Zero-Knowledge Proofs for Mix-nets of Secret Shares and a Version of ElGamal with Modular Homomorphism *

Marius-Călin Silaghi
Florida Institute of Technology

May 1, 2005

Abstract

Mix-nets can be used to shuffle vectors of shared secrets. This operation can be an important building block for solving combinatorial problems where constraints depend on secrets of different participants. A main contribution of this paper is to show how participants in the mix-net can provide Zero-Knowledge proofs to convince each other that they do not tamper with the shuffled secrets, and that inverse permutations are correctly applied at unshuffling. The approach is related to the proof of knowing an isomorphism between large graphs. We also make a detailed review and comparison with rationales and analysis of Chaum's and Merritt's mix-nets.

Another contribution is a $(+ \bmod \nu, \times)$-homomorphic encryption scheme that can be parametrized by a public prime value ν and that is obtained from a version of ElGamal [DGS02]. This cryptosystem allows for guarantees of security in the aforementioned mix-net. A generalization shows how to obtain modular arithmetic homomorphic schemes from other cryptosystems.

Mix-nets offer only computational security since participants get encrypted versions of all the shares. Information theoretically secure algorithms can be obtained using secure arithmetic circuit evaluation. The arithmetic circuit previously proposed for shuffling a vector of size k was particularly slow. Here we also propose a new arithmetic circuit for performing the operation in $O(k^2)$ multiplications and requiring $k-1$ shared random numbers with different domains. Another contribution is to provide more efficient arithmetic circuits for combinatorial optimization problems, exploiting recent secure primitives. Examples are shown of how these techniques can be used in the Secure Multi-party Computation (SMC) language [Sil04b]. SMC's procedures for generating uniformly distributed random permutations are also detailed.

1 Introduction

It is known that any probabilistic function on an arithmetic finite field can be securely computed using addition/multiplication or AND/XOR logic operators on shared secrets [BOGW88]. In securely addressing large problems, like auctions and scheduling, a major step consists in randomly selecting one out of several possible results. This is an important problem since not only the fairness of the MPC, but also a lot of privacy implications depend heavily on the way it is addressed. Details about the privacy implications of the process for selecting the result are detailed in [SR04].

Here we describe computationally secure techniques for selecting randomly a solution when several solutions are possible. This can be done with information theoretical security by using solely addition/multiplication or AND/XOR operations on secret shares [Sil04c]. Previous solutions of this type were slow, and now we propose a faster version. The approach addressed here to select a value randomly from a certain set is based on the concept of mix-net and yields faster algorithms.

Mix-nets were introduced by Chaum as a way to provide anonymity in an insecure Internet [Cha81]. The term mix-net suggests a network of volunteers that mixes messages to hide their relation with the senders.

*This version contains some additions to the version submitted on March 8, 2005. ZK-proofs for the inverse permutations were added on April 16. Some citations are added and a section is deleted on April 30. ElGamal version with modular homomorphism described on May 1.

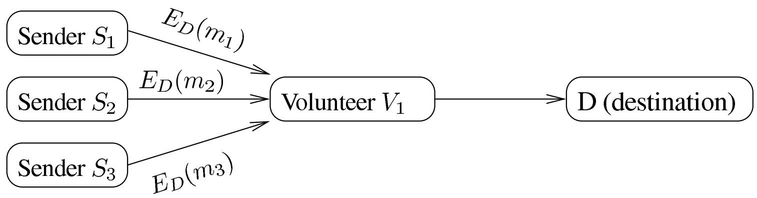

Figure 1: Using a volunteer V_1 which shuffles messages from different senders before delivering them to the destination.

The technique relies on the trust in a set of volunteers where the collusion of all of them is required to break the anonymity. Secrets from many volunteers are combined in a more secure secret that hides the identity of a message sender. After we describe Chaum's mix-nets we also pay attention to a variation by Merritt [Mer83], very similar to what is used for secure function evaluation.

The main part of the article is then spent in describing and analyzing a mix-net that can shuffle a set of shared secrets such that no participant can know to which of the initial secrets does an obtained secret correspond. This technique can be used to select a random value from a set of secrets, such that the selection is fair and nobody can know which secret was selected. [Sil04c] shows how this can be used to select a value with certain properties (namely being a solution to a problem at hand) from a set of alternatives with secret properties. At the end we describe some related arithmetic circuits.

2 Chaum's Anonymity Mix-net

In an Internet where messages can be supervised and where the identity of their sender can be verified, the accountability is expected to offer us less unsolicited email, better security and a better satisfaction in general. However, there exist cases where anonymity of the sender is desirable. A typical case is voting and submission of signatures for popular initiatives [KS05, SK05, AS04]. In such applications we want that the vote/signature authors remain as anonymously as possible (and eligibility is proven using some type of digital signature, called credentials).

Trying to provide anonymity, Chaum's idea was to ask help from volunteers that accept to forward the message to the destination and to remove the name of the sender (see Figure 1). If the volunteers act for many messages simultaneously shuffling them before delivery, then (even knowing who gave messages to the volunteers) the receiver cannot know the mapping between senders and received messages. To avoid that the volunteers see the messages, these messages can be encrypted with the key of the destination.

Assume that sender S gives $E_D(m)$ to volunteer V_1 asking delivery to the destination D. However, if V_1 is coerced to reveal his secret shuffling then the anonymity is lost. A small improvement is achieved if the sender does not give his message directly to V_1 but asks an additional volunteer V_2 to anonymously deliver the message to V_1 (assuming V_1 knows to always deliver to D). Volunteer V_2 acts for many senders and shuffles the messages, too. The security is now based on two volunteers. To avoid that V_2 can be corrupted by D and reveal the messages directly to D together with the senders, Chaum found a way to force V_2 and D to also need V_1. The idea was to perform an additional encryption of the message with a key known only by V_1, namely a cipher E_1. The new message to be sent by the sender is $M = E_1(E_D(m))$. V_2 and D cannot use the message M if V_1 does not decrypt it. In this way, both volunteers have to be corrupted in order to find the mapping between senders and messages received by D. Note that in the two volunteers version shown in Figure 2, each volunteer is specialized as either delivering all messages straight to the destination (the case of V_1), or delivering to a volunteer of the previous type (like V_2 which delivers to V_1).

The concept of performing several encryptions one over another is called *onion* due to the resemblance between the layers of encryption around the message and the layers of an onion. The next improvement proposed by Chaum is a way to avoid the need of specializing volunteers as being either the first or the last on a path. Chaum did this by telling each volunteer separately for each message the identity of the next hop on the path. Such a message has the format $M_2 = \langle V_1||E_1(D||E_D(m))\rangle$ (see Figure 3), where $a||b$ denotes the concatenation of a and b.

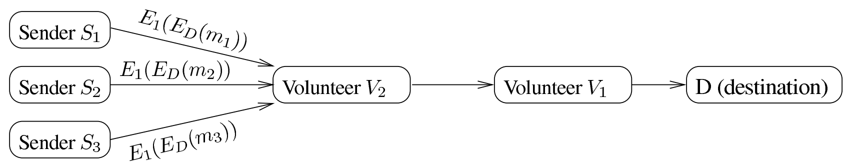

Figure 2: Onion: Using two specialized volunteers.

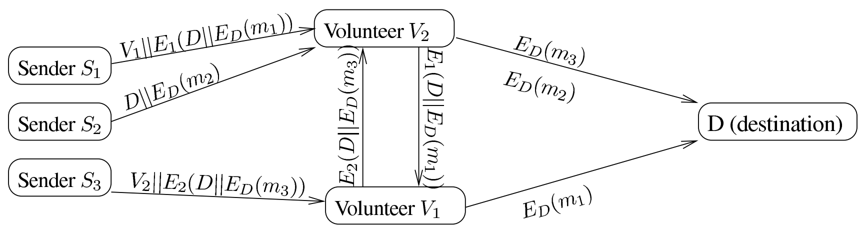

Figure 3: Onion: Using up to two unspecialized volunteers.

An additional advantage of the lack of specialization is that we can easily extend the protocol to use three volunteers (which all need to collude with D in order for D to break the anonymity). Namely, given an additional volunteer V_3, this additional volunteer can be asked by S to be an intermediary in anonymously handing M_2 to V_2. As with the introduction of V_2, to avoid that V_3 sends the message directly to D without needing V_2, M_2 will be encrypted with a key known to V_2, getting $E_2(M_2)$. The next hop, V_2, is also communicated to V_3. The obtained message is:

$$M_3 = \langle V_2||E_2(V_1||E_1(D||E_D(m)))\rangle$$

Since the identity of the hop to which V_2 will send its message (i.e., V_1) is encrypted with the key of V_2, V_3 will not know the path that will be followed by the message. With three volunteers, V_2 is the single one knowing all the three hops on the path of M_3.

Even more, similarly to the addition of V_2 and V_3 one can add as many volunteers as desired increasing the security up to the level of volunteers needed by the sender to trust at least one of them. Given a message M_k built to generate a path with k hops, one can set a path with $k+1$ nodes by adding a new volunteer, V_{k+1}, and handing him the message $M_{k+1} = \langle V_k||E_k(M_k)\rangle$. One can easily verify by mathematical induction that the message M_k that has to be sent to a volunteer V_k to make security of anonymous delivery dependent on k volunteers $V_1, ..., V_k$ is:

$$M_k = \langle V_{k-1}||E_{k-1}(V_{k-2}||E_{k-2}(..., E_2(V_1||E_1(D||E_D(m)))...))\rangle$$

Often, the message is sent to V_k encrypted with E_k, both to guarantee that nobody seeing the message learns the identity of V_{k-1}, and to make V_k's procedure identical to the procedure of all other volunteers. I.e.,

$$M_k = E_k(V_{k-1}||E_{k-1}(V_{k-2}||E_{k-2}(..., E_2(V_1||E_1(D||E_D(m)))...)))$$

Moreover, making the processing homogeneous for all volunteers makes it possible for the sender to claim to be a volunteer himself [GRS96]. [XNJS04, AS04] proposed to use digital currency for paying servers in a mix-net. In [JJR02], it is shown how to improve the reliability of the servers by probabilistic checking.

To avoid that the path of a message can be analyzed by routers due to reduced traffic, it is possible to maintain a minimal traffic by generating fake messages between all volunteers, according to a Poisson distribution. This technique combined with adding random delays in processing each received message is sufficient to offer robustness to sniffers. To offer robustness from analysis of message sizes, the fake messages generated should always have the size of the recent messages that would have to be forwarded (fake or correct) [RR98, FM02].

Remark 1 *Mix-nets are computationally expensive due to the length of the messages. Public key encryption with growing messages can be performed with Pollig-Hellman cryptosystem. One can also use symmetric key encryption, sending only the session key in the onion encryption:*

$$M_k = E_k(K_k)||E_{K_k}(V_{k-1}||E_{k-1}(K_{k-1})||E_{K_{k-1}}(...||E_1(K_1)||E_{K_1}(D||E_D(m)...))$$

where E_{K_i} denotes symmetric encryption with the key K_i.

The usage of symmetric encryption as described in this remark is particularly useful when other data is passed to the volunteers, such as digital coins and/or reply-path data [AS04].

3 Merritt Election Protocol

As mentioned above, Chaum's mix-nets can be used for anonymously voting. From the perspective of voting, some of the drawbacks of Chaum's technique are that (a) timing analysis of vote arrival and sending can reduce anonymity, (b) anonymity depends on the honesty of the volunteers, (c) message delivery is not reliable.

3.1 Modifications to Chaum's method

Let us see some solutions to the three threats observed when Chaum's method is applied to voting:

(a) To avoid the timing analysis (especially if fake messages are not used), all the votes should be submitted simultaneously and should arrive simultaneously. For arriving simultaneously, the last volunteer on the chain should be the same for all messages. To avoid that this volunteer could help an attacker based on timing and path analysis, the previous volunteer should also be the same for all messages. The same reasoning can be applied to the cooperation between the last two volunteers and an attacker to infer that the third volunteer should also be the same for all votes. By induction we get that all the votes should travel using the same path, simultaneously.

(b) There is a natural bound on the anonymity that can be achieved in voting. Namely, if all participants except one reveal their vote, then the vote of the remaining participant is revealed. In Chaum's approach, if all volunteers are corrupted then the anonymity is lost. Note that if all voters are also volunteers and all messages are handled by all volunteers, then the security offered by Chaum's method is equal to the maximal security achievable for the problem.

(c) The fact that message delivery is not reliable comes from the fact that each volunteer can discard a message without forwarding it, and the identity of that volunteer cannot be established without the cooperation of all volunteers (which should be hard to get). Some approaches trying to fix this are based on statistical tests [JJR02], or on providing incentives [XNJS04, AS04]. A more secure approach, usable if we accept to reveal the trajectory of all messages (as in the aforementioned solution to problem (a)), is to ask all volunteers to broadcast signed versions of the messages that they forward. Since everybody sees the broadcast messages, everybody can detect the user that did not forward a message. Since each participant can see and recognize the forwarded component of his own message, everybody can detect when (if ever) his/her message is tampered with.

3.2 A simplified version of Merritt's protocol

These solutions to the three threats mentioned above are integrated in Merritt's election protocol [Mer83]. Several descriptions and versions of Merritt's election protocol exist [GB96, Gen95]. Merritt's election protocol ensures the privacy of the relation vote-elector by reordering (shuffling) the votes. The shuffling is obtained by a chain of permutations (each being the secret of an election center) on the encrypted votes. Note that maximal security would require each voter to act as election center and we will assume this to be the case.

The n election centers, $A_1, ..., A_n$, are ordered in a chain (called in the following, Merritt chain). Each A_i distributes a public key for a probabilistic public key cryptosystem, the obtained encryption function being denoted $E_i(m,r)$ where m is the plaintext and r the random value. A_i keeps corresponding private decryption function D_i, i.e., $D_i(E_i(m,r)) = m$. Each voter A_i that submits a vote v_i chooses a large random number h_i and random numbers $r_{1,i}, ..., r_{n,i}$, and computes a message $y_{1,i}$ to be submitted to the first volunteer in the Merritt chain, A_1:

$$y_{1,i} = E_1(E_2(...E_{n-1}(E_n(v_i||h_i, r_{n,1}), r_{n-1,i})..., r_{2,i}), r_{1,i})$$

By $v_i||h_i$ we denote the number obtained by concatenating v_i and h_i. The $y_{1,j}$ values gathered by A_1 in a vector $y_1 = \{y_{1,j}\}_j$ ordered according to their second index, j, are posted through the chain of participants in order from A_1 to A_n. Each A_i chooses a secret random permutation $\pi_i : [1..n] \rightarrow [1..n]$. We define the application of such a permutation on a vector as $\pi_i(\{x_k\}_{k\in[1..n]}) \stackrel{\text{def}}{=} \{x_{\pi_i^{-1}(k)}\}_{k\in[1..n]}$. After receiving $\{y_{1,j}\}_j$, A_1 broadcasts $y_2 = \{y_{2,j'}\}_{j'} = \pi_i(\{D_i(y_{i,j})\}_j)$. In each subsequent hop, A_i, of the Merritt chain, after receiving $\{y_{i,j^{(i-1)}(k)}\}_{j^{(i-1)}(k)}$, it broadcasts $\{y_{i+1,j^{(i)}(k)}\}_{j^{(i)}(k)} = \pi_i(\{D_i(y_{i,j^{(i-1)}(k)})\}_{j^{(i-1)}(k)})$. A_n publishes $\{y_{n+1,j^{(n)}}\}_{j^{(n)}}$ which equals $\{v_k||h_k\}_{j^{(n)}(k)}$. Votes have been shuffled since the relation $(j, j^{(n)})$ was lost. Each voter A_i can check that his own vote was recorded correctly by verifying the vote that appears in association with its random number h_i. If the random numbers are sufficiently large then they can be expected to be unique.

Example 1 *Assume A_1, A_2, and A_3 want to vote on an issue using the simplified Merritt protocol.*
A_1 submits to himself the vote $y_{1,1} = E_1(E_2(E_3(v_1||h_1, r_{3,1}), r_{2,1}), r_{1,1})$.
A_2 submits to A_1 the vote $y_{1,2} = E_1(E_2(E_3(v_2||h_2, r_{3,2}), r_{2,2}), r_{1,2})$.
A_3 submits to A_1 the vote $y_{1,3} = E_1(E_2(E_3(v_3||h_3, r_{3,3}), r_{2,3}), r_{1,3})$.

A_1 builds the vector $y_1 = \langle y_{1,1}, y_{1,2}, y_{1,3}\rangle$, chooses the permutation $\pi_1 = \langle 2, 3, 1\rangle$ and applies it on the decrypted elements of y_1 obtaining: $y_2 = \langle D_1(y_{1,2}), D_1(y_{1,3}), D_1(y_{1,1})\rangle$ $= \langle E_2(E_3(v_2||h_2, r_{3,2}), r_{2,2}), E_2(E_3(v_3||h_3, r_{3,3}), r_{2,3}), E_2(E_3(v_1||h_1, r_{3,1}), r_{2,1})\rangle = \langle y_{2,1}, y_{2,2}, y_{2,3}\rangle$. y_2 is broadcast signed by A_1.

A_2 chooses the permutation $\pi_2 = \langle 1, 3, 2\rangle$ and applies it on the decrypted elements of y_2 obtaining: $y_3 = \langle D_2(y_{2,1}), D_2(y_{2,3}), D_2(y_{2,2})\rangle = \langle E_3(v_2||h_2, r_{3,2}), E_3(v_1||h_1, r_{3,1}), E_3(v_3||h_3, r_{3,3})\rangle = \langle y_{3,1}, y_{3,2}, y_{3,3}\rangle$. y_3 is broadcast signed by A_2.

A_3 chooses the permutation $\pi_3 = \langle 3, 1, 2\rangle$ and applies it on the decrypted elements of y_3 obtaining: $y_4 = \langle D_3(y_{3,3}), D_3(y_{3,1}), D_3(y_{3,2})\rangle = \langle v_3||h_3, v_2||h_2, v_1||h_1\rangle = \langle y_{4,1}, y_{4,2}, y_{4,3}\rangle$.

A_3 now publishes the votes $y_4 = \langle v_3||h_3, v_2||h_2, v_1||h_1\rangle$. Each voter A_i looks in this vector and checks that his h_i appears associated with the submitted vote v_i, meaning that the voting went correctly.

The simplified Merritt election protocol shown here has one more weakness. The problem is that A_n could change a vote that she/he dislikes, and the only one observing it is the voter whose vote was changed. If the voter complains, then its anonymity is lost. The full version of the Merritt election protocol solves this problem

Remark 2 *As it can be noticed, Merritt's protocol is computationally expensive, mainly due to the length of the messages. Public key encryption with growing messages can be performed with symmetric key encryption, sending only the session key in the onion encryption:*

$$y_{1,i} = E_1(K_1)||E_{K_1}(r_{1,i}||E_2(K_2)||E_{K_2}(r_{2,i}||...E_n(K_n)||E_{K_n}(r_{n,i}||v_i||h_i)...))$$

Remark 3 *Instead of submitting the votes using onion encryption, A_k's vote could be submitted only as $y_{1,k} = E_n(v_k||h_k, r)$, where E_n is a $(\circ, \bullet)$-homomorphic encryption, for some operations $\circ$ and $\bullet$. Then, each participant (election center) A_i in the Merritt chain only needs to compute $y_{i+1} = \pi_i(\{y_{i,k} \bullet E_n(e_\circ, r_{i,k})\}_k)$. Here $e_\circ$ is the identity element for the operation $\circ$ and $r_{i,k}$ is a secure random number.*

After this operation the secrets are shuffled and nobody can reconstruct the permutation due to the randomization based on the homomorphic encryption. However, if A_1 colludes with A_n then the privacy of the voters is lost. Nevertheless, we will see in Section 4 how a similar idea is successful for secure function evaluation.

3.3 The Merritt Election Protocol

As mentioned before, one needs to make sure that the last person decrypting the votes getting out of a mix-net cannot modify the votes without being detected by everybody. The reason why A_n can tamper with the votes undetected in the simplified version is that the encryption scheme was probabilistic, meaning that nobody other then the sender can check that the published decrypted messages correspond to the actual ciphertexts (the randomization parameters being lost). Probabilistic encryption was needed to hide the shuffling permutation.

An immediate fix is to add an additional layer of encryption at the end, this time with an encryption scheme that is not probabilistic (i.e., is publicly reversible). This uses the public key E'_k of a volunteer A_k, where A_k may or may not be one of the election centers shuffling the votes so far. During the corresponding decryption it is useless to perform any shuffling, since such shuffling can be reconstructed by everybody. One should either use an encryption that is not probabilistic (RSA or Rabin), or an encryption where the randomization factor can be retrieved (as in the non-probabilistic Paillier's scheme [Pai99]), or one can simply send the randomization factor as an additional plaintext to enable verification.

$$y_{1,i} = E_1(E_2(...E_{n-1}(E_n(E'_k(v_i||h_i), r_{n,1}), r_{n-1,i})..., r_{2,i}), r_{1,i})$$

Everybody can reverse the last operation of A_k and can detect if A_k does not decrypt correctly the signed message $E'_k(v_i||h_i)$ that he gets from A_n.

However, it follows that a single additional encryption layer is not enough since the new volunteer A_k could collude with the last shuffler A_n to see the votes in advance and to have A_n tamper with them. To increase the security, one needs to ask more than one volunteer, having an additional layer of encryption for each of them. The maximum security is achieved if all voters are present once again in the new set of volunteers with non-probabilistic encryption layers. That would requires all of them to cooperate in order to tamper a known vote.

All volunteers in the newly added set of encryption layers have to broadcast the messages that they forward, enabling everybody to detect when a message is tampered with. To enable user to prove in court that a certain volunteer is guilty for tampering, the messages should be digitally signed. A volunteer proceeds to do his decryption round only if all participants acknowledge that the protocol went correctly so far. Let us now see the obtained protocol, the Merritt election protocol.

The n election centers, $A_1, ..., A_n$, are ordered in a chain (the Merritt chain). Each agent A_i distributes two public keys, one for a probabilistic encryption cryptosystem resulting in an encryption function $E_i(m, r)$, and a public key for a non-probabilistic encryption scheme (i.e., having a publicly reversible decryption) resulting in an encryption function $E'_i(m)$. A_i keeps corresponding private decryption functions D_i and D'_i. Also, each A_i possesses a scheme for digital signatures, given by the function $S_i(m)$ which returns the message together with its signature. We denote with U_i a function that takes a signed message, verifies and removes the signature, and returns the message. Each election center A_i that sends a vote v_i chooses a large random number h_i and random numbers $r_{1,i}, ..., r_{n,i}$, and computes:

$$y_{1,i} = E_1(E_2(...E_n(E'_1(E'_2(...E'_{n-1}(E'_n(v_i||h_i))...)), r_{n,i})..., r_{2,i}), r_{1,i})$$

The $y_{1,j}$ values gathered by A_1 in a vector $\{y_{1,j}\}_j$, each on a position given by their second index, j, are posted through the Merritt chain in order from A_1 to A_n. Each A_i chooses a secret random permutation $\pi_i : [1..n]\rightarrow[1..n]$. As in the simplified version, we also define $\pi_i(\{x_k\}_{k\in[1..n]}) \stackrel{\text{def}}{=} \{x_{\pi_i^{-1}(k)}\}_{k\in[1..n]}$. The composition of the secret permutations of the first t participants is denoted by $j^{(t)}$. After receiving $\{y_{i,j^{(i-1)}(k)}\}_{j^{(i-1)}(k)}$, A_i sends to A_{i+1} the shuffled vector $\{y_{i+1,j^{(i)}(k)}\}_{j^{(i)}(k)}=\pi_i(\{D_i(y_{i,j^{(i-1)}(k)})\}_{j^{i-1}(k)})$. A_n broadcasts $\{y_{n+1,j^{(n)}}\}_{j^{(n)}}$:

$$y_{n+1,j^{(n)}(k)} = E'_1(E'_2(...E'_{n-1}(E'_n(v_{j^{(n)}(k)}||h_{j^{(n)}(k)}))...))$$

At this point, values have been securely shuffled (as the relation $(j,j^{(n)})$ was lost). The shuffled value can be found immediately by an additional decryption round in the order $A_1\rightarrow A_2\rightarrow...\rightarrow A_n$. A_1 broadcasts $\{y'_{2,j^{(n)}(k)}\}_{j^{(n)}(k)}$, where $y'_{2,j^{(n)}(k)} = S_1(D'_1(y_{n+1,j^{(n)}(k)}))$. Then, each subsequent election center $A_i, 1<i<n$ computes $y'_{i+1,j^{(n)}(k)} = S_i(D'_i(U_{i-1}(y'_{i,j^{(n)}(k)})))$. A_n broadcasts the signed votes $\{v_j||h_j\} = S_n(D'_n(U_{n-1}(y'_{n,j^{(n)}})))$. Each A_j checks for the presence of its h_j.

Example 2 *Assume A_1, A_2, and A_3 want to vote on an issue using the simplified Merritt protocol.*
A_1 computes $y_{1,1} = E_1(E_2(E_3(E'_1(E'_2(E'_3(v_1||h_1))), r_{3,1}), r_{2,1}), r_{1,1})$.
A_2 submits to A_1 the vote $y_{1,2} = E_1(E_2(E_3(E'_1(E'_2(E'_3(v_2||h_2))), r_{3,2}), r_{2,2}), r_{1,2})$.
A_3 submits to A_1 the vote $y_{1,3} = E_1(E_2(E_3(E'_1(E'_2(E'_3(v_3||h_3))), r_{3,3}), r_{2,3}), r_{1,3})$.
A_1 builds the vector $y_1 = \langle y_{1,1}, y_{1,2}, y_{1,3}\rangle$, chooses the permutation $\pi_1 = \langle 2, 3, 1\rangle$ and applies it on the decrypted elements of y_1 obtaining: $y_2 = \langle D_1(y_{1,2}), D_1(y_{1,3}), D_1(y_{1,1})\rangle = \langle E_2(E_3(E'_1(E'_2(E'_3(v_2||h_2))), r_{3,2}), r_{2,2}), E_2(E_3(E'_1(E'_2(E'_3(v_3||h_3))), r_{3,3}), r_{2,3}), E_2(E_3(E'_1(E'_2(E'_3(v_1||h_1))), r_{3,1}), r_{2,1})\rangle = \langle y_{2,1}, y_{2,2}, y_{2,3}\rangle$. y_2 is sent to A_2.
A_2 chooses the permutation $\pi_2 = \langle 1, 3, 2\rangle$ and applies it on the decrypted elements of y_2 obtaining: $y_3 = \langle D_2(y_{2,1}), D_2(y_{2,3}), D_2(y_{2,2})\rangle = \langle E_3(E'_1(E'_2(E'_3(v_2||h_2))), r_{3,2}), E_3(E'_1(E'_2(E'_3(v_1||h_1))), r_{3,1}), E_3(E'_1(E'_2(E'_3(v_3||h_3))), r_{3,3})\rangle = \langle y_{3,1}, y_{3,2}, y_{3,3}\rangle$. y_3 is sent to A_3.
A_3 chooses the permutation $\pi_3 = \langle 3, 1, 2\rangle$ and applies it on the decrypted elements of y_3 obtaining: $y_4 = \langle D_3(y_{3,3}), D_3(y_{3,1}), D_3(y_{3,2})\rangle = \langle E'_1(E'_2(E'_3(v_3||h_3))), E'_1(E'_2(E'_3(v_2||h_2))), E'_1(E'_2(E'_3(v_1||h_1)))\rangle = \langle y_{4,1}, y_{4,2}, y_{4,3}\rangle$. A_3 broadcasts y_4.
A_1 computes and broadcasts $y'_2 = \langle S_1(D'_1(y_{4,1})), S_1(D'_1(y_{4,2})), S_1(D'_1(y_{4,3}))\rangle = \langle S_1(E'_2(E'_3(v_3||h_3))), S_1(E'_2(E'_3(v_2||h_2))), S_1(E'_2(E'_3(v_1||h_1)))\rangle = \langle y'_{2,1}, y'_{2,2}, y'_{2,3}\rangle$.
A_2 computes and broadcasts $y'_3 = \langle S_2(D'_2(U_1(y'_{2,1}))), S_2(D'_2(U_1(y'_{2,2}))), S_2(D'_2(U_1(y'_{2,3})))\rangle = \langle S_2(E'_3(v_3||h_3)), S_2(E'_3(v_2||h_2)), S_2(E'_3(v_1||h_1))\rangle = \langle y'_{3,1}, y'_{3,2}, y'_{3,3}\rangle$.
A_3 computes and broadcasts $y'_4 = \langle S_3(D'_3(U_2(y'_{3,1}))), S_3(D'_3(U_2(y'_{3,2}))), S_3(D'_3(U_2(y'_{3,3})))\rangle = \langle S_3(v_3||h_3), S_3(v_2||h_2), S_3(v_1||h_1)\rangle = \langle y'_{4,1}, y'_{4,2}, y'_{4,3}\rangle$.

Each voter A_i looks in this vector, verifies the signature of A_3 by computing $\langle U_3(y'_{4,1}), U_3(y'_{4,2}), U_3(y'_{4,3})\rangle = \langle v_3||h_3, v_2||h_2, v_1||h_1\rangle$, and checks that his h_i appears associated with the submitted vote v_i, meaning that the voting was performed correctly.

Remark 4 *As for the simplified Merritt protocol, onion mix-nets are very computational and memory expensive, mainly due to the length of the messages. Public key encryption with growing messages can be performed with symmetric key encryption, sending only the session key in the onion encryption:*

$$y_{1,i} = E_1(K_1)||E_{K_1}(r_{1,i}||...E_n(K_n)||E_{K_n}(r_{n,i}||E'_1(...E'_{n-1}(E'_n(v_i||h_i))...))...)$$

3.4 Copying Votes

A remaining problem for applying the Merritt election protocol to voting is that A_1 can learn A_k's vote by copying his message. Instead of submitting his own vote, A_1 will duplicate the message of A_k. If two identical votes have the same random number $h_x = h_y$, then A_i learns that A_k has casted the corresponding vote, v_x.

For blocking this possible attack, the participants have to stop the protocol if they detect that two of the messages that they process are identical. However, if the encryption used has $(\circ, \circ)$-homomorphism property for some operator $\circ$, then A_1 can hide the resemblance of the two messages using this homomorphism (combining the message with a message built for random number h_i and for vote $e_\circ$, i.e., the identity element for $\circ$). After the end of the protocol, A_i has to find two votes with the same value and random numbers h_x and h_y in the relation $h_x = h_y \circ h_i$. A_i learns that A_k's vote is v_x. The solution for blocking this attack is to avoid encryptions with $(\circ, \circ)$-homomorphism (like RSA, ElGamal, Rabin). Paillier encryption is robust to this attack.

4 Mix-nets for Shuffling Shared Secrets

A probabilistic function that is expensive to evaluate on secret shares is the computation of a secret random permutation of a set of secrets where the permutation is not known by any participant. For fairness and privacy one prefers the permutation to be chosen with a probability given by the uniform distribution over the set of possible permutations. Given an arithmetic structure F we denote $R = \log_{|F|}(n!)$. For n secrets, the total number of possible permutations is $n!$. To select such a permutation, a participant needs to specify values for R variables in F. For fairness and for hiding the total permutation, each participant must be able to specify such a permutation and the performed permutation has to be a composition of the permutations proposed by each participant.

Such a function for k secrets and n participants can be defined as $f : F^k \times F^{nR} \rightarrow F^k$ where $f(\vec{x}, \vec{r}) = \pi_{\vec{r}}(\vec{x})$. $\pi_{\vec{r}}$ is a permutation defined by $\vec{r}$ and there are several ways to define it such that each permutation has equal probability to be obtained. This function is an important step in securely solving combinatorial problems that can have several solutions. A particular definition of $\pi_{\vec{r}}$ based on an arithmetic circuit is shown in [Sil04c] and has a very high complexity of $O(k!k)$ multiplications of shared secrets. Even if that solution is information theoretically secure, its cost makes it prohibitive and a computationally secure approach becomes acceptable. Here we will see such an approach based on mix-nets.

It is possible to randomize the permutation of the secrets by letting participants to jointly generate the secret permutation. In order to destroy the visibility of the relations between the initial order on the secrets and the resulting secrets one can exploit random joint permutations that are not known to any participant, similar to the simplified Merritt election protocol. The attack of tampering with reconstructed secrets by the last participant in the Merritt chain (which prompted the extension to the second round of the full version of the Merritt election protocol) is not possible in this technique since the result consists of shared secrets and the last participant in the chain does not get their reconstruction. The attack of duplicating a secret or tampering with a secret using homomorphism (especially by the first participant in the Merritt chain) can be thwarted as shown later in Section 5.

4.1 $(+ \bmod \nu, [\times]^2)$-Homomorphic Public Encryption Schemes

An encryption scheme $(E_{P_K}(m), D_{S_K}(c))$ is $(\circ, \bullet)$-homomorphic if $E_{P_K}(m_1) \bullet E_{P_K}(m_2) = E_{P_K}(m_1 \circ m_2)$.

We will denote a vector a with k elements as $\{a_i\}_{i\in[1..k]}$, where a_i is the element at position i. Given some group $(G, \circ)$, let us denote by $[\circ]^k$ the operation defined as:

$[\circ]^k : G^k \times G^k \rightarrow G^k; \{a_i\}_{i\in[1..k]}[\circ]^k\{a_i\}_{i\in[1..k]} = \{a_i \circ a_i\}_{i\in[1..k]}$

Note that Paillier encryption scheme with the public modulus μ is $(+ \bmod \mu, \times \bmod \mu^2)$-homomorphic. This means that it cannot be used for summations in a predefined field (aka modulus some predefined prime number ν). This seems not to be a problem when $\nu < 2 * \mu$ since the summation will still be correct. However, this is a problem for the use of the technique in secure multiparty computations, since after adding secret shares, the result can reveal the input parameters (as long as the reduction mod ν cannot be done in the encrpted form).

Remark 5 *In general, given a public key encryption scheme (E(m),D(c)) accepting plaintexts m from a* $\mathbb{Z}_n$ *where n is a multiple of* ν*, one can create a* $(+ \bmod \nu, \bullet)$*-homomorphic encryption scheme* $(E^\nu(m, k), D^\nu(c))$ *defined as:*
$k \in \mathbb{Z}_{\frac{n}{\nu}}$*, a secure random parameter*
$E^\nu(m, k) = E(m + k\nu)$ *and*
$D^\nu(c) = D(c) \bmod \nu$*.*

Such a scheme is obtained for the $(+, [\times]^2)$-homomorphic version of ElGamal [DGS02] with secret key x and public keys p, g, h, a where p is a large prime, g is a generator or an element with a large order mod p, $h = h^x \bmod p$, and a is an element whose order n is a multiple of ν.

Our encryption is then:

$$E_{p,g,h,a}(m, r, k) = \langle a^{m+k\nu} h^r \bmod p, g^r \rangle$$

where $m \in \mathbb{Z}_\nu$ is the plaintext, $k \in \mathbb{Z}_{\frac{ord(a)}{\nu}}$ and $r \in \mathbb{Z}_{ord(g)}$ are random numbers.

Our decryption is:

$$D_{p,g,x,a}(\langle \alpha, \beta \rangle) = \mathrm{ind}_{a,p}(\frac{\alpha}{\beta^x} \bmod p) \bmod \nu$$

where $\langle \alpha, \beta \rangle$ is a ciphertext and $\mathrm{ind}_{a,p}(c)$ is the discrete logarithm of c in base a mod p

Correctness and security of this scheme are guaranteed since it is a case of ElGamal. For ease of computing the discrete logarithm at decryption, a should be chosen such that its order n is small. Also, p should be chosen such that $\nu|(p-1)$. Given a generator b of $\mathbb{Z}_p$, one can select a with order ν as $a = b^{\frac{ord(b)}{\nu}} \bmod p = b^{\frac{p-1}{\nu}} \bmod p$.

Remark 6 *Given an element b of $\mathbb{Z}_p$ with order n multiple of ν, $n = t\nu$, one can select a with order ν as* $a = b^{\frac{ord(b)}{\nu}} \bmod p = b^t \bmod p$.

Note that an element b such that $b^{\frac{p-1}{\nu}} \neq 1$ has an order multiple of ν.

Remark 7 *This same modification can be added similarly to the composite version of Diffie-Hellman [Mah05], or other elliptic curve versions of elGamal.*

4.2 Shuffling unidimensional secrets

Now we address the problem n participants $A_1, ..., A_n$ shuffling a vector $S = \langle s_1, ..., s_k \rangle$ of k shared secrets from $\mathbb{Z}_\nu$, using a mix-net. Each participant A_i chooses a random secret permutation π_i, picked with a uniform distribution over the set of possible permutations: $\pi_i : [1..k] \rightarrow [1..k]$. The shares of A_i for the secrets in S are the vector $S_i = \langle s_{1,i}, ..., s_{k,i} \rangle$, $s_{j,i} \in \mathbb{Z}_\nu$.

Each participant A_i chooses a pair of keys for a $(+ \bmod \nu, \bullet)$-homomorphic public encryption scheme (for some operation $\bullet$), and publishes the public key (the obtained encryption function being denoted $E_i(m)$).[1] A_i encrypts with her own public key her shares in S_i obtaining a vector $ES_i = \langle E_i(s_{1,i}), ..., E_i(s_{k,i}) \rangle$. The serialized encrypted vectors are then sent to A_1. A_1 shuffles the serialized vectors according to her permutation π_1, then passes the result to A_2 which applies π_2, etc., until the agent A_n which applies π_n. A_n sends each vector to the agent that originated it.

To avoid that agents get a chance to learn the final permutation by matching final shares with the ones that they encrypted, a randomization step is also applied at each shuffling. Each participant A_j applies a randomization step on the set of shares for each element of S, by adding corresponding shares of zero [BOGW88]. Since operands are encrypted, to be able to perform this summation one exploits the $(+ \bmod \nu, \bullet)$-homomorphic properties of E_i. A_j computes for each secret s_k the Shamir shares of a zero, $z_{j,k,1}, ..., z_{j,k,n}$. Then, A_j sets $E_i(z_{j,k,i}) \bullet E_i(s_{k,i})$ as the new value of $E_i(s_{k,i})$, randomizing the sharing of s_k.

Example 3 *Participants A_1, A_2, and A_3 share two secrets from $\mathbb{Z}_3$, $s_1 = 0$ and $s_2 = 1$. s_1 is shared as $\langle 1, 2, 0 \rangle$ and s_2 as $\langle 0, 2, 1 \rangle$, using Shamir's (2,3)-threshold scheme. A_i holds the i^{th} shares of each secret. For simplicity in this example we do not use our cryptosystem but Paillier's, and each A_i uses the Paillier key for n=15 (note that this is not secure).*
A_1 (shares $\langle 1, 0 \rangle$) computes $v_1 = \langle E_1(1, 4), E_1(0, 7) \rangle = \langle 34, 118 \rangle$.
A_2 (shares $\langle 2, 2 \rangle$) submits $v_2 = \langle E_2(2, 2), E_1(2, 13) \rangle = \langle 158, 67 \rangle$ to A_1.
A_3 (shares $\langle 0, 1 \rangle$) submits $v_3 = \langle E_3(0, 11), E_1(1, 4) \rangle = \langle 26, 34 \rangle$ to A_1.
A_1 generates two sharing of 0, $z_{1,0} = \langle 0, 0, 0 \rangle$ and $z_{1,1} = \langle 1, 2, 0 \rangle$, and a random permutation $\pi_1 = \langle 0, 1 \rangle$, and computes:
$v_1 = \pi_1 \langle 34 * E_1(0, 7), 118 * E_1(1, 2) \rangle = \langle 34 * 118, 118 * 38 \rangle = \langle 187, 209 \rangle$,
$v_2 = \pi_1 \langle 158 * E_2(0, 11), 67 * E_2(2, 4) \rangle = \langle 158 * 26, 67 * 94 \rangle = \langle 58, 223 \rangle$,
$v_3 = \pi_1 \langle 26 * E_3(0, 2), 34 * E_3(0, 4) \rangle = \langle 26 * 143, 34 * 199 \rangle = \langle 118, 16 \rangle$,
and sends them to A_2.
A_2 generates two sharing of 0, $z_{1,0} = \langle 2, 1, 0 \rangle$ and $z_{1,1} = \langle 1, 2, 0 \rangle$, and a random permutation $\pi_2 = \langle 1, 0 \rangle$, and computes:
$v_1 = \pi_2 \langle 187 * E_1(2, 7), 209 * E_1(1, 1) \rangle = \pi_2 \langle 187 * 58, 209 * 16 \rangle = \langle 194, 46 \rangle$.
$v_2 = \pi_2 \langle 58 * E_2(1, 4), 223 * E_2(2, 7) \rangle = \pi_2 \langle 58 * 34, 223 * 58 \rangle = \langle 109, 172 \rangle$.
$v_3 = \pi_2 \langle 118 * E_3(0, 2), 16 * E_3(0, 8) \rangle = \pi_2 \langle 118 * 143, 16 * 107 \rangle = \langle 137, 224 \rangle$. *and sends them to A_3.*
A_3 generates two sharing of 0, $z_{1,0} = \langle 1, 2, 0 \rangle$ and $z_{1,1} = \langle 2, 1, 0 \rangle$, and a random permutation $\pi_3 = \langle 1, 0 \rangle$, and computes:
$v_1 = \pi_3 \langle 194 * E_1(1, 11), 46 * E_1(2, 13) \rangle = \langle 46 * 67, 194 * 191 \rangle = \langle 157, 154 \rangle$.
$v_2 = \pi_3 \langle 109 * E_2(2, 1), 172 * E_2(1, 14) \rangle = \langle 172 * 209, 109 * 31 \rangle = \langle 173, 4 \rangle$.
$v_3 = \pi_3 \langle 137 * E_3(0, 7), 224 * E_3(0, 2) \rangle = \langle 224 * 143, 137 * 118 \rangle = \langle 82, 191 \rangle$.
It sends v_1 to A_1, v_2 to A_2, and decrypts v_3.
A_1 gets by decryption shares $\langle 5, 3 \rangle = \langle 2, 0 \rangle$.
A_2 gets by decryption shares $\langle 4, 8 \rangle = \langle 1, 2 \rangle$.

[1] Note that the use of the original $(+ \bmod \mu, \times \bmod \mu^2)$-homomorphic Paillier scheme requires $\mu > (n+1)\nu$, and is not fully secure. The use of $(+ \bmod \nu, \bullet)$-homomorphic public encryption schemes as the one we propose above is recommended.

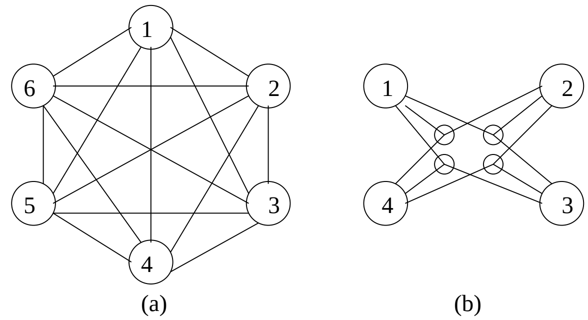

Figure 4: Graphs where permutations are homomorphic: (a) a fully connected binary graph; (b) a fully connected hyper-graph with ternary edges.

A_3 gets by decryption shares $\langle 0, 1\rangle$.
This is a shuffled re-sharing of the initial secrets (it happens that the permutations canceled each other and the secrets are in their initial positions).

4.3 Shuffling multidimensional vectors

Previously we have seen how a random hidden reordering of a unidimensional vector can be obtained using a version of the Merritt election protocol. Now we will show how one can shuffle a multidimensional space containing multidimensional vectors of secrets, performing secret permutations on all dimensions. The vectors may share some dimensions and the whole space has m dimensions, $\{1, ..., m\}$. The possible coordinates for the dimension i are $\{1, 2, ..., d_i\}$. Additionally one can also shuffle the dimensions themselves.

Remark 8 *This problem has application to the shuffling of descriptions secret for combinatorial problems (satisfiability (SAT) problems, or constraint satisfaction (CSP) problems) where the dimensions correspond to variables, the set of possible coordinates to the domains of these variables, and the multi-dimensional vectors to the representation of secret predicates/constraints [Sil03].*

Remark 9 *Note that the dimensions can be shuffled securely only if a) all dimensions have the same number of coordinates; and b) the hyper-graph induced by the vectors on any permutation of the dimensions is homomorphic to the initial graph. This holds for fully connected graphs of any arity (see Figure 4).*

As before, each participant A_i chooses a pair of keys for a $(+, \bullet)$-homomorphic public encryption scheme with plaintexts from $\mathbb{Z}_\mu$, and publishes the public key (the obtained encryption function being denoted $E_i(m)$).

Assume the participants share secrets organized in k multidimensional vectors $\{V_i\}_i \in [1..k]$. Each V_i having t_i dimensions, $\langle i_1, ..., i_{t_i}\rangle$, and is denoted $\{s^i_{v_{i_1},...,v_{i_{t_i}}}\}_{\forall u, v_{i_u} \in [1..d_{i_u}]}$. The secrets and their shares are in $\mathbb{Z}_\nu$. Participant A_j has for V_i the set of secret shares $\{s^{i,j}_{v_{i_1},...,v_{i_{t_i}}}\}_{\forall u, v_{i_u} \in [1..d_{i_u}]}$. Each participant A_j encrypts its secret shares for all elements of each vector V_i with its own public key E_j, and submits the obtained vectors, $EV_i = \{E_j(s^{i,j}_{v_{i_1},...,v_{i_{t_i}}})\}_{\forall u, v_{i_u} \in [1..d_{i_u}]}$, to a mix-net.

All the submissions are made to A_1, the first participant in the Merritt chain. For each vector V_i, each participant A_k generates a set of n vectors of the same size containing at each element sharing of zero, $\{Z_{k,i,j}\}_{j \in [1..n]}$. $Z_{k,i,j}$ has the form $\{z^{k,i,j}_{v_{i_1},...,v_{i_{t_i}}}\}_{\forall u, v_{i_u} \in [1..d_{i_u}]}$. Note that $\langle z^{k,i,0}_{v_{i_1},...,v_{i_{t_i}}}, ..., z^{k,i,n}_{v_{i_1},...,v_{i_{t_i}}}\rangle$ is a sharing of zero. A_k also generates the secret permutations:

$$\begin{aligned} \pi^k : \quad & [1..m] \to [1..m], && \text{(for variables)} \\ l \in [1..m], \pi^k_l : \quad & [1..d_l] \to [1..d_l], && \text{(for domains)} \end{aligned}$$

When A_1 or a subsequent A_k receives (all the elements of) all the vectors with shares for a participant A_j, it transforms each $\{E_j(s^{i,j}_{v_{i_1},...,v_{i_{t_i}}})\}_{\forall u, v_{i_u} \in [1..d_{i_u}]}$ into $\{E_j(z^{k,i,j}_{v_{i_1},...,v_{i_{t_i}}}) \bullet E_j(s^{i,j}_{\pi^k_{i_1}(v_{i_1}),...,\pi^k_{i_{t_i}}(v_{i_{t_i}})})\}_{\forall u, v_{i_u} \in [1..d_{i_u}]}$ by applying the permutations π^k_i on each corresponding dimension i.

Note that if a permutation π^k is applied on dimensions (in this case all dimensions have size d), the resulting vectors are: $\{E_j(z^{k,i,j}_{v_{i_1},\ldots,v_{i_{t_i}}}) \bullet E_j(s^{i,j}_{\pi^k(\pi^k_{i_1}(v_{i_1}),\ldots,\pi^k_{i_{t_i}}(v_{i_{t_i}}))})\}_{\forall u, v_{i_u} \in [1..d]}$. If permutations are applied on dimensions, then the order of the coordinates in vector V_i has to be shuffled consistently over the different shares for the same vector (e.g., ordered by the new IDs of the dimensions). If there are several vectors, they are shuffled with the same permutations. The order on vectors should also be shuffled in this case.

As with the shuffling of unidimensional secrets, the homomorphism of the encryption scheme ensures that each set of shares is randomized by addition with a 0.

Example 4 *Participants A_1, A_2, and A_3 share three vectors of secrets from $\mathbb{Z}_3$ in a 2-dimensional space (x,y), $s^x_1 = \langle 0,1\rangle$, $s^y_2 = \langle 2,0\rangle$, and $s^{x,y}_3 = \langle\langle 1,2\rangle,\langle 2,0\rangle\rangle$. The two-dimensional vector $s_{x,y}$ shares the first dimension with s_x and the second with s_y. All participants submit their encrypted shares to A_1.*
A_1 generates permutations $\pi^1 = \langle 1,0\rangle$ (creating new dimensions x' and y'), and $\pi^1_1 = \langle 0,1\rangle, \pi^1_2 = \langle 1,0\rangle$, and by shuffling and randomizing all shares of the vectors gets shares for
$s^{y'}_1 = \pi^1_1(s^x_1) = \pi^1_1\langle 0,1\rangle = \langle 0,1\rangle$,
$s^{x'}_2 = \pi^1_2(s^y_2) = \pi^1_2\langle 2,0\rangle = \langle 0,2\rangle$, *and*
$s^{y',x'}_3 = \pi^1_1\langle \pi^1_2\langle 1,2\rangle, \pi^1_2\langle 2,0\rangle\rangle = \langle\langle 2,1\rangle,\langle 0,2\rangle\rangle$. *These are sent shuffled to A_2.*
A_2 generates permutations $\pi^2 = \langle 0,1\rangle, \pi^2_1 = \langle 1,0\rangle, \pi^2_2 = \langle 0,1\rangle$, and by shuffling and randomizing all shares of the vectors gets shares for
$s^{y''}_1 = \pi^2_2(s^{y'}_1) = \pi^2_2\langle 0,1\rangle = \langle 0,1\rangle$,
$s^{x''}_2 = \pi^2_1(s^{x'}_2) = \pi^2_1\langle 0,2\rangle = \langle 2,0\rangle$, *and*
$s^{y'',x''}_3 = \pi^2_2\langle \pi^2_1\langle 2,1\rangle, \pi^2_1\langle 0,2\rangle\rangle = \langle\langle 1,2\rangle,\langle 2,0\rangle\rangle$. *These are sent to A_3.*
A_3 generates permutations $\pi^3 = \langle 1,0\rangle, \pi^3_1 = \langle 1,0\rangle, \pi^3_2 = \langle 0,1\rangle$, and by shuffling and randomizing all shares of the vectors gets shares for
$s^{x'''}_1 = \pi^3_2(s^{y''}_1) = \pi^3_2\langle 0,1\rangle = \langle 0,1\rangle$,
$s^{y'''}_2 = \pi^3_1(s^{x''}_2) = \pi^3_1\langle 2,0\rangle = \langle 0,2\rangle$, *and*
$s^{x''',y'''}_3 = \pi^3_2\langle \pi^3_1\langle 1,2\rangle, \pi^3_1\langle 2,0\rangle\rangle = \langle\langle 2,1\rangle,\langle 0,2\rangle\rangle$.
In this case the permutations of dimensions canceled each other and the final shuffling consists only in a permutation of the second dimension, y.

4.4 Generating permutations uniformly distributed

Each participant in the mix-net for shuffling shares has to generate random permutations uniformly distributed over all possible permutations. The total number of possible permutations for a vector with k values is $k!$. A way to generate such a permutation is to generate randomly a number in the interval $[0..k!-1]$ and from this number to reconstruct the corresponding permutation. The drawback of Algorithm 1 is the need to work with very large numbers. Another solution that we propose here is to generate the permutation by construction using a simpler method. We will then show that this method generates permutations with a uniform distribution. The method is as follows:

Algorithm 1: Generating a permutation of k values from a number x.

```
function PermNum(k, x)
    for (i = 1; i ≤ k; i++) do
        m = k − i!;
        r[i] = ⌊x/m⌋ + 1;
        x = x mod m;
```

Theorem 1 *Algorithm 2 generates permutations that are uniformly distributed over the set of possible permutations.*

Proof. The probability for a value x to be placed on the first position in the result vector is equal to $1/k$.

Algorithm 2: Generating a random permutation

```
function GenPerm(k)
    list l = [1..k];
    for (i = 1; i ≤ k; i++) do
        x=random([1..k − i + 1]);
        r[i] = l[x];
        delete l[x] from list l;
```

Its probability to be placed on the second position is the probability of not being selected for the first one, $(k-1)/k$, multiplied with the probability of it being selected for the second $1/(k-1)$. Overall, the probability is $\frac{k-1}{k}\frac{1}{k-1} = \frac{1}{k}$.

By mathematical induction, if the probability of a value x being assigned to any of the first i positions is $1/k$, then the probability of it being assigned to the $(i+1)^{th}$ position is the probability of not being on the first i positions, $(k-i)/k$, multiplied to the probability of it being chosen in the $(i+1)^{th}$ cycle, $1/(k-i)$. I.e. $\frac{k-i}{k}\frac{1}{k-i} = \frac{1}{k}$.

This shows that the probability of any permutation being chosen is equal, and therefore the distribution is uniform. *Q.E.D.*

5 ZK IP for Shared Secret Shuffling

The attack of duplicating a secret or tampering with a secret using homomorphism (by a participant in the Merritt chain) can be addressed with an interactive zero-knowledge proof. It is inspired from the proof of knowledge of an isomorphism between two large graphs [GMR85, Gol04, GMW86, GMW87, Blu86].

Note that other efficient ZK proofs (e.g., non interactive zero-knowledge proofs) can be derived from the ones below, similarly to how the techniques used for shuffling encrypted secrets are built [Nef01, Gro05].

5.1 ZK proof for knowledge of isomorphism between large graphs

Peggy needs to prove to Victor that she knows the isomorphism f between two large graphs G_1 and G_2 (e.g., to get access to some resource).

1. Peggy generates a random isomorphism f_1 and computes a new graph $G = f_1(G_1)$. Peggy send G to Victor claiming that G is isomorphic to both G_1 and G_2.

2. Victor generates a random challenge $c \in \{0, 1\}$ and sends it to Peggy. If c is 0 then Victor expects in return the isomorphism between G_1 and G. Otherwise he asks for the isomorphism between G_2 and G.

3. if c is 0 then Peggy sends f_1 to Victor. Otherwise she sends $f^{-1} \circ f_1$.

With each iteration of this interactive proof, Victor's doubts about Peggy knowing f decrease with 50%.

5.2 ZK proof for shuffling secret shares

A participant A_j in the Merritt chain for shuffling n vectors of k shared secrets $v = \{\{v_i^1\}_{i\in[1..k]}, ..., \{v_i^n\}_{i\in[1..k]}\}$ (see Figure 5) can prove the correctness of his/her shuffling $u = \{\{u_i^1\}_{i\in[1..k]}, ..., \{u_i^n\}_{i\in[1..k]}\}$ (obtained with a permutation π and summation with a vector of shares of zero, $\{\{z_i^1\}_{i\in[1..k]}, ..., \{z_i^n\}_{i\in[1..k]}\}$). by an interactive proof. Each step t of the interactive proof consists of:

- A_j generates a claim, i.e., an additional shuffling $y = \{\{y_i^1\}_{i\in[1..k]}, ..., \{y_i^n\}_{i\in[1..k]}\}$ (obtained with a permutation π_t and summation with a vector of shared zeros, $\{\{z_{t,i}^1\}_{i\in[1..k]}, ..., \{z_{t,i}^n\}_{i\in[1..k]}\}$), and sends it to the verifier.

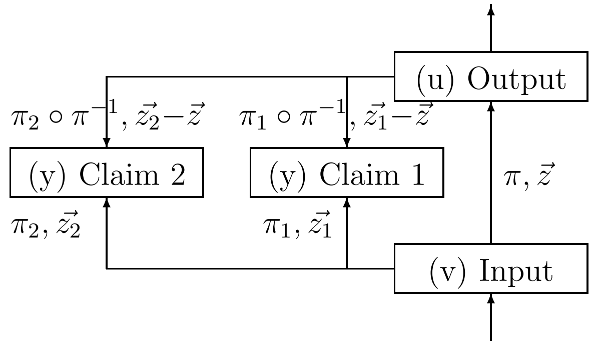

Figure 5: A participant's Zero Knowledge proofs for shuffling shared secrets. Two claims are generated, allowing a 75% certitude of the trust in the shuffler.

- The verifier sends a challenge c, $c \in \{0, 1\}$, to A_j.
- Upon the reception of the challenge c:
  - If $c = 0$ then A_j reveals the permutation and the set of k shared zeros that translate from u to y (namely, permutation $\pi_t \circ \pi^{-1}$ and the set of shared zeros $\{\{z^1_{t,i} - z^1_i\}_{i\in[1..k]}, ..., \{z^n_{t,i} - z^n_i\}_{i\in[1..k]}\}$).
  - If $c = 1$ then A_j reveals a permutation and the set of k shared zeros that translate from v to y (namely, permutation π_t and the set of shared zeros $\{\{z^1_{t,i}\}_{i\in[1..k]}, ..., \{z^n_{t,i}\}_{i\in[1..k]}\}$).

Each challenge successfully answered reduces the suspicions about A_j with 50%. t such steps can be performed simultaneously (to reduce suspicions $\frac{1}{2^t}$ times at once).

To allow all other participants to be simultaneously convinced by a single answer by A_j to t challenges, the challenges can be constructed using a protocol for tossing a coin over the phone. For example:

- Each of the verifiers A_j, $j \neq i$, commits to t guesses $c_{1,j}, ..., c_{t,j}$, where $c_{k,j} = 0$ if A_j guesses the first form and $c_{k,j} = 1$ otherwise. Each verifier broadcasts his commitments.
- Everybody opens his commitments.
- Everybody computes the set of joint challenges, $c_t = \oplus^n_{x=1,x\neq i}(c_{t,x})$.

A commitment to a bit b can be achieved as the ciphertext obtained by encrypting with AES a message whose last bit is b, using a randomly obtained key. The commitment is opened by revealing the key and the message yielding the ciphertext of the commitment.

Each of the shufflers that answers all t simultaneous challenges to t simultaneously published claims can be trusted (with probability $1 - 1/2^t$). All n participants in the Merritt chain can play the IP simultaneously for proving their corresponding operation to minimize message overhead. The memory overhead is $k(t+1)n(n-1)$ shares of secrets from $n-1$ provers for t claims (+1 actual output), each with n shares for k secrets.

All the participants in the Merritt chain have to prove their operation to avoid any tampering with the secrets.

Remark 10 *If one only plans to protect the shares from a set of colluders knowing the identity of the secrets, A_1, then the concept of a (t, n)-threshold scheme would require the first $n-t$ participants to provide the proof (since $n - t + 1$ colluding participants can always modify any shared secret).*

5.3 ZK proof for inversion of a shuffling

A related problem is the un-shuffling. Assume that a set of participants have shuffled a vector of k shared secrets at some prior time. Unshuffling is the problem where another vector of k shared secrets has to be un-shuffled (using the inverse permutations). To our knowledge, this has not been addressed for the similar problem of reversing the shuffling of encrypted secrets [Nef01, Gro05], and our solution proposed

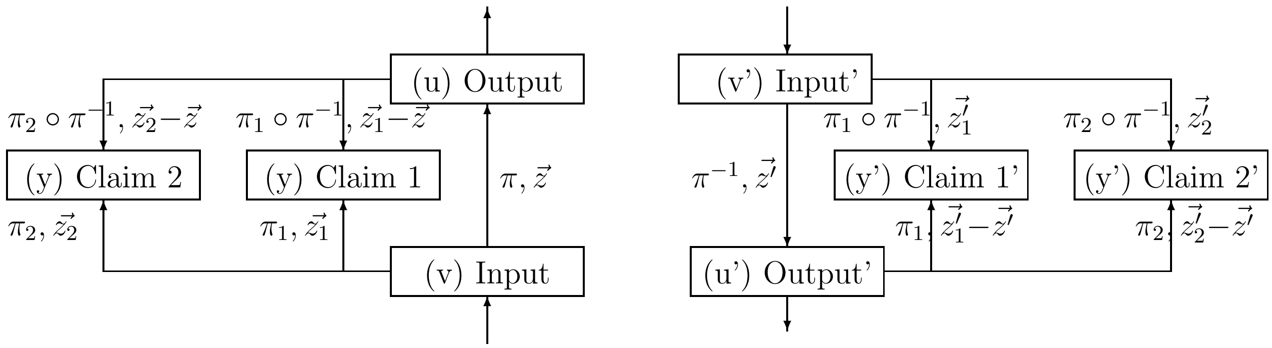

Figure 6: Zero Knowledge proofs for performing an inverse shuffling. Shuffling is shown on the left and unshuffling on the right. Two claims are generated, allowing a 75% certitude of the trust in the shuffler.

in the following can be quite straightforwardly be translated to that setting (and enriched with features from [Nef01, Gro05]).

This problem occurs when one desires to perform different operations an some shared secrets wihout knowing the identity of the secrets. Typically one shuffles these secrets with hidden permutations, performs the operations on the shuffled secrets with a result potentially represented as a vector, vector that has to be unshuffled with the reverse permutation. An example of application is the random selection: setting to 0 all but one randomly picked (non-zero) secret.

Unshuffling is done by a procedure similar with the one for shuffling, just that each participant has to use the inverse of the permutation employed there. Malicious participants in the mix-net could try to use a different permutation to swap the values of some secrets. We now show how to construct a zero-knowledge proof allowing mix-net members to prove the correctness of their actions.

We propose a technique that proves simultaneously the shuffling and the unshuffling. We denote by $\{v_i^t\}_{i\in[1..k]}$ the set of shares generated for A_t, for the secrets v_1 through v_k. A participant A_j in the Merritt chain for shuffling n vectors of k shared secrets $v = \{\{v_i^1\}_{i\in[1..k]}, ..., \{v_i^n\}_{i\in[1..k]}\}$ (Input in Figure 6) and unshuffling n vectors of k shared secrets $v' = \{\{v_i'^1\}_{i\in[1..k]}, ..., \{v_i'^n\}_{i\in[1..k]}\}$ (Input' in Figure 6), can prove the correctness of his/her

- shuffling $u = \{\{u_i^1\}_{i\in[1..k]}, ..., \{u_i^n\}_{i\in[1..k]}\}$ (Output in Figure 6), obtained with a permutation π and summation with a vector of shares of zero, $\{\{z_i^1\}_{i\in[1..k]}, ..., \{z_i^n\}_{i\in[1..k]}\}$,
- and unshuffling $u' = \{\{u_i'^1\}_{i\in[1..k]}, ..., \{u_i'^n\}_{i\in[1..k]}\}$ (Output' in Figure 6), obtained with a permutation π^{-1} and summation with a vector of shares of zero, $\{\{z_i'^1\}_{i\in[1..k]}, ..., \{z_i'^n\}_{i\in[1..k]}\}$.

Each step t of the interactive proof consists of:

- A_j generates a claim, i.e.,
  - an additional shuffling $y = \{\{y_i^1\}_{i\in[1..k]}, ..., \{y_i^n\}_{i\in[1..k]}\}$ (Claim t in Figure 6), obtained with a permutation π_t and summation with a vector of shared zeros, $\{\{z_{t,i}^1\}_{i\in[1..k]}, ..., \{z_{t,i}^n\}_{i\in[1..k]}\}$,
  - and an additional unshuffling $y' = \{\{y_i'^1\}_{i\in[1..k]}, ..., \{y_i'^n\}_{i\in[1..k]}\}$ (Claim t' in Figure 6), obtained with permutation $\pi_t \circ \pi^{-1}$ and summation with a vector of shared zeros, $\{\{z_{t,i}'^1\}_{i\in[1..k]}, ..., \{z_{t,i}'^n\}_{i\in[1..k]}\}$, and sends them to the verifier.
- The verifier sends a challenge c, $c \in \{0, 1\}$, to A_j.
- Upon the reception of the challenge c:
  - If $c = 0$ then A_j reveals the permutation and the two sets of k shared zeros that translate from u to y and from v' to y' (namely, permutation $\pi_t \circ \pi^{-1}$ and the two sets of shared zeros $\{\{z_{t,i}^1 - z_i^1\}_{i\in[1..k]}, ..., \{z_{t,i}^n - z_i^n\}_{i\in[1..k]}\}$ respectively $\{\{z_{t,i}'^1\}_{i\in[1..k]}, ..., \{z_{t,i}'^n\}_{i\in[1..k]}\}$).

– If $c = 1$ then A_j reveals a permutation and the two sets of k shared zeros that translate from v to y and from u' to y' (namely, permutation π_t^{-1} and the two sets of shared zeros $\{\{z^1_{t,i}\}_{i\in[1..k]}, ..., \{z^n_{t,i}\}_{i\in[1..k]}\}$ respectively $\{\{z'^1_{t,i} - z'^1_i\}_{i\in[1..k]}, ..., \{z'^n_{t,i} - z'^n_i\}_{i\in[1..k]}\}$).

Each challenge successfully answered reduces the suspicions about A_j with 50%. t such steps can be performed simultaneously (to reduce suspicions $\frac{1}{2^t}$ times at once). Common verification by several participants can be done as in the previous section.

5.4 Faster arithmetic circuit for shuffling vectors

An arithmetic circuit for shuffling a vector of k shared secrets was proposed in [Sil04c] and was inspired from Algorithm 1, requiring $O(k!k)$ multiplications. Now we propose a better arithmetic circuit based on the Algorithm 2. Let $\mathbb{Z}_q$ be the field of the arithmetic circuit. We will use the secure primitives:

Equality test: $\delta(x)$ is a function returning a shared 1 when $x = 0$ and returning a shared 0 otherwise. A secure implementation with a constant number of rounds is proposed in [Kil05].

Random: $Random(i)$ is a function returning a vector r_i of i shared secrets where exactly one has value 1 and all the other secrets have value 0 [Sil04c]. The index of the secret equal to 1 is generated according to a uniform distribution over the interval $[1..i]$. A random number r uniformly distributed in $[0..q]$ can be generated in one round by having each participant A_j share a random number $r[j]$ and compute $r = \sum_{j=1}^{n} r[j]$. One can test that $r < i$ by using the secure comparison in [Kil05]. If the test fails, a new random number can be tried. Otherwise the index with value 1 is chosen as $r+1$, by computing $r_i[k] = \delta(r + 1 - k)$. The expected number of trials is $\frac{q}{i}$.

The Algorithm 3 performs the random permutation of a vector with k secrets. To generate k random numbers of size increasing from 2 to k, the total expected number of trials is $\sum_{i=2}^{k} \frac{q}{i}$ which is in $O(q \log k)$. All random numbers can be generated in parallel, and several trials for each random number can be also performed in parallel, leading to an expected number of rounds denoted $\mathcal{R}_q(i)$.

Algorithm 3: Permuting randomly a vector v with k shared secrets

function $(v[1..k])$
  for $i = 1$ *to* $k{-}1$ **do** $r_i[1..k{-}i{+}1] \leftarrow Random(k{-}i{+}1)$;
1 **for** $i = 1$ *to* $k{-}1$ **do**
    $v'[i] = \sum_{j=i}^{k}(r_i[j - i + 1] * v[j])$;
    for $j = i + 1$ *to* k **do**
      $v'[j] = v[j] + r_i[j - i + 1] * (v[i] - v[j])$;
    $v[i..k] \leftarrow v'[i..k]$;
  return $v[1..k]$;

A loop started at Line 1 in Algorithm 3 permutes the value at position i with the one at position $i + k_i - 1$, where k_i is the position of the 1 element in vector r_i. All the multiplications in such a loop can be performed in parallel, so that the whole computation (after generating the random numbers) can be performed in k rounds. The total number of multiplications in round i is $2(k - i) + 1$, leading to a total number of $k^2 - k + 1$ multiplications.

The remaining issue is how to perform on a vector the inverse permutation with respect to the shuffling in Algorithm 3. A method for performing the inverse permutation is given in Algorithm 4, and simply performs the permutations in the reverse order starting with the last round.

6 Applications

We show how the technique will be used in a few applications and exemplify the simplest ones with the corresponding programs in the SMC programming language.

Algorithm 4: Inverting the permutation on vector v in Algorithm 3

function $(v[1..k],\ r_1[1..k],\ldots,r_{k-1}[1..2])$
  for $i = k{-}1$ *to* 1 **do**
    $v'[i] = \sum_{j=i}^{k}(r_i[j-i+1] * v[j])$;
    for $j = i+1$ *to* k **do**
      $v'[j] = v[j] + r_i[j-i+1] * (v[i] - v[j])$;
    $v[i..k] \leftarrow v'[i..k]$;
  return $v[1..k]$;

6.1 Shuffling secrets with the SMC language

To express the problem of shuffling three secrets from three participants and revealing them in a random order generated with fairness, the SMC program (version 1.4.2) is:

```
MODULUS      31
PARTICIPANTS 3

INPUTS
1 # nb. of inputs from the 1-st participant
# variable, vector-size, label, type
s1 I secret Integer
1 # nb. of inputs from the 2-nd participant
s2 I secret Integer
1
s3 I secret Integer

INTERMEDIARY-INPUTS 5
# main is the entry point for this program
# main calls _r to create the array, and then calls the mix-net
main=SEQUENCE(_r(0),SHUFFLE(_r,3))
# _r creates the array _r and returns the x-th value
_r(x)=ARRAY(_r,x,s1,s2,s3)
# define the outputs
o0=_r(0)
o1=_r(1)
o2=_r(2)

OUTPUTS
3 # nb. of outputs for the 1-st participant
# label revealed-value
result1 o0
result2 o1
result3 o2
3 # nb. of outputs for the 2-nd participant
result1 o0
result2 o1
result3 o2
3 # nb. of outputs for the 3-rd participant
result1 o0
result2 o1
result3 o2
```

To reveal only a randomly selected value out of the three secrets in the previous program, one only needs to change the OUTPUTS section to:

```
OUTPUTS
1 # nb. of outputs for the 1-st participant
result o0
1 # nb. of outputs for the 2-nd participant
result o0
1 # nb. of outputs for the 3-rd participant
result o0
```

The shuffling of multidimensional vectors of secrets is implemented in SMC's shuffling for CSPs, and is called with SHUFFLE(CSP). In SMC each participant saves the last permutations used for shuffling and unshuffling can be dune with a call to UNSHUFFLE(vector-name,vector-length) and UNSHUFFLE(CSP), respectively.

The efficiency of the mix-net for secret shares depends on the key size, the number of participants and the number of secrets. With 1024 bit keys, the mix-net for secret shares in the version 1.4.2 of SMC requires approximatively 1 second per secret per participant (without ZK proofs). In consequence, ZK proofs with certitude $1-\frac{1}{2^t}$ are expected to need $2t$ seconds per secret (t seconds for generating claims, and t for verifying them, the cost of jointly building the challenges being negligible).

6.2 Selecting a random element out of a secret vector

One can use mix-nets of shared secrets for finding a random solution for a combinatorial problem. A simple method (called *Shuffle&Select*) is to (1) first compute a secret vector v with the properties of each alternative (i.e., potential solution) of the combinatorial problem[2]. (2) The vector v is shuffled with a mix-net. (3) The first element with acceptable properties in the shuffled v is chosen using the boolean circuit shown in Algorithm 5, *CF*. The *CF* algorithm returns a vector r containing shared secrets equal to 0 on all positions except on the position of the first element of v that has the desired property g. (4) If identifiers of the alternatives are among the properties stored in v then one can find the selected alternative by opening the elements of r and then opening the identity of the only non-zero element of r.

An alternative method (called *Shuffle&Select&Unshuffle*) differs only starting with the fourth step. (4') r can be unshuffled with the UNSHUFFLE operator. In this case, opening r will implicitely reveal the identity of the selected alternative (from the way v was built). Moreover, in this case one can compute a shared secret containing the position of the non-zero element of r by using the boolean circuit *V2S* shown in Algorithm 6.

The boolean circuit *CF* selects the first element with a certain property g out of a vector v with k secret shares (we described a solution based on arithmetic circuits in [Sil05]). The idea is to build a result vector r storing a shared 1 in the position holding the selected value, and shared 0s in all other positions. An intermediary vector of secrets h is used storing in each position the secret 1 if the selection was not yet done (up to that index in v) and 0 otherwise. The vector r can be transformed in a shared secret.

Algorithm 5: Choose First: Select the first element of v satisfying g. Given the boolean circuit $g(x)$ returning 1 if x has property g and 0 otherwise, *CF* builds the vector r.

function *CF*(v, g)
  $h[1] = 1$, $r[1] = g(v[1])$;
  for $(i = 2; i \leq k; i{+}{+})$ **do**
    $h[i] = h[i{-}1] \wedge (1{-}r[i{-}1])$;
    $r[i] = g(v[i]) \wedge h[i]$;
  return r;

Another way of computing CF is with the functions $r[i] = g(r[i]) \prod_{u=1}^{i-1}(1{-}g(r[u]))$ in $\mathbb{Z}_2$. These functions can be efficiently evaluated in constant number of rounds with the techniques in [BIB89, CD01].

[2] E.g., for addressing several properties, each property can be stored in a separate vector and then the different obtained vectors are treated simultaneously.

Algorithm 6: Vector to Secret: Find index of the first non-zero value in v. $bit_i(R)$ and $bit_i(j)$ denote the i^{th} bit of R and j, respectively.

function *V2S(r)*
  foreach $i = [0..\lfloor \log_2 k \rfloor]$ **do**
    $bit_i(R) = \oplus_{j=1}^{k}(bit_i(j) \wedge r[j])$;
  return R;

The approach shown in this section has immediate applicability to problems like meeting scheduling where g is the identity function 1_d simply returning the parameter and the elements of v are computed by combining with logic AND the availability of each participant for the corresponding alternative.

6.3 Combinatorial Optimization

Here we will use a simplified but sufficiently illustrative framework for combinatorial optimization (a more complete one appears in [SM04]). Namely, a combinatorial optimization problem has k alternatives $\{\varepsilon_1, ..., \varepsilon_k\}$ and the participants have a secret joint estimation $q(\varepsilon_i)$ for each tuple ε_i. Let B_1 be the best possible value of q and B_2 its upper bound for an acceptable alternative.

Example 5 *Assume each participant A_j can have estimating secret functions $\phi_{j,u}(\varepsilon)$, and each participant can have either none or several such estimating functions (the set of all such functions being C). If these functions cumulate in an additive manner to get a global estimate, then $q(\varepsilon)$=$\sum_{\phi \in C} \phi(\varepsilon)$.*

Minimization A solution is an alternative ε_i such that $q(\varepsilon_i)$ is mimimized. To isolate alternatives ε whose $q(\varepsilon)$ equal some value, x_0 (we need the minimal x_0 for which there exists ε with $q(\varepsilon) = x_0$), we use a function $p(\varepsilon, x_0)$ defined as either:

$$p(\varepsilon, x_0) = \delta(q(\varepsilon) - x_0) \tag{1}$$

or:

$$p(\varepsilon, x_0) = cmp(q(\varepsilon), x_0) \tag{2}$$

$\delta(x)$ is a function returning 1 when $x = 0$ and returning 0 otherwise. $cmp(x, y)$ is a function returning 1 when $x \leq y$ and 0 otherwise. Implementations of $\delta(x)$ and of $cmp(x, y)$ with a constant number of rounds are proposed in [Kil05]. We denote with v_j the vector computed with $v_j[i] = p(\varepsilon_i, j)$. A vector of secrets $\{w_j\}_{j \in [B_j - 1, ..., B_2]}$ is computed where w_j holds the value of i such that ε_i is the best solution with value $q(\varepsilon_i)$ between B_1 and j.

$$w_j \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } j{=}B_1{-}1 \\ V2S(CF(v_j, 1_d)) & \text{if } w_{j-1}{=}0 \\ w_{j-1} & \text{if } w_{j-1} \neq 0 \end{cases}$$

This can be computed with (for $j \in [B_1..B_2]$):

$$\begin{aligned} w_{B_1 - 1} &= 0 \\ w_j &= w_{j-1}(1 - \delta(w_{j-1})) + V2S(CF(v_j, 1_d))\delta(w_{j-1}) \end{aligned}$$

Optimization (minimization) consists of the following phases:

1. First the input secrets ϕ are shared and then shuffled through the mix-net.

2. w_{B_2} is computed by iteratively building the secrets w_j for j increasing from B_1 to B_2 (the shuffled index of the selected best alternative is w_{B_2}).

3. Compute a vector r of k shared secrets with value 1 on position w_{B_2} and value 0 elsewhere. The circuit for this is $S2V(w_{B_2}, k)$ in Algorithm 7.

Algorithm 7: Secret to Vector: Transform a secret s to a vector of k shared secrets with secret 1 on position s and 0 elsewhere. See [Sil04a] for an alternative with $3k$ multiplications of shared secrets (of which $2k$ multiplications can be computed with the constant round method of [BIB89, CD01]).

function *S2V(s, k)*
  foreach $i \in [1..k]$ **do** $r[i] = \delta(s - i)$;

4. Vector r is unsuffled and the index i of the solution ε_i is given by *V2S(r)*.

To learn the value of q for the best alternative one should compute:

$$w = \sum_{u \in [B_1..B_2]} u(1 - \delta(w_u))\delta(w_{u-1})$$

The single non-zero term in the summation defining w is for the round u where w^u is for the first time non-zero. w specifies the value $q(\varepsilon_i)$ of the selected solution ε_i.

Maximization For maximization, the only difference is that computations of w_j are done for j descending from B_2 to B_1, B_2 being the best value. Also, instead of Equation 2 one needs $p(\varepsilon, x_0) = cmp(x_0, q(\varepsilon))$. The value of w_j depends on w_{j+1} (instead of w_{j-1}).

$$w_j \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } j{=}B_2{+}1 \\ V2S(CF(v_j, 1_d)) & \text{if } w_{j+1}{=}0 \\ w_{j+1} & \text{if } w_{j+1} \neq 0 \end{cases}$$

This can be computed with (for $j{\in}[B_1..B_2]$):

$$\begin{aligned} w_{B_2+1} &= 0 \\ w_j &= w_{j+1}(1 - \delta(w_{j+1})) + V2S(CF(v_j, 1_d))\delta(w_{j+1}) \end{aligned}$$

The vector r at optimization phase 3 is computed with *S2V*(w_{B_1}, k).

7 Conclusions

In this article we propose a $(+ \bmod \nu, \times)$-homomorphic encryption scheme that can be parametrized by a public prime value ν and that is obtained with a restriction to the version of ElGamal proposed in [DGS02]. It has applications to a mix-net for securely shuffling shared secrets.

We have shown how participants in a mix-net for secret shares can provide Zero Knowledge proofs of the correctness of their operation. The designed proofs should increase the confidence of all participants that the mix-net did not tamper with the values of the secret shares, and that inverse permutations are correctly applied. The approach is related to the proof of knowing a isomorphism between large graphs. We make a detailed study of the rationales behind each design decision, as well as a detailed review of the major related mix-nets. Alternative versions and implementations (e.g., used cryptosystems) are discussed with their advantages and drawbacks. Discussions related to the usage of public-key versus symmetric-key encryptions for mix-nets are described within the basic techniques of Chaum and Merritt. We show how the technique can be used in the Secure Multi-party Computations (SMC) programming language and we give some examples. The shuffling of secrets using mix-nets offers computational security. We also propose a technique based on arithmetic circuit evaluation that is faster than previous techniques, and offers information theoretical security.

8 Acknowledgements

We thank Jean-Sebastien Coron for his comments.

References

[AS04] T. Atkinson and M.-C. Silaghi. Reply-pay and hansdhaking for incentives with anonymizer servers. Technical Report TR-FIT-16/2004, Florida Institute of Technology, Melbourne, FL, November 2004.

[BIB89] J. Bar-Ilan and D. Beaver. Non-cryptographic fault-tolerant computing in a constant number of rounds interaction. In *8th ACM Symposium Annual on Principles of Distributed Computing*, pages 201–209, August 1989.

[Blu86] M. Blum. How to prove a theorem so no one else can claim it. In *International Congress of Mathematicians*, pages 1444–1451, 1986.

[BOGW88] M. Ben-Or, S. Goldwasser, and A. Widgerson. Completeness theorems for non-cryptographic fault-tolerant distributed computating. In *STOC*, pages 1–10, 1988.

[CD01] Ronald Cramer and Ivan Damgrd. Secure distributed linear algebra in a constant number of rounds. In *Advances in Cryptology (CRYPTO 2001)*, volume 2139 of *LNCS*, pages 119–139, 2001.

[Cha81] D. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Com. of ACM*, 24(2):84–88, 1981.

[DGS02] I. Damgard, J. Groth, and G. Salomonsen. *Secure Electronic Voting*, chapter The theory and Implementation of an electronic voting systems. Kluwer, 2002.

[FM02] Freedman and Morris. Tarzan: a peer-to-peer anonymizing network layer. In *ACM CCS'02*, 2002.

[GB96] S. Goldwasser and M. Bellare. Lecture notes on cryptography. MIT, 1996.

[Gen95] R. Gennaro. 6.915 computer and network security, lecture 24, Dec 1995.

[GMR85] S. Goldwasser, S. Micali, and C. Rackoff. Knowledge complexity of interactive proofs. In *Proc. of 17th STOC*, pages 291–304, 1985. Earlier version: Knowledge complexity, unpublished manuscript, (submitted to FOCS, 1984).

[GMW86] O. Goldreich, S. Micali, and A. Wigderson. Proofs that yield nothing but their validity and a methodology of cryptographic protocol design. In *FOCS*, pages 174–187, Toronto, 1986.

[GMW87] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game — a completeness theorem for protocols with honest majority. In *STOC*, pages 218–229, 1987.

[Gol04] Oded Goldreich. *Foundations of Cryptography*, volume 2. Cambridge, 2004.

[Gro05] Jens Groth. Non-interactive zero-knowledge arguments for voting. In *Applied Cryptography and Network Security*, 2005.

[GRS96] D. Goldschlag, M. Reed, and P. Syverson. Hiding routing information. In *Information Hiding*, number 1174 in LNCS, pages 137–150, 1996.

[JJR02] M. Jakobsson, A. Juels, and R. Rivest. Making mix nets robust for electronic voting by randomized partial checking. In *USENIX*, 2002.

[Kil05] Eike Kiltz. Unconditionally secure constant round multi-party computation for equality, comparison, bits and exponentiation. Cryptology ePrint Archive, Report 2005/066, 2005. http://eprint.iacr.org.

[KS05] K.-R. Kattamuri and M.-C. Silaghi. Supporting debates over citizen initiatives. Technical Report TR-FIT-3/2005, Florida Institute of Techology, Melbourne, FL, January 2005.

[Mah05] Ayan Mahalanobis. Diffie-hellman key exchange protocol and non-abelian nilpotent groups. Cryptology ePrint Archive, Report 2005/110, 2005. http://eprint.iacr.org/.

[Mer83] M. Merritt. *Cryptographic Protocols*. PhD thesis, Georgia Inst. of Tech., Feb 1983.

[Nef01] C. A. Neff. A verifiable secret shuffle and its application to e-voting. In *8th ACM conference on Computer and Communications Security*, pages 116 – 125, 2001.

[Pai99] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Eurocrypt'99*, volume 1592 of *LNCS*, pages 223–238, 1999.

[RR98] Reiter and Rubin. Crowds: Anonymity for web transactions. *ACM Trans. on Information and System Security*, 1(1):66–92, 1998.

[Sil03] M.-C. Silaghi. Solving a distributed CSP with cryptographic multi-party computations, without revealing constraints and without involving trusted servers. In *IJCAI-DCR*, 2003.

[Sil04a] M.-C. Silaghi. Meeting scheduling system guaranteeing n/2-privacy and resistant to statistical analysis (applicable to any DisCSP). In *3rd IC on Web Intelligence*, pages 711–715, 2004.

[Sil04b] M.-C. Silaghi. Secure multi-party computation (SMC) programming language. http://www.cs.fit.edu/ msilaghi/SMC/, 2004.

[Sil04c] Marius C. Silaghi. Secure multi-party computation for selecting a solution according to a uniform distribution over all solutions of a general combinatorial problem. Cryptology e-print Archive 2004/333, 2004.

[Sil05] Marius-Călin Silaghi. Hiding absence of solution for a discsp. In *FLAIRS'05*, 2005.

[SK05] M.-C. Silaghi and K. R. Kattamuri. Publicly verifiable private credentials for citizen initiatives. Technical Report TR-FIT-2/2005, Florida Institute of Techology, Melbourne, FL, January 2005.

[SM04] M.-C. Silaghi and D. Mitra. Distributed constraint satisfaction and optimization with privacy enforcement. In *3rd IC on Intelligent Agent Technology*, pages 531–535, 2004.

[SR04] M.-C. Silaghi and V. Rajeshirke. The effect of policies for selecting the solution of a DisCSP on privacy loss. In *AAMAS*, pages 1396–1397, 2004.

[XNJS04] S. Xu, W. Nelson Jr., and R. Sandhu. Enhancing anonymity via market competition. information assurance and security. In *IEEE ITCC'04*, 2004.