methods of the proof systems with the corresponding proof and the public input as arguments, and make sure that the same permutation commitment is included in both public inputs. The proof systems are either given by the context or can be created based on common values.

```
// Verify permutation commitment proof
boolean v1 = pcps.verify(offlineProof, offlinePublicInput);

// Verify shuffle proof
boolean v2 = rsps.verify(onlineProof, onlinePublicInput);

// Verify equality of permutation commitments
boolean v3 =
        offlinePublicInput.isEquivalent(onlinePublicInput.getFirst());

if (v1 && v2 && v3) success();
```

Listing 5: Proof Verification

4 Conclusion

In this paper, we presented a short summary of the shuffle proof of Wikström and Terelius and presented an overview of a new implementation embedded in a lightweight Java library. With our example of shuffling a list of ElGamal ciphertexts, we illustrated the construction of a shuffle proof from a programming perspective. It turns out that the using the library is straightforward and intuitive. As a standalone library, it can be easily integrated in any mix-net based electronic voting system.

The results of preliminary performance tests are comparable to the results reported in the literature for other shuffle proof implementations (100,000 ElGamal ciphertexts within a few minutes, for a residue class of 160/1024 bits and on a standard notebook). We expect performance improvements by further optimizing the implementation (multi-exponentiation, pre-computations, caching, etc.). The flexibility of our library with respect to working with different groups—for example by using elliptic curves—can further speed up the proof generation and verification without code modifications.

References

[BG12] S. Bayer and J. Groth. Efficient Zero-Knowledge Argument for Correctness of a Shuffle. In D. Pointcheval and T. Johansson, editors, *EUROCRYPT'12, 31st Annual International Conference on Theory and Applications of Cryptographic Techniques*, LNCS 7237, pages 263–280, Cambridge, UK, 2012.

[BGP11] P. Bulens, D. Giry, and O. Pereira. Running Mixnet-Based Elections with Helios. In H. Shacham and V. Teague, editors, *EVT/WOTE'11, Electronic Voting Technology Workshop/Workshop on Trustworthy Elections*, San Francisco, USA, 2011.

[FMM+02] J. Furukawa, H. Miyauchi, K. Mori, S. Obana, and K. Sako. An Implementation of a Universally Verifiable Electronic Voting Scheme based on Shuffling. In M. Blaze, editor, *FC'02, 6th International Conference on Financial Cryptography*, LNCS 2357, pages 16–30, Southampton, Bermuda, 2002.

[FMS10] J. Furukawa, K. Mori, and K. Sako. An Implementation of a Mix-Net Based Network Voting Scheme and Its Use in a Private Organization. In D. Chaum, M. Jakobsson, R. Rivest, P. Y. A. Ryan, J. Benaloh, M. Kutylowski, and B. Adida, editors, *Towards Trustworthy Elections*, LNCS 6000, pages 141–154. Springer, 2010.

[FS86] A. Fiat and A. Shamir. How to Prove Yourself: Practical Solutions to Identification and Signature Problems. In A. M. Odlyzko, editor, *CRYPTO'86, 6th Annual International Cryptology Conference on Advances in Cryptology*, pages 186–194, 1986.

[FS01] J. Furukawa and K. Sako. An Efficient Scheme for Proving a Shuffle. In J. Kilian, editor, *CRYPTO'01, 21st Annual International Cryptology Conference on Advances in Cryptology*, LNCS 2139, pages 368–387, Santa Barbara, USA, 2001.

[GI08] J. Groth and Y. Ishai. Sub-Linear Zero-Knowledge Argument for Correctness of a Shuffle. In N. Smart, editor, *EUROCRYPT'08, 27th International Conference on the Theory and Applications of Cryptographic Techniques*, LNCS 4965, pages 379–396, Istanbul, Turkey, 2008.

[Gro10] J. Groth. A Verifiable Secret Shuffle of Homomorphic Encryptions. *Journal of Cryptology*, 23(4):546–579, 2010.

[HS11] R. Haenni and O. Spycher. Secure Internet Voting on Limited Devices with Anonymized DSA Public Keys. In H. Shacham and V. Teague, editors, *EVT/WOTE'11, Electronic Voting Technology Workshop/Workshop on Trustworthy Elections*, 2011.

[JCJ05] A. Juels, D. Catalano, and M. Jakobsson. Coercion-Resistant Electronic Elections. In V. Atluri, S. De Capitani di Vimercati, and R. Dingledine, editors, *WPES'05, 4th ACM Workshop on Privacy in the Electronic Society*, pages 61–70, Alexandria, USA, 2005.

[Joa14] R. Joaquim. How to Prove the Validity of a Complex Ballot Encryption to the Voter and the Public. *Journal of Information Security and Applications*, accepted, 2014.

[Mau09] U. Maurer. Unifying Zero-Knowledge Proofs of Knowledge. In B. Preneel, editor, *AFRICACRYPT'09, 2nd International Conference on Cryptology in Africa*, volume 5580 of *LNCS 5580*, pages 272–286, Gammarth, Tunisia, 2009.

[Nef01] C. A. Neff. A Verifiable Secret Shuffle and its Application to E-Voting. In P. Samarati, editor, *CCS'01, 8th ACM Conference on Computer and Communications Security*, pages 116–125, Philadelphia, USA, 2001.

[RBH+09] P. Y. A. Ryan, D. Bismark, J. Heather, S. Schneider, and X. Zhe. Prêt à Voter: a Voter-Verifiable Voting System. *IEEE Transactions on Information Forensics and Security*, 4(4):662–673, 2009.

[TW10] B. Terelius and D. Wikström. Proofs of Restricted Shuffles. In D. J. Bernstein and T. Lange, editors, *AFRICACRYPT'10, 3rd International Conference on Cryptology in Africa*, LNCS 6055, pages 100–113, Stellenbosch, South Africa, 2010.

[Wik09] D. Wikström. A Commitment-Consistent Proof of a Shuffle. In C. Boyd and J. González Nieto, editors, *ACISP'09, 14th Australasian Conference on Information Security and Privacy*, LNCS 5594, pages 407–421, Brisbane, Australia, 2009.

[Wik12] D. Wikström. *How to Implement a Stand-alone Verifier for the Verificatum Mix-Net*. Verificatum AB, Stockholm, Sweden, 2012.

[Wik13] D. Wikström. *User Manual for the Verificatum Mix-Net Version 1.2.0*. Verificatum AB, Stockholm, Sweden, 2013.