Blackbox Constructions from Mix-Nets

Douglas Wikström

KTH Royal Institute of Technology
dog@kth.se

Abstract. Mix-nets constructed from homomorphic cryptosystems can be generalized to process lists of ciphertexts as units and use different public keys for different parts of such lists. We present a number of black-box constructions that enriches the set of operations provided by such mix-nets. The constructions are simple, fully practical, and eliminates the need for some specialized protocols.

1 Introduction

A mix-net is a cryptographic protocol executed by a set of mix-servers. They first run a distributed key generation protocol, i.e., they generate a joint public key for which the secret key is secret shared among the servers. Then they take a list of ciphertexts as input and output the corresponding plaintexts in random order without disclosing any knowledge about how they are related to the input ciphertexts. Additional functionality is possible such as only decrypting or outputting re-encrypted and permuted ciphertexts instead plaintexts. The notion of mix-nets was introduced by Chaum [5].

Although there are a few examples of non-standard applications of mix-nets, e.g., Jakobsson et al. [22], the main application is in electronic voting systems or in applications which can be framed as such.

Already in early work on efficient universally verifiable mix-nets such as Neff [23] and Furukawa and Sako [13] it was noted that each input could be a list of ciphertexts that is processed as a unit. This allows processing longer plaintexts and multiple choices in an election at once. A related generalization is that ciphertexts encrypted under different public keys can be processed as a unit. This is made explicit by Groth [16], albeit in abstract form.

The problem of constructing a provably secure mix-net was arguably solved years ago through a long line of theoretical works leading up to a few variations with components from, e.g., [9, 26, 14, 25, 23, 13, 16, 30].

There is a rich literature on mix-nets, in particular numerous failed attempts to construct a verifiable mix-net that led up to the current state of the art, as well as different forms of proofs of shuffles, but this literature is not relevant in this paper. There are also mix-nets that are not provably secure, but they should not be used in security critical applications such as elections, if for no other reason since they are not verifiable. Thus, they are not considered here.

VoteHere and NEC [12] presented the first implementations. The softwares were never publicly available and we do not know exactly what VoteHere implemented. NEC's implementation used a central party for verification and did not implement processing of long ciphertexts [11].

As far as we know the Verificatum Mix-Net (VMN) [1] was the first complete and fully distributed implementation that is faithful to cryptographic theory (completed 2008 and released under a free and open source license 2010). Both generalizations of the El Gamal cryptosystem are implemented using abstractions similar to those used in [16]. Versions of VMN have been used in Wombat [3] (2011) for student elections and election of the party leader of Meretz, by nVotes (formerly Agora Voting) [31, 24] in primary and local elections from 2011 onwards, and in municipal elections in Norway 2013 and Estonia 2017 and in national elections in Estonia 2019.

The Helios electronic voting system in it's mix-net incarnation [20, 4] and the UniCrypt/UniVote [10] implementations of voting systems both contain mix-nets that were implemented after the completion of VMN. The worthwhile goal of the former is to have a simplistic implementation in a script language with a different information flow. The latter is an attempt to re-implement a subset of the functionality in the same programming language (and other functionality), but using different programming patterns [17].

Both use the proof of a shuffle of Terelius and Wikström [30]. Neither project is comparable to VMN in scope, security, or completeness, but both can be used with the blackbox constructions to some extent, so the constructions in this paper are generic.

We remark that some results mentioned above have been re-discovered with minor changes, e.g., the notion of "parallel mixing" [27] is essentially shuffling multiple ciphertexts encrypted under the same or different public keys, and "exponentiation mixing" [18], for which a special purpose protocol follows trivially from [32] and straightforward a blackbox solution is described here.

1.1 Contributions

We describe a number of ways in which a mix-net can be used as a blackbox. Special purpose protocols are frequently proposed in the electronic voting literature that replicate such functionality in various more or less sound ways, so there is a need to give an explicit unified treatment. The constructions presented here are at most a small constant factor slower than a corresponding special purpose protocol.

We explain how to swap public key encryption for one-time pad encryption for a designated party. Then we sketch how some non-threshold access structures can be implemented. We describe how to jointly compute any group homomorphism on encrypted values with public and random scalars. This observation gives weak universal mixing.

Then we show how a hypergraph with encrypted labels can be shuffled, i.e., the labels are re-encrypted and the hypergraph randomly permuted. This con-

struction seems novel, but simple cases of it are well known and of immediate practical value.

Finally, we consider how to swap encryption under one public key for encryption under another public key. This is necessarily not blackbox, since it depends on the structure of the cryptosystem itself, but it suffices to make blackbox calls to the mix-net.

We think it is worthwhile to present these constructions in explicit form, since we have seen several papers where the main results can either be viewed as instances of these constructions or can be explained in a simpler and much more focused way using these constructions, e.g., distributed Prêt à Votér ballot generation [27] or checking numbers in Selene [28].

1.2 Why Are Blackbox Constructions Important?

In theory, blackbox constructions are important to reduce complexity and the need for assumptions. One of the main foci of theoretical cryptography is to relate different notions and assumptions. A well-known example is that a pseudo-random generator can be constructed from any one-way function [19]. Blackbox constructions are typically also easier to use as tools when developing novel ideas and presenting them.

However, for the case of mix-nets the practical implications are much more important. Implementing security critical software correctly is time consuming and difficult. The software should have as few dependencies to external libraries as possible, and all code must be analyzed, tested, optimized, and documented both in code and in manuals. All data formats must be documented as well.

The use of blackbox constructions instead of special purpose protocols drastically simplifies protocol construction and reduces the expertise needed to use the functionality in a secure way. This reduces both security risks and costs. In particular, it simplifies the specification of universally verifiable proofs of correctness.

1.3 Motivating Example

Even in elections where only a single ciphertext is needed to encrypt a vote it is useful to be able to process multiple ciphertexts as units. Consider an election with thousands of voting precincts of a few hundred up to a few thousand voters each. There is no need to keep information about the voting district secret at the time the vote is cast, whether in a polling station or on the Internet, and votes should only be shuffled within the precinct before revealing the set of votes.

We can solve this problem either by processing the ciphertexts from each voting precinct separately, or we could use a so-called restricted proof of a shuffle [30] to ensure that the mix-servers never mixes ciphertexts from different voting precincts.

An alternative solution is to simply add the index of the precinct as a prefix to each vote, encrypt the index, and process both ciphertexts as a unit. The

prefix ciphertext could be added by the voter when casting the vote, but it can just as well be added by the mix-servers at the time of tallying since the precinct index is public. We stress that fixed and public randomness can be used to form the encryptions of the indices.

The prefix ciphertexts of the shuffled pairs of ciphertexts are then decrypted before or at the same time as the encryptions of the votes. Sorting with respect to indices separates the ciphertexts (or votes) of each precinct.

2 Background

Although most of the constructions work for any homomorphic cryptosystem, we concentrate on the El Gamal cryptosystem for concreteness. We also introduce notation for hypergraphs to consider processing of more complex objects than mere ciphertexts.

2.1 El Gamal Cryptosystem

The El Gamal cryptosystem [8] is defined over a group G_q of prime order q with standard generator g. The set $\mathcal{M}$ of plaintexts is defined to be the group G_q and the set of ciphertexts $\mathcal{C}$ is the product space $\mathcal{M} \times \mathcal{M}$. The randomness used to encrypt is sampled from $\mathcal{R} = \mathbb{Z}_q$.

A secret key $x \in \mathcal{R}$ is sampled randomly, and a corresponding public key $pk = (g, y)$ is defined by $y = g^x$. To encrypt a plaintext $m \in \mathcal{M}$, a random exponent $s \in \mathcal{R}$ is chosen and the ciphertext in $\mathcal{C}$ is computed as $\mathsf{Enc}_{pk}(m, s) = (g^s, y^s m)$. A plaintext can then be recovered from such a ciphertext (u, v) as $\mathsf{Dec}_x(u, v) = u^{-x} v = m$.

To encrypt an arbitrary string of bounded length t we also need an injection $\{0,1\}^t \to \mathcal{M}$, which can be efficiently computed and inverted.

Homomorphic. The cryptosystem is homomorphic, i.e., for every public key pk and every ciphertexts

$$(u_1, v_1) = \mathsf{Enc}_{pk}(m_1, s_1) \quad \text{and} \quad (u_2, v_2) = \mathsf{Enc}_{pk}(m_2, s_2)$$

their element-wise product

$$(u_1 u_2, v_1 v_2) = \mathsf{Enc}_{pk}(m_1 m_2, s_1 + s_2)$$

is an encryption of $m_1 m_2$. If we set $m_2 = 1$, then this feature can be used to *re-encrypt* (u_1, v_1) without knowledge of the secret key. To see this, note that for every fixed s_1 and random s_2, the sum $s_1 + s_2$ is randomly distributed in $\mathcal{R}$.

Distributed El Gamal Cryptosystem. The El Gamal cryptosystem also allows efficient protocols for distributed key generation and distributed decryption of ciphertexts by k parties with threshold λ, i.e., λ parties are needed to decrypt.

The distributed key generation protocol generates a Shamir secret [29] sharing polynomial $p(z) = \sum_{s=0}^{\lambda-1} \gamma_s z^s$ of degree $(\lambda - 1)$ defined over $\mathcal{R}$. The joint output of the protocol is then a *polynomial in the exponent* $(\Gamma_0, \ldots, \Gamma_{\lambda-1})$ defined by $\Gamma_s = g^{\gamma_s}$ and the secret output x_l to the lth party is defined as the secret share $x_l = p(l)$. The joint public key is then $y = \Gamma_0 = g^x$, where $x = p(0)$, and the public key y_l of the lth party can be derived as $y_l = \prod_{s=0}^{\lambda-1} \Gamma_s^{l^s} = g^{x_l}$. The details [9, 14] of the verifiable secret sharing scheme are not important in this paper, but the point of the sharing polynomial is that each server can verify that its share is correct using this relation.

To jointly decrypt a ciphertext (u, v), the lth party publishes a *partial decryption factor* f_l computed as $\mathsf{PDec}_{x_l}(u, v) = u^{-x_l}$ and proves using a zero-knowledge proof that it computed the decryption factor correctly relative to its public key y_l. Let Δ be a set of size λ of indices l such that the proofs are correct. Then Lagrange coefficients

$$c_l = \prod_{i \in \Delta \setminus \{l\}} \frac{i}{i - l}$$

such that $\sum_{l \in \Delta} c_l x_l = x$ can be computed. Instead of recovering x in the open we can perform similar operations "in the exponent". More precisely, decryption factors can be combined to a joint decryption factor

$$f = \prod_{l \in \Delta} f_l^{c_l} = \mathsf{PDec}_x(u, v) \ .$$

The ciphertext can then be trivially decrypted as $\mathsf{TDec}((u, v), f) = vf = m$. If no set Δ of size λ exists, then no knowledge about the plaintext is leaked and the ciphertexts cannot be decrypted.

2.2 Encrypting Longer Messages with Multiple Keys

The El Gamal cryptosystem can be generalized in several ways to encrypt longer messages. One way is to simply use multiple public keys. More precisely, suppose that $pk = (pk_1, \ldots, pk_\kappa)$ is a list of public keys with corresponding secret keys $sk = (sk_1, \ldots, sk_\kappa)$, where $pk_i \in \mathcal{C}$ and $sk_i \in \mathcal{R}$. Then a message $m = (m_1, \ldots, m_\kappa) \in \mathcal{M}^\kappa$ can be encrypted as

$$\mathsf{Enc}_{pk}(m, s) = \big(\mathsf{Enc}_{pk_1}(m_1, s_1), \ldots, \mathsf{Enc}_{pk_\kappa}(m_\kappa, s_\kappa)\big) \ ,$$

where $s \in \mathcal{R}^\kappa$. We view this as the natural generalization of El Gamal to product groups. We say that κ is the *key width* and define the message space to be $\mathcal{M}_\kappa = \mathcal{M}^\kappa$, the randomness space to be $\mathcal{R}_\kappa = \mathcal{R}^\kappa$, and the ciphertext space to be $\mathcal{C}_\kappa = \mathcal{M}_\kappa \times \mathcal{M}_\kappa$.

Let $g_\mathcal{M} \in \mathcal{M}$ be a generator. Then the element $g = (g_\mathcal{M}, g_\mathcal{M}, \ldots, g_\mathcal{M})$ generates $\mathcal{M}_\kappa$ in the sense that for each element $u \in \mathcal{M}_\kappa$ there is a unique element $s \in \mathcal{R}_\kappa$ such that $u = (g_\mathcal{M}^{s_1}, \ldots, g_\mathcal{M}^{s_\kappa})$. We use the following notation

$$g^s = g^{(s_1, \ldots, s_\kappa)} = (g_\mathcal{M}^{s_1}, \ldots, g_\mathcal{M}^{s_\kappa}) \ ,$$

i.e., exponentiation is interpreted component wise.

With this notation, a secret key sk for El Gamal with key width κ is a randomly chosen element $x \in \mathcal{R}_\kappa$ and the corresponding public key pk is defined as (g, y), where $y = g^x$. To encrypt a message $m \in \mathcal{M}_\kappa$ an element $s \in \mathcal{R}_\kappa$ is sampled randomly and then the ciphertext is computed as $\mathsf{Enc}_{pk}(m, s) = (g^s, y^s m)$, where $y^s m$ is interpreted as component-wise multiplication. Decryption and computation of decryption factors are defined similarly.

2.3 Encrypting Longer Messages with Multiple Ciphertexts

A standard hybrid argument implies that a longer plaintexts of the form $m = (m_1, \ldots, m_\omega) \in \mathcal{M}_\kappa^\omega$ can be encrypted using a public key $pk \in \mathcal{C}_\kappa$ by encrypting each component independently, as $\big(\mathsf{Enc}_{pk}(m_1, s_1), \ldots, \mathsf{Enc}_{pk}(m_\omega, s_\omega)\big)$, where $s = (s_1, \ldots, s_\omega) \in \mathcal{R}_\kappa^\omega$ is chosen randomly.

It is convenient to generalize our notation similarly to the generalization used for multiple keys above. Thus, we let $\mathcal{M}_{\kappa,\omega} = \mathcal{M}_\kappa^\omega$ be the plaintext space, we let $\mathcal{R}_{\kappa,\omega} = \mathcal{R}_\kappa^\omega$ be the randomness space, and let $\mathcal{C}_{\kappa,\omega} = \mathcal{M}_{\kappa,\omega} \times \mathcal{M}_{\kappa,\omega}$ be the ciphertext space. With this notation, encryption of a message $m \in \mathcal{M}_{\kappa,\omega}$ using randomness $s \in \mathcal{R}_{\kappa,\omega}$ is simply denoted $\mathsf{Enc}_{pk}(m, s) = (g^s, y^s m)$, where g is understood to be a generator of $\mathcal{M}_\kappa$.

Note that with this generalization $pk \in \mathcal{C}_\kappa$ and not in $\mathcal{C}_{\kappa,\omega}$ which might have been expected by the reader, but this is natural since the ω is variable even for a fixed public key. Decryption and computation of decryption factors can be defined in the natural way.

Abuse of Notation. We abuse notation when convenient and view a ciphertext in $\mathcal{C}_{\kappa,\omega}$ to be a list of ciphertexts $\big(w_1, \ldots, w_\omega\big)$, where $w_i \in \mathcal{C}_\kappa$, despite that it is of the form $\big((u_1, \ldots, u_\omega), (v_1, \ldots, v_\omega)\big)$, i.e., when we abuse notation we think of w_i as (u_i, v_i).

2.4 Blackbox Mix-net

The input to a mix-net is a list of ciphertexts. The mix-servers then take turns and re-encrypt each ciphertext and permute their order. The output of the λth mix-server is a list of ciphertexts that have been re-encrypted and permuted by λ mix-servers. Thus, if less than λ mix-servers are corrupted, this guarantees that no knowledge is leaked about the correspondence between the input and output ciphertexts. This operation is often called a *shuffle*.

To ensure that no ciphertexts are replaced each mix-server also proves in zero-knowledge that it did indeed only re-encrypt and permute its input to form its output. Such proofs are called *proofs of shuffles*. The first efficient solutions were given by Neff [23] and Furukawa and Sako [13], but several proposals have been presented and in VMN the proof of a shuffle of Terelius and Wikström [30] is used. We think this allows a more intuitive explanation which is important in the context of electronic voting, but this is arguably a matter of taste. These

proofs of shuffles work for both generalizations of El Gamal. A blackbox mix-net is capable of:

1. *joint key generation* (joint public key and secret shared secret key),
2. *shuffling* ciphertexts (re-encrypting and permuting),
3. *decrypting* ciphertexts, and
4. *mixing* ciphertexts (shuffling and decrypting).

For the three latter operations the plaintexts of the input ciphertexts must be straight-line extractable by a simulator in a proof of security, and sometimes the ciphertexts must also be globally unique. From now on we assume that the application ensures this when needed.

2.5 Hypergraphs

We denote a graph G by a pair (V, E), where V is a set of vertices and E is a set of edges. In the case of undirected graphs an edge e is a set of two nodes $u_1, u_2 \in V$, whereas in the case of a directed graph the edge is an ordered pair to encode that the edge goes from u_1 to u_2. The same convention is used for hypergraphs, i.e., the edge of an undirected hypergraph is a set of nodes $u_1, u_2, \ldots, u_s \in V$ and the corresponding edge in a directed hypergraph is the ordered list $(u_1, u_2, \ldots, u_s)$.

3 Swapping to One-Time Pad Encryption

Sometimes we want the plaintexts of a mix-net to be available only to a dedicated third party. There is a simple light-weight approach based on one-time pad encryption to achieve this.

For an encryption $\mathsf{Enc}_{pk}(m)$ we simply let the receiver generate an encryption $\mathsf{Enc}_{pk}(\alpha)$ of a randomly chosen value in $\alpha \in \mathcal{M}$. The mix-net then decrypts the ciphertext $\mathsf{Enc}_{pk}(\alpha)\mathsf{Enc}_{pk}(m) = \mathsf{Enc}_{pk}(\alpha m)$ to get αm. Thus, the message m is encrypted using the one-time pad α, which is known by the receiver. Lists of ciphertexts are processed accordingly.

4 Non-threshold Access Structures

Recall that the secret key of the public key generated by most mix-nets is secret shared with a threshold access structure, i.e., if there are k mix-servers a threshold λ can be chosen such that any set of λ mix-servers can decrypt, but no set of at most $\lambda - 1$ mix-servers can gain any knowledge of the encrypted plaintexts.

There is a rich literature on how to achieve other access structures [21], but in this paper we focus on deriving as much functionality as possible in a practical blackbox way from the key generation functionality of a standard mix-net.

The standard assumption in electronic voting systems is that there are k equally trusted/untrusted parties that can play the roles of mix-servers. In practice, this is not necessarily a plausible assumption. Setting up a secure server

and maintaining it is difficult and expensive. Thus, it is worthwhile to consider more flexible access structures. The problem is that simply implementing one of the more flexible protocols available in the cryptographic literature brings added software complexity, more complex specifications, and makes it more difficult to implement verifiers of universally verifiable proofs of correctness. Hence, it is a good idea to investigate alternatives.

Suppose that we have public keys $pk_1, \ldots, pk_t$ generated by different, possibly non-disjoint, sets $S_1, \ldots, S_t$ of mix-servers.

4.1 Simple Special Cases

Suppose that $t = 2$ and $pk_1 = (g, y)$ and $pk_2 = (g, z)$. Then it is obvious that a single public key (g, yz) can be formed, and that ciphertexts encrypted under this public key can only be decrypted if both S_1 and S_2 are able to decrypt. This is readily generalized to multiple sets of mix-servers. Conversely, if the same plaintext m is encrypted using the public keys $pk_1, \ldots, pk_t$ along with a zero-knowledge proof that this is the case, then any set S_i of mix-servers can decrypt.

These simple constructions show that we can easily handle 1-out-of-t as well as t-out-of-t access structures on top of the threshold access structure of each set of mix-servers.

4.2 Nested Threshold Access Structure

A sender generates an additional public key pk, which is only used for commitment purposes. It interprets its message as the first component $\Gamma_0 \in \mathcal{M}$ of Feldman's verifiable secret sharing scheme, and chooses $\Gamma_i \in \mathcal{M}$ randomly for $1, \ldots, \delta - 1$. Then it forms encryptions $D_i = \mathsf{Enc}_{pk}(\Gamma_i)$ to be used as commitments, computes $\alpha_l = \prod_{s=0}^{\delta-1} \Gamma_s^{l^s}$ for $l = 1, \ldots, t$, and forms encryptions $w_l = \mathsf{Enc}_{pk_l}(\alpha_l)$. Finally, it combines these ciphertexts to a tuple $\big((D_i)_{i\in[0,\delta-1]}, (w_l)_{l\in[t]}\big)$ and provides a zero-knowledge proof that w_l and $\prod_{s=0}^{\delta-1} D_s^{l^s}$ encrypt the same message (under the public keys pk and pk_l). Note that the list $(w_l)_{l\in[t]}$ can be interpreted as a ciphertext for a generalized El Gamal cryptosystem when it is shuffled.

This is effectively a Feldman-like verifiable secret sharing scheme such that any subset Δ of at least δ plaintexts allows reconstructing the plaintext

$$\Gamma_0 = \prod_{l\in\Delta} \alpha_l^{c_l} \ .$$

where c_l is the lth Lagrange coefficient. We remark that the D_i's are used as commitments, they are never decrypted, and they could be replaced by Pedersen commitments.

Which Mix-servers Must Shuffle? If the sets $S_1, \ldots, S_t$ are disjoint, then at least λ_l mix-servers from S_l must take part in the shuffling, where λ_l is the threshold number of mix-servers needed to decrypt ciphertexts encrypted under the public key pk_l. This allows us to consider the full sequence of shufflings as a sequence of t "mix-servers". Strictly speaking a smaller number of servers than $\sum_{l\in[t]} \lambda_l$ is needed if the sets of mix-servers are not disjoint, but using this optimization in practice is error prone.

5 Group Homomorphisms with Random Scalars

Suppose that one is given ciphertexts of the form $(u_i, v_i) = \mathsf{Enc}_{pk}(m_i)$ for $i = 1, \ldots, N$ and wish to compute ciphertexts of the form $(u'_i, v'_i) = \mathsf{Enc}_{pk}(m_i^z)$ for some random value $z \in \mathcal{R}$, or indeed simply m_i^z. The original ciphertexts may have been shuffled. This is needed in so-called identity shuffling. A special purpose protocol for this is easily derived from [32].

The mix-servers run the key generation phase as usual to generate a public key pk and corresponding secret key, but they run it an additional time to generate a public key pk' for which the corresponding secret key is a randomly chosen $z \in \mathcal{R}$.

To form the ciphertexts (u'_i, v'_i) they decrypt the ciphertexts

$$\big((u_i^{-1}, v_i^{-1}), (1, 1)\big)_{i\in[N]}$$

viewed as ciphertexts in $\mathcal{M}_{1,2} \times \mathcal{M}_{1,2}$ with the same public key pk' to get "plaintexts" $(u'_i, v'_i)_{i\in[N]} = (u_i^z, v_i^z)_{i\in[N]}$. If these are decrypted, then the plaintexts are of the form m_i^z.

The above is in fact all we need to compute any homomorphism with random scalars under encryption. To see this, given secretly shared random values $z_1, \ldots, z_t \in \mathcal{R}$, and ciphertexts of the form $\mathsf{Enc}_{pk}(m_{1,i}), \ldots, \mathsf{Enc}_{pk}(m_{t,i})$ we can compute ciphertexts of the form $\mathsf{Enc}_{pk}(m_{1,i}^{z_1}), \ldots, \mathsf{Enc}_{pk}(m_{t,i}^{z_1})$, which in turn allows us to compute a ciphertext of the form $\mathsf{Enc}_{pk}(\prod_{j=1}^{t} m_{j,i}^{z_j})$.

Remark 1 (Exponents may be biased). Careful readers may observe that the exponents z_i are not uniformly distributed when verifiable secret sharing is implemented directly using Feldman's scheme [9]. We are not aware of any natural example where this matters. We refer the reader to Gennaro et al. [14] for a discussion about this issue.

This problem can be resolved using Pedersen VSS [26] and a slight modification to the decryption protocol, but we are not aware of any mix-net where this is implemented.

When the number of servers is small the bias is so small that the biased distribution is relatively close to the uniform distribution, so unless there is a large number of secret shared values, any attack on the system that assumes samples from the biased distribution immediately translates to an attack that does not require this, albeit with a slightly worse success probability.

6 Weak Universal Shuffle

For the El Gamal cryptosystem ciphertexts formed using any public key defined over group $\mathcal{M}$ are not only indistinguishable from each other under chosen plaintext attacks, they are indistinguishable from random elements in $\mathcal{C}$.

Furthermore, a public key $pk = (g, y)$ may be viewed as an encryption of $1 \in \mathcal{M}$. We can re-encrypt it as any other ciphertext by choosing $r \in \mathcal{R}$ and forming $\mathsf{Enc}_{pk}(1, r)pk$, but this has the same distribution as $pk^r = (g^r, y^r)$. This is interesting, since we can form a pair (pk, w) of a public key and an encryption under this key and re-encrypt both without reference to any external public key, i.e., we can choose $r, s \in \mathcal{R}$ randomly and compute $(pk', w') = (pk^r, \mathsf{Enc}_{pk}(1, s)w)$.

This allows construction of a mix-net that shuffles ciphertexts encrypted under *distinct* keys. Note that holder of a secret key can always identify any randomized version of its own public key. Thus, the strongest privacy property we can expect is weaker than what is common for mix-nets, namely that no knowledge of how the plaintexts and ciphertexts of non-corrupted parties are related is leaked. For correctness on the other hand, we can require full correctness and universal verifiability. The first paper we are aware of that makes use of these observations to construct a *universal shuffle* is Golle et al. [15], but their construction is different from the one given here.

The challenge in constructing an efficient proof of a universal shuffle is that the ciphertexts no longer share a common public key in the underlying group, which means that key techniques such as batching [2] fails. However, with a mild assumption on the public keys we can shuffle pairs of the above form using homomorphic maps instead of individually generated randomness for each pair, and this is straightforward using a blackbox mix-net as explained in Section 5.

6.1 Randomized Public Keys and Plaintext Awareness

Let $pk_1, \ldots, pk_N \in \mathcal{C}$ be public keys of the form $pk_i = (g_i, y_i)$, where $g_i \in \mathcal{M}_i$ is randomly chosen and $y_i = g_i^{x_{j_i}}$ for some $x_{j_i} \in \mathcal{R}$ for $j_i \in [N']$. In other words, some public keys may be randomized versions of the same public key. If the x_{j_i} are randomly chosen, then $(g_i^z, y_i^z)_{i \in [N]}$ is indistinguishable from a randomly chosen element in $\mathcal{M}^{2N}$ under the Decision Diffie-Hellman assumption.

To guarantee privacy the blackbox mix-net requires that each public key-ciphertext pair $\big(pk_i, \mathsf{Enc}_{pk_i}(m_i, r_i)\big)$ is submitted along with a mechanism for extracting the plaintext m_i, e.g., a straight-line extractable proof of knowledge, but here we also need to be able to extract $a_i \in \mathcal{R}$ such that $g_i = g^{a_i}$ and $y_i = g_i^{x_{j_i}}$. This is needed to ensure that the pair as a whole is non-malleable.

In practice we can define $g_i = \mathsf{H}(\text{"app, sid, id, counter"})$, where H is a random oracle and the input is unique for the application, session, submittor, and index. Another way is to let a mix-net re-encrypt public keys before use.

6.2 Protocol

A universal shuffle can now be implemented using a blackbox mix-net. The protocol takes as input $\big(pk_i, w_i\big)_{i\in[N]}$, where $pk_i = (g_i, y_i)$, $g_i \in \mathcal{M}$ and $x_{j_i} \in \mathcal{R}$ are random, and $y_i = g_i^{x_{j_i}}$ and $w_i = \mathsf{Enc}_{pk_i}(m_i)$ for some $m_i \in \mathcal{M}$. The output is indistinguishable from ciphertexts of the form

$$\big(\mathsf{Enc}_{pk_{\pi(i)}}(1, z_1)pk_{\pi(i)}, \mathsf{Enc}_{pk_{\pi(i)}}(1, z_2)w_{\pi(i)}\big)_{i\in[N]} \ ,$$

where $z_1, z_2 \in \mathcal{R}$ and π are randomly chosen. The protocol proceeds as follows:

1. generate a public key pk of width two,
2. form ciphertexts
$$\big(a_i, b_i\big)_{i\in[N]} = \big(\mathsf{Enc}_{pk}(pk_i, 0), \mathsf{Enc}_{pk}(w_i, 0)\big)_{i\in[N]} \ ,$$
3. shuffle to form permuted and re-encrypted ciphertexts
$$\big(a'_i, b'_i\big)_{i\in[N]} = \big(\mathsf{Enc}_{pk}(pk_{\pi(i)}), \mathsf{Enc}_{pk}(w_{\pi(i)})\big)_{i\in[N]}$$
4. compute $(a''_i, b''_i)_{i\in[N]} = \big((a'_i)^{z_1}, (a'_i)^{z_2}b'_i\big)_{i\in[N]}$, with random $z_1, z_2 \in \mathcal{R}$, and
5. decrypt $(a''_i, b''_i)_{i\in[N]}$ relative pk.

By construction z_1 and z_1 maps down to the permuted public key $pk_{\pi(i)}$ and ciphertext $w_{\pi(i)}$, so the output is of the form $\big(pk_{\pi(i)}^{z_1}, pk_{\pi(i)}^{z_2}w_{\pi(i)}\big)_{i\in[N]}$ as claimed.

We ignore inputs from corrupted parties above to illustrate how privacy is provided for non-corrupted parties, but the inputs from corrupted parties are processed in the same way.

7 Shuffling Hypergraphs with Encrypted Vertex Labels

Suppose that we are given a hypergraph $G = (V, E)$ and an encryption w_v of some label for each vertex $v \in V$, i.e., we have a public hypergraph with encrypted vertex labels. Our goal is to apply a random isomorphism to the hypergraph and at the same time re-encrypt the encryptions of the labels to hide how the resulting hypergraph is related to the original.

Example 1 (Pre-computed Ciphertexts). In the Prêt à Votér [6] submission scheme each voter is handed a set of pre-computed choice-ciphertexts pairs sorted lexicographically with respect to the ciphertexts and printed on paper. Generating the sets of ciphertexts can be done as in the motivating example. To transfer the plaintexts to the printer we simply swap to encryption under the printer's public key or one-time pads.

We may think of the encrypted choices of a voter as the encrypted labels on a tree with a unique root representing the voter, so we may view the set of all encrypted labels as a forest to be shuffled and hence a simple example of shuffling a graph.

Example 2 (Rotation). There are efficient special purpose protocols for proving a rotation shuffle of homomorphic ciphertexts introduced by de Hoogh et al. [7]. We can simply encode the directed cyclic graph, view the ciphertexts as encrypted labels, and shuffle the graph to achieve the same result.

Our method does not give a way to enforce the use of a permutation that is *not* an isomorphism of a certain graph as in [30].

7.1 Simplified Notation

In Section 2 we have given an explicit exposition of the fact that we can shuffle lists of ciphertexts, encrypted under the same or different public keys, but the standard notation is not convenient for our purposes.

In the following we write $\langle m \rangle_{pk}$ for an encryption of a message m under a public key pk. We stress that m may be a tuple $(m_1, \ldots, m_k)$ of elements, in which case we write $\langle m_1, \ldots, m_k \rangle_{pk}$. When the public key is understood from the context, then we drop it from our notation.

7.2 Encoding Edges in Undirected Hypergraphs

The basic idea is simple: we generate encryptions of unique tags (values in the message space) and use these to encode edges similarly to the motivating example by prepending ciphertexts with the encryptions of tags. Then we shuffle the resulting longer ciphertexts and only decrypt the ciphertexts containing the tags. This allows identifying the edges of the randomized hypergraph with re-encrypted encryptions of the labels.

For each edge $e = \{u, v\}$ between vertices labeled by ciphertexts $\langle m_u \rangle$ and $\langle m_v \rangle$ we generate an encryption $\langle \alpha_e \rangle$ of a random tag α_e and form the ciphertexts $\langle \alpha_e, m_u \rangle$ and $\langle \alpha_e, m_v \rangle$. Then we shuffle the resulting twin ciphertexts and only decrypt the tags to reveal which re-encrypted encryptions of labels belong to the same edge. However, most vertices are contained in more than one edge, e.g., u could be contained in an edge $e' = \{u, v'\}$. Naively we could add an encryption of another tag $\alpha_{e'}$ to form $\langle \alpha_{e'}, \alpha_e, m_u \rangle$ and $\langle \alpha_{e'}, m_{v'} \rangle$.

The difference in lengths is not a big problem, since the degree of each vertex will be public in the output isomorphic hypergraph anyway. In other words the ciphertexts of each length can be shuffled separately, or we can add dummy ciphertexts of the form $\langle \perp \rangle$ for a symbol $\perp$ that is not used as an tag.

The problem is that the order in which we add the encrypted tags will reveal information about how the encrypted labels in the output correspond to the encrypted labels of the original hypergraph, but this is easily resolved by using the mix-net to shuffle the order of the tags.

For each vertex u, we form the ciphertexts $\langle \alpha_{e_1} \rangle, \ldots, \langle \alpha_{e_k} \rangle$, where $E_u = \{e_i\}_{i \in [k]}$ is the set of edges containing u, and shuffle them. We can apply the trick from the motivating example to shuffle the set of encrypted tags of all vertices in a single execution of the mix-net. We can then re-combine the ciphertexts giving a longer encryption of the tags in random order. This permits us to abuse

notation and denote the result by $\langle \beta_u, m_u \rangle$, where $\beta_u = \{\alpha_e\}_{e \in E_u}$, i.e., we can assume that we can use encryptions of *unordered sets* of tags.

The key properties we need from the encrypted random tags is that they are secret, unique, and matched with edges randomly. We do not require that the tags themselves are randomly chosen. We consider several ways of generating such encrypted tags in Section 7.4.

7.3 Directed Hypergraphs

Encoding directed hypergraphs is readily done by partitioning the set of tags of a vertex into groups where each group contains the tags of edges in which the vertex appears at a given position. For example, if a vertex u has an encrypted label $\langle m_u \rangle$, then its complete ciphertext would be

$$\langle \beta_1, \circ, \beta_2, \circ, \beta_3, \circ, \ldots \beta_s, m_u \rangle \ ,$$

where β_i is the set random tags of edges where it appears as the ith vertex. Here we use an encryption of a special symbol $\circ$, different from both $\perp$ and all tags, to divide the random tags into groups.

For graphs where each vertex appears approximately the same number of times at each position in some edge, we can instead add the needed number of dummy encryptions of $\perp$ and avoid using $\circ$ at all, since the sets can be inferred. What is most efficient depends on the application.

7.4 On Encrypted Tags

The most obvious and general way to generate the encrypted tags we need is to simply form encryptions $\langle 1 \rangle, \langle 2 \rangle, \ldots, \langle |E| \rangle$ using fixed randomness, shuffle the list of ciphertexts, and use the resulting ciphertexts $\langle \alpha_1 \rangle, \langle \alpha_1 \rangle, \ldots, \langle \alpha_{|E|} \rangle$. This means that the ith edge e_i is assigned a randomly chosen tag α_i for every index $i \in [|E|]$.

The most efficient approach in the random oracle model is to simply interpret the output of a random oracle evaluated over a suitable session identifier as ciphertexts $\langle \alpha_1 \rangle, \langle \alpha_1 \rangle, \ldots, \langle \alpha_{|E|} \rangle$. In this case the indices will be randomly chosen in $\mathcal{M}$. If this space is exponentially large, then the indices will be distinct with overwhelming probability.

If the message space $\mathcal{M}$ is a dense subgroup of a multiplicative group, then the indices can be chosen as small unique primes $p_e \in \mathcal{M}$ for $e \in E$. The advantage of this approach is that $\langle \{p_e\}_{e \in E_u} \rangle$ can be represented as $\langle \prod_{e \in E_u} p_e \rangle$ in a single ciphertext provided that E_u is not too large. If it is too large, then we can still use the general approach and prepare encrypted indices in random order that are used and combined as products in single ciphertexts. Recovering the individual primes is done by fast factorization.

We can in fact use the idea of compacting multiple encrypted indices into one for any known set of indices. The only property we need is unique factorization over the set of indices, which is the case with overwhelming probability for

randomly chosen indices, but we can of course also define $\alpha_{e_i} = g^{2^{i-1}}$ for the ith edge e_i to ensure this. The problem is that factoring (or computing discrete logarithms) is expensive, so this is only feasible when the degree of each vertex is small.

Again, what is the fastest approach depends on the application.

7.5 Restricting the Set of Isomorphisms

So far we have only considered the full set of isomorphisms of a graph, but it is quite natural to consider cases where some edges can be identified, e.g., a set of disjoint and possibly different labeled graphs could be enumerated and an encrypted label added to each, with edges to all its vertices, to keep track of each graph (generalization of motivating example). To achieve this we would simply encrypt *publicly known* and unique tags using fixed randomness.

More generally, we can start with the generic construction for any graph and add encryptions of publicly known edge tags or vertex labels to effectively reduce the set of isomorphisms to those that keep these edges or vertices fixed.

7.6 Shuffling Graphs Spanning Multiple Sessions

Multiple sessions correspond to using a single session with publicly known unique tags to not only separate inputs in the different sessions, but also to identify which input belongs to which session. This cannot be avoided, since ciphertexts from different sessions are not mixed. Thus, using multiple sessions simply imposes a natural restriction on the set of admissible isomorphisms of the labeled graph.

Strictly speaking the generalization in Section 2.2 is not needed to process ciphertexts containing components encrypted under different public keys as units, since we can consider the graph with edges between the ciphertexts in different sessions. Processing ciphertexts encrypted under different public keys separately in different sessions and using tags to link them is convenient if the inputs come in batches, but it also allows a tradeoff in practice between this approach and using the generalization of Section 2.2.

To see this, note that the straightforward way to handle both multiple public keys and multiple ciphertexts is to think of plaintexts and ciphertexts as rectangles. The width corresponds to how many ciphertexts there are and the height corresponds to the number of keys used. This maps directly to the notation in Section 2.2 and Section 2.3.

However, more complex configurations are possible, e.g., we could have three ciphertexts encrypted with one public key and two with another. Handling this in: the mix-net implementation, configuration files, documentation, the description of the universally verifiable proof of correctness, and in verifiers' code is adds too much complexity.

Combining dummy ciphertexts, the generalization of El Gamal to multiple public keys, and linking of ciphertexts gives a rich set of possible tradeoffs in practice.

7.7 Adding Edge Labels

We can handle encrypted edge labels in addition the encrypted vertex labels by adding vertices to carry the edge labels. Concretely, for each edge e with encrypted label $\langle m_e \rangle$ we add a vertex u_e to e and interpret $\langle m_e \rangle$ as its vertex label. Then we add two edges e_{old} and e_{new} with publicly known tags containing all original vertices, and all added vertices, respectively. This is equivalent to shuffling the edge labels in a separate session.

7.8 Dual Construction and Encrypted Edge Labels

Note that our construction correspond to representing a graph (V, E) by enumerating the vertices and edges and representing structure by the vertex-edge set pairs $\{(i, J_{u_i}) \mid i \in [|V|]\}$, where $J_u = \{j \mid u \in e_j\}$. We present this construction since it seems most natural that vertices have encrypted labels.

However, if the edges have encrypted labels, then it is more natural to represent it using the index-edge pairs $\{(j, I_{e_j}) \mid e_j \in E\}$, where $I_e = \{i \mid u_i \in e\}$. The encrypted random edge tags are then replaced by encrypted random vertex tags, and we can also add vertex labels by adapting the method of Section 7.7. In applications we choose the most convenient representation.

8 Semi-blackbox Construction for Swapping Public Keys

We end with a construction that is not fully blackbox in that it necessarily uses the structure of the cryptosystem, but only uses blackbox calls to the mix-net to swap encryption under one public key for encryption under another public key.

Consider an encryption of m of the form $(u, v) = (g^r, y^r m)$ under a public key (g, y) and let (g, z) be another public key. We can simply let the mix-net decrypt $(u^{-1}, 1)$ relative to (g, z) to get a plaintext of the form $v' = z^r$, and form $(u, v'v) = (g^r, (yz)^r m)$. Re-encrypting this relative (g, yz) gives a ciphertext of the form $(g^s, (yz)^s m)$, which can be decrypted relative (g, y) to get the plaintext $z^s m$ from which the ciphertext $(g^s, z^s m)$ can be formed.

Observe that a simulator that holds r can compute both m and v' without knowing the logarithms of y or z, so provided that the mix-net is secure the protocol has the same effect as having a trusted party that holds both secret keys and outputs the intermediate results.

For multiple ciphertexts we can modify the mix-net to use the identity permutation, but we can also use the trick from the motivating example.

9 Security

To prove the overall security of a protocol that makes use of the constructions presented here we need rigorous analysis, but due to the diversity of the constructions this is out of scope in this paper.

However, standard methods from cryptography can be used to argue that the constructions behave as if a trusted party performed the mix-net operations. All we need to observe is that every secret exponent that appears in the use of the mix-net can be extracted by a simulator and decryption can be simulated. If we require that the simulator can extract the randomness used to form all input ciphertexts, then this suffices. To what extent ciphertexts must be distinct is application dependent and out of scope in this paper.

10 Complexity

Complexity estimates for mix-nets made in the literature are often overly pessimistic, and benchmarks for implementations of proofs of shuffles in isolation do not take into account the tradeoffs needed in practice to properly implement a complete mix-net.

We refer the reader to the benchmarks and heuristic complexity analysis of VMN [34] for more information and invite the reader to run the benchmark suite for any parameters of particular interest. We are not aware of any other implementation of a mix-net with a serious benchmark.

The complexity of a shuffle for a fixed security parameter is $O(\lambda N \kappa \omega)$ where the threshold is λ, the number of plaintexts is N, the key width is κ, and the width of each plaintext is ω. For fixed λ, κ, and ω, the complexity is of the form $aN + b$.

However, for fixed λ and N it is of the form $a\kappa\omega + b\kappa + c\omega + d$ where a, b, and c are notably smaller than d. For example, processing ciphertexts of width ten takes roughly three times as much time as shuffling single ciphertexts for some standard security parameters.

Moreover, the running time of decryption is almost independent of the number of mix-servers due to a parallelized variation of the standard distributed decryption protocol (see [33] and the source code for details). This applies to verification of the universally verifiable proof as well.

Thus, the constructions are less costly than the reader may expect, certainly fully practical, and suffices in all applications except those with the most stringent time constraints.

References

1. Verificatum AB. Verificatum site. `http://www.verificatum.org`, January 2017.
2. M. Bellare, J.A. Garay, and T. Rabin. Batch verification with applications to cryptography and checking. In *LATIN*, volume 1380 of *Lecture Notes in Computer Science*, pages 170–191. Springer Verlag, 1998.
3. Jonathan Ben-Nun, Niko Fahri, Morgan Llewellyn, Ben Riva, Alon Rosen, Amnon Ta-Shma, and Douglas Wikström. A new implementation of a dual (paper and cryptographic) voting system. In Manuel J. Kripp, Melanie Volkamer, and Rüdiger Grimm, editors, *5th International Conference on Electronic Voting 2012, (EVOTE*

2012), Co-organized by the Council of Europe, Gesellschaft für Informatik and E-Voting.CC, July 11-14, 2012, Castle Hofen, Bregenz, Austria, volume 205 of *LNI*, pages 315–329. GI, 2012.

4. Philippe Bulens, Damien Giry, and Olivier Pereira. Running mixnet-based elections with helios. In Hovav Shacham and Vanessa Teague, editors, *2011 Electronic Voting Technology Workshop / Workshop on Trustworthy Elections, EVT/WOTE '11, San Francisco, CA, USA, August 8-9, 2011*. USENIX Association, 2011.
5. D. Chaum. Untraceable electronic mail, return addresses and digital pseudo-nyms. *Communications of the ACM*, 24(2):84–88, 1981.
6. David Chaum, Peter Y. A. Ryan, and Steve A. Schneider. A practical voter-verifiable election scheme. In Sabrina De Capitani di Vimercati, Paul F. Syverson, and Dieter Gollmann, editors, *Computer Security - ESORICS 2005, 10th European Symposium on Research in Computer Security, Milan, Italy, September 12-14, 2005, Proceedings*, volume 3679 of *Lecture Notes in Computer Science*, pages 118–139. Springer, 2005.
7. S. de Hoogh, B. Schoenmakers, B. Skoric, and J. Villegas. Verifiable rotation of homomorphic encryptions. In *Public Key Cryptography – PKC 2009*, volume 5443 of *Lecture Notes in Computer Science*, pages 393–410. Springer Verlag, 2009.
8. Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.
9. P. Feldman. A practical scheme for non-interactive verifiable secret sharing. In *28th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 427–438. IEEE Computer Society Press, 1987.
10. Bern University for Applied Sciences. Univote site. `https://www.univote.ch`, January 2017.
11. J. Furukawa. Private communication., April 2017.
12. J. Furukawa, H. Miyauchi, K. Mori, S. Obana, and K. Sako. An implementation of a universally verifiable electronic voting scheme based on shuffling. In *Financial Cryptography 2002*, volume 2357 of *Lecture Notes in Computer Science*, pages 16–30. Springer Verlag, 2002.
13. J. Furukawa and K. Sako. An efficient scheme for proving a shuffle. In *Advances in Cryptology – Crypto 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 368–387. Springer Verlag, 2001.
14. R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Secure distributed key generation for discrete-log based cryptosystems. *J. Cryptology*, 20(1):51–83, 2007.
15. Philippe Golle, Markus Jakobsson, Ari Juels, and Paul F. Syverson. Universal re-encryption for mixnets. In Tatsuaki Okamoto, editor, *Topics in Cryptology - CT-RSA 2004, The Cryptographers' Track at the RSA Conference 2004, San Francisco, CA, USA, February 23-27, 2004, Proceedings*, volume 2964 of *Lecture Notes in Computer Science*, pages 163–178. Springer, 2004.
16. J. Groth. A verifiable secret shuffle of homomorphic encryptions. In *Public Key Cryptography – PKC 2003*, volume 2567 of *Lecture Notes in Computer Science*, pages 145–160. Springer Verlag, 2003.
17. Rolf Haenni and Eric Dubuis. Private communication., February 2016.
18. Rolf Haenni and Oliver Spycher. Secure internet voting on limited devices with anonymized dsa public keys. In *Proceedings of the 2011 Conference on Electronic Voting Technology/Workshop on Trustworthy Elections*, EVT/WOTE'11, pages 8–8, Berkeley, CA, USA, 2011. USENIX Association.

19. J. Håstad, R. Impagliazzo, L. A. Levin, and M. Luby. Construction of a pseudorandom generator from any one-way function. *SIAM Journal on Computing*, 28(4):1364–1396, 1999.
20. Helios. Helios site. `http://www.heliosvoting.org`, January 2017.
21. Martin Hirt and Ueli M. Maurer. Player simulation and general adversary structures in perfect multiparty computation. *J. Cryptology*, 13(1):31–60, 2000.
22. Markus Jakobsson and Ari Juels. Mix and match: Secure function evaluation via ciphertexts. In Tatsuaki Okamoto, editor, *Advances in Cryptology - ASIACRYPT 2000, 6th International Conference on the Theory and Application of Cryptology and Information Security, Kyoto, Japan, December 3-7, 2000, Proceedings*, volume 1976 of *Lecture Notes in Computer Science*, pages 162–177. Springer, 2000.
23. A. Neff. A verifiable secret shuffle and its application to e-voting. In *8th ACM Conference on Computer and Communications Security (CCS)*, pages 116–125. ACM Press, 2001.
24. nVotes. nvotes site. `https://nvotes.com`, March 2019.
25. C. Park, K. Itoh, and K. Kurosawa. Efficient anonymous channel and all/nothing election scheme. In *Advances in Cryptology – Eurocrypt '93*, volume 765 of *Lecture Notes in Computer Science*, pages 248–259. Springer Verlag, 1994.
26. T. P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *Advances in Cryptology – Crypto '91*, volume 576 of *Lecture Notes in Computer Science*, pages 129–140. Springer Verlag, 1992.
27. Kim Ramchen and Vanessa Teague. Parallel shuffling and its application to prêt à voter. In *Proceedings of the 2010 International Conference on Electronic Voting Technology/Workshop on Trustworthy Elections*, EVT/WOTE'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.
28. Peter Y. A. Ryan, Peter B. Rønne, and Vincenzo Iovino. Selene: Voting with transparent verifiability and coercion-mitigation. In Jeremy Clark, Sarah Meiklejohn, Peter Y. A. Ryan, Dan S. Wallach, Michael Brenner, and Kurt Rohloff, editors, *Financial Cryptography and Data Security - FC 2016 International Workshops, BITCOIN, VOTING, and WAHC, Christ Church, Barbados, February 26, 2016, Revised Selected Papers*, volume 9604 of *Lecture Notes in Computer Science*, pages 176–192. Springer, 2016.
29. A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.
30. B. Terelius and D. Wikström. Proofs of restricted shuffles. In *Africacrypt 2010*, volume 6055 of *Lecture Notes in Computer Science*, pages 100–113, 2010.
31. Agora Voting. Agora voting site. `https://www.agoravoting.com`, January 2017.
32. D. Wikström. A sender verifiable mix-net and a new proof of a shuffle. In *Advances in Cryptology – Asiacrypt 2005*, volume 3788 of *Lecture Notes in Computer Science*, pages 273–292. Springer Verlag, 2005.
33. D. Wikström. How to implement a stand-alone verifier for the Verificatum mix-net. `http://www.verificatum.org`, October 2011.
34. D. Wikström. Complexity analysis of the Verificatum mix-net. `http://www.verificatum.org`, October 2016.