egk mixnet maths

John Caron, 1/13/2024

Preliminary explorations of mixnet implementations to be used with the ElectionGuard Kotlin library.

The ElectionGuard Kotlin library [7] is used for the cryptography primitives. This library closely follows the ElectionGuard 2.0 specification [1].

The math here mostly recapitulates the work of Wikström [6]; Haenni et. al. [2], [3] in explaining the Terelius / Wikström (TW) mixnet algorithm [4], [5]; and the work of Haines [9] that gives a formal proof of security of TW when the shuffle involves vectors of ciphertexts.

Instead of psuedocode, the kotlin code acts as the implementation of the math described here. It can act as a reference and comparison for ports to other languages.

Ive tried to avoid notation that is hard to read, preferring for example, multiple character symbols like pr instead of $\tilde{r}$ or $\hat{r}$, since the glyphs can get too small to read when they are used in exponents or subscripts, and can be hard to replicate in places other than high quality Tex or PDF renderers.

Table of Contents

Definitions

The ElectionGuard Group

- $\mathbb{Z} = \{\dots, -3, -2, -1, 0, 1, 2, 3, \dots\}$ is the set of integers.
- $\mathbb{Z}_n = \{0, 1, 2, \dots, n-1\}$ is the ring of integers modulo n.
- $\mathbb{Z}_n^*$ is the multiplicative subgroup of $\mathbb{Z}_n$ that consists of all invertible elements modulo n. When p is a prime, $\mathbb{Z}_p^* = \{1, 2, 3, \dots, p-1\}$
- $\mathbb{Z}_p^r$ is the set of r-th-residues in $\mathbb{Z}_p^*$. Formally, $\mathbb{Z}_p^r = \{y \in \mathbb{Z}_p^*$ for which there exists $x \in \mathbb{Z}_p^*$ where $y = x^r$ mod p}. When p is a prime for which p − 1 = q * r with q a prime that is not a divisor of the integer r, then $\mathbb{Z}_p^r$ is an order-q cyclic subgroup of $\mathbb{Z}_p^*$, and for any $y \in \mathbb{Z}_p^*$, $y \in \mathbb{Z}_p^r$ if and only if y^q mod p = 1.

The ElectionGuard Kotlin library [7] and ElectionGuard 2.0 specification [1] is used for the cryptography primitives, in particular the parameters for $\mathbb{Z}_p^r$, the variant of ElGamal encryption described next, and the use of HMAC-SHA-256 for hashing.

Permutations

A *permutation* is a bijective map $\psi : 1..N \to 1..N$. We use **px** to mean the permutation of a vector **x**, **px** = $\psi(\mathbf{x})$, so that x_i = px_j, where $i = \psi(j)$ and $j = \psi^{-1}(i)$. $x_i = px_{\psi^{-1}(i)}$, $px_j = x_{\psi(j)}$,

A *permutation* ψ has a *permutation matrix* B_ψ , where b_{ij} = 1 if $\psi(i)$ = j, otherwise 0. Note that $\psi(\mathbf{x})$ = **px** = B**x** (matrix multiply).

If B_ψ = (b_{ij}) is an N -by-N matrix over $\mathbb{Z}_q$ and **x** = $(x_1, \dots, x_N)$ a vector of N independent variables, then B_ψ is a permutation matrix if and only

$$\sum_{i=1}^{n} b_{ij} = 1 \quad (Condition\ 1)$$

$$\sum_{i=1}^{n} \sum_{j=1}^{n} b_{ij} x_i = \sum_{i=1}^{n} x_i \quad (Condition\ 2)$$

ElGamal Encryption and Reencryption

$$(2a)$$
$$Encr(m, \xi) = (g^{\xi}, K^{m+\xi}) = (a, b)$$
$$Encr(0, \xi') = (g^{\xi'}, K^{\xi'})$$

$$(2b)$$
$$(a, b) * (a', b') = (a * a', b * b')$$
$$Encr(m, \xi) * Encr(m', \xi') = (g^{\xi+\xi'}, K^{m+m'+\xi+\xi'}) = Encr(m + m', \xi + \xi')$$

$$(2c)$$
$$(a, b)^k = (a^k, b^k)$$
$$Encr(m, \xi)^k = (g^{\xi*k}, K^{(m*k+\xi*k)}) = Encr(m * k, \xi * k)$$

$$(2d)$$
$$\prod_{j=1}^{n} Encr(m_j, \xi_j) = (g^{\sum_{j=1}^{n} \xi_j}, K^{\sum_{j=1}^{n} m_j + \sum_{j=1}^{n} \xi_j}) = Encr(\sum_{j=1}^{n} m_j, \sum_{j=1}^{n} \xi_j)$$
$$\prod_{j=1}^{n} Encr(m_j, \xi_j)^{k_j} = Encr(\sum_{j=1}^{n} (m_j * k_j), \sum_{j=1}^{n} (\xi_j * k_j))$$

$$(2e)$$
$$ReEncr(m, r) = (g^{\xi+r}, K^{m+\xi+r}) = Encr(0, r) * Encr(m, \xi)$$
$$ReEncr(m, r)^k = Encr(0, r * k) * Encr(m * k, \xi * k)$$

$$(2f)$$
$$\prod_{j=1}^{n} ReEncr(e_j, r_j) = (g^{\sum_{j=1}^{n} (\xi_j + r_j)}, K^{\sum_{j=1}^{n} (m_j + \xi_j + r_j)})$$
$$= ReEncr(\prod_{j=1}^{n} e_j, \sum_{j=1}^{n} r_j)$$

$$(2e)$$
$$\prod_{j=1}^{n} ReEncr(m_j, r_j)^{k_j} = \prod_{j=1}^{n} Encr(0, r_j * k_j) * \prod_{j=1}^{n} Encr(m_j * k_j, \xi_j * k_j)$$
$$= Encr(0, \sum_{j=1}^{n} (r_j * k_j)) * \prod_{j=1}^{n} Encr(m_j, \xi_j)^{k_j}$$

Let

1. $e_j = Encr(m_j, \xi_j)$
2. $re_j = ReEncr(m_j, r_j) = ReEncr(e_j, r_j) = Encr(0, r_j) * e_j$

Then

$$
\begin{aligned}
re_j &= Encr(0, r_j) * e_j \\
\prod_{j=1}^{n} re_j^{k_j} &= \prod_{j=1}^{n} Encr(0, r_j)^{k_j} * \prod_{j=1}^{n} e_j^{k_j} \\
&= Encr(0, \sum_{j=1}^{n} (r_j * k_j)) * \prod_{j=1}^{n} e_j^{k_j}, \quad (Equation\ 1)
\end{aligned}
$$

Verificatum

Pedersen Commitments

For a set of messages $\mathbf{m} = (m_1 .. m_n) \in \mathrm{Z}_q$, the *Pedersen committment* to $\mathbf{m}$ is

$$Commit(\mathbf{m}, cr) = g^{cr} * h_1^{m_1} * h_2^{m_2} * .. h_n^{m_n} = g^{cr} * \prod_{i=1}^{n} h_i^{m_i}$$

where ($g, \mathbf{h}$) are generators of $\mathbb{Z}_p^r$ with randomization nonce $cr \in Z_q$. (section 1.2 of [4])

If $\mathbf{b}_i$ is the i^{th} column of B_ψ, then the *permutation commitment to* ψ is defined as the vector of committments to its columns:

$$Commit(\psi, \mathbf{cr}) = (Commit(\mathbf{b}_1, cr_1), Commit(\mathbf{b}_2, cr_2), .. Commit(\mathbf{b}_N, cr_N)) =$$

where

$$c_j = Commit(\mathbf{b}_j, cr_j) = g^{cr_j} * \prod_{i=1}^{n} h_i^{b_{ij}} = g^{cr_j} * h_i, \ for\ i = \psi^{-1}(j)$$

Definitions

Let

- n = number of rows (eg ballots)
- width = number of ciphertexts in each row
- W = matrix of ciphertexts (n x width), with entries $w_{i,j}$; its row vectors of width ciphertexts are $\vec{w}_i, i = 1..n$; and its column vectors of n ciphertexts are $\vec{w}_j, j = 1..width$
- W' = matrix of shuffled and reencrypted ciphertexts (n x width), with entries, row vectors and column vectors $w'_{i,j}, \vec{w'}_i, \vec{w'}_j$ respectively
- R = matrix of reencryption nonces $\in \mathbb{Z}_q$ (unpermuted)
- ψ = permutation function
- ψ^{-1} = inverse permutation function
- $\vec{h}$ = generators of $\mathbb{Z}_p^r, h_0 = \vec{h}_1$

Mix

Choose R = (n x width) matrix of reencryption random nonces, ie separate nonces for each ciphertext.

W' = ψ^{-1}(Reencrypt(W, R))

Proof Construction

The Proof equations are reverse engineered from reading the Verificatum code. AFAIK, there is no documentation of these except in the Verificatum code, in particular not in [6], although likely they are implied in [4] using different notation. In any case, these equations are implemented in the kotlin library *ShuffleProver* and verify with *ShuffleVerifier*. The *ShuffleVerifier* also verifies against the proofs output by Verificatum itself, leading to some confidence that these equations capture the TW algorithm as implemented in Verificatum.

Commitment to permutation

Choose a vector of *n* random permutation nonces $\vec{pn}$.
Form permutation commitments $\vec{u}$ that will be public:

$$u_j = g^{pn_j} \cdot h_i, \quad j = \psi(i) \quad TODO$$

Commitment to shuffle

Compute *n* nonces $\vec{e}$ that will be public. Let e' = $\psi^{-1}(\vec{e})$.
Choose vectors of *n* random nonces $\vec{b}, \vec{\beta}, \vec{\epsilon}$.
Choose random nonces α, γ, δ .

Form the following values $\in \mathbb{Z}_p^r$:

$$A' = g^{\alpha} \prod_{i=1}^{n} h_i^{\epsilon_i}$$
$$B_i = g^{b_i}(B_{i-1})^{e'_i},\ where\ B_0 = h_0,\ i = 1..N$$
$$B'_i = g^{\beta_i}(B_{i-1})^{\epsilon_i},\ where\ B_0 = h_0,\ i = 1..N$$
$$C' = g^{\gamma}$$
$$D' = g^{\delta}$$

Commitment to exponents

Choose *width* random nonces $\vec{\phi}$.

Form the following ciphertext values:

$$F'_j = Encr(0, -\phi_j) \cdot \prod_{i=1}^{n} (w'_{i,j})^{\epsilon_i}\ ,\ j = 1..width$$

Note that $\vec{F'}$ has *width* components, one for each of the column vectors of $W' = \vec{w'}_j$. For each column vector, form the product of it exponentiated with $\vec{\epsilon}$. We can use any of the following notations:

$$= \prod_{i=1}^{n} (w'_{i,j})^{\epsilon_i},\ j = 1..width$$
$$= \prod_{i=1}^{n} (\vec{w'}_j)_i^{\epsilon_i}$$
$$= \prod_{i=1}^{n} (W')^{\epsilon}$$

This disambiguates the equations in Algorithm 19 of [6], for example: $\prod w_i^{e_i}$. and $\prod (w'_i)^{k_{E,i}}$.

Reply to challenge v:

A challenge v $\in \mathbb{Z}_q$ is given, and the following values $\in \mathbb{Z}_q$ are made as reply:

$$k_A = v \cdot < \vec{pn} \cdot \vec{e} > + \alpha$$
$$\vec{k_B} = v \cdot \vec{b} + \vec{\beta}$$
$$k_C = v \cdot \sum_{i=1}^{n} pn_i + \gamma$$
$$k_D = v \cdot d + \delta$$
$$\vec{k_E} = v \cdot \vec{e'} + \vec{\epsilon}$$

and

$$Let\ \vec{R}_j = \ jth\ column\ of\ reencryption\ nonces\ R$$
$$k_{F,j} = v \cdot < \vec{R}_j, \vec{e'} > + \phi_j\ ,\ j = 1..width$$

where < , > is the inner product of two vectors.

Proof of Shuffle Data Structure

```
data class ProofOfShuffle(
    val mixname: String,
    val u: VectorP, // permutation commitment

    // τ^pos = Commitment of the Fiat-Shamir proof.
    val B: VectorP,
    val Ap: ElementModP,
    val Bp: VectorP,
    val Cp: ElementModP,
    val Dp: ElementModP,
    val Fp: VectorCiphertext, // width

    // σ^pos = Reply of the Fiat-Shamir proof.
    val kA: ElementModQ,
    val kB: VectorQ,
    val kC: ElementModQ,
    val kD: ElementModQ,
    val kE: VectorQ,
```

```
    val kF: VectorQ, // width
)
```

Proof Verification

The following equations are taken from Algorithm 19 of [6] and checked against the Verificatum implementation. The main ambiguity is in the meaning of $\prod_{i=1}^{n} w_i^{e_i}$ and $\prod_{i=1}^{n} wp_i^{k_{E,i}}$ in steps 3 and 5. These are interpreted as a short hand for *width* equations on the column vectors of *w* and *wp*. We use one-based array indexing for notational simplicity.

The Verifier is provided with:

- n = number of rows
- width = number of ciphertexts in each row
- $\vec{w}$ = rows of ciphertexts (n x width)
- $\vec{wp}$ = shuffled and reencrypted rows of ciphertexts (n x width)
- $h_0, \vec{h}$ = generators of $\mathbb{Z}_p^r$
- *ProofOfShuffle* and *Reply*

The $\vec{h}$ (generators), $\vec{e}$ nonces, and challenge are deterministically recalculated. This prevents those from being carefully chosen to subvert the proof.

The following values $\in \mathbb{Z}_p^r$ are computed:

$$A = \prod_{i=1}^{n} u_i^{e_i}$$
$$C = (\prod_{i=1}^{n} u_i)/(\prod_{i=1}^{n} h_i)$$
$$D = B_n \cdot h_0^{\prod_{i=1}^{n} e_i}$$

and

$$F_j = \prod_{i=1}^{n} (w_{i,j})^{e_i}\ ,\ j = 1..width$$

Then the following are checked, and if all are true, the verification succeeds:

$$A^v \cdot A' = g^{k_A} \prod_{i=1}^{n} h_i^{k_{E,i}}$$
$$B_i^v \cdot B_i' = g^{k_{B,i}} (B_{i-1})^{k_{E,i}},\ where\ B_0 = h_0,\ i = 1..n$$
$$C^v \cdot C' = g^{k_C}$$
$$D^v \cdot D' = g^{k_D}$$

and

$$F_j^v F'_j = Encr(0, -k_{F,j}) \prod_{i=1}^{n} (w'_{i,j})^{k_{E,i}},\ j = 1..width$$

issues

Calculation of $\vec{h}$ (generators) , $\vec{e}$ and the challenge nonces are highly dependent on the VMN implementation. The verifier is expected to independently generate, ie they are not part of the ProofOfShuffle output.

generators may need to be carefully chosen, see section 6.8 of vmnv: "In particular, it is not acceptable to derive exponents x1 , . . . , xN in Zq and then define hi = g^xi"

ChVote

This follows Haenni et. al. [2], which has a good explanation of TW, sans vectors.

Pedersen Commitments

For a set of messages $\mathbf{m} = (m_1 .. m_n) \in \mathbb{Z}_q$, the *Extended Pedersen committment* to $\mathbf{m}$ is

$$Commit(\mathbf{m}, cr) = g^{cr} * h_1^{m_1} * h_2^{m_2} * .. h_n^{m_n} = g^{cr} * \prod_{i=1}^{n} h_i^{m_i}$$

where ($g, \mathbf{h}$) are generators of $\mathbb{Z}_p^r$ with randomization nonce $cr \in Z_q$.

If $\mathbf{b}_i$ is the i^{th} column of B_ψ, then the *permutation commitment to* ψ is defined as the vector of committments to its columns:

$$Commit(\psi, \mathbf{cr}) = (Commit(\mathbf{b}_1, cr_1), Commit(\mathbf{b}_2, cr_2), .. Commit(\mathbf{b}_N, cr_N)) =$$

where

$$c_j = Commit(\mathbf{b}_j, cr_j) = g^{cr_j} * \prod_{i=1}^{n} h_i^{b_{ij}} = g^{cr_j} * h_i,\ for\ i = \psi^{-1}(j)$$

Proof of permutation

Let **c** = $Commit(\psi, \mathbf{r})$ = $(c_1, c_2, .. c_N)$, with randomization vector **cr** = $(cr_1, cr_2, .. cr_N)$, and $crbar = \sum_{i=1}^{n} cr_i$.

$Condition$ 1 implies that

$$\prod_{j=1}^{n} c_j = \prod_{j=1}^{n} g^{cr_j} \prod_{i=1}^{n} h_i^{b_{ij}} = g^{crbar} \prod_{i=1}^{n} h_i \ = Commit(\mathbf{1}, crbar). \quad (5.2)$$

Let $\mathbf{u} = (u_1 .. u_n)$ be arbitrary values $\in \mathrm{Z}_q$, $\mathbf{pu}$ its permutation by ψ, and $cru = \sum_{j=1}^{N} cr_j u_j$.

$Condition$ 2 implies that:

$$\prod_{i=1}^{n} u_i = \prod_{j=1}^{n} pu_j \quad (5.3)$$

$$\prod_{j=1}^{n} c_j^{u_j} = \prod_{j=1}^{n} (g^{cr_j} \prod_{i=1}^{n} h_i^{b_{ij}})^{u_j} = g^{cru} \prod_{i=1}^{n} h_i^{pu_i} \ = Commit(\mathbf{pu}, cru) \quad (5.4)$$

Which constitutes proof that condition 1 and 2 are true, so that c is a commitment to a permutation matrix.

Proof of equal exponents

Let $\mathbf{m}$ be a vector of messages, **e** their encryptions **e** = Encr($\mathbf{m}$), and **re(e, r)** their reenryptions with nonces **r**. A shuffle operation both reencrypts and permutes, so $shuffle(\mathbf{e}, \mathbf{r}) \rightarrow (\mathbf{pre}, \mathbf{pr})$, where **pre** is the permutation of **re** by ψ, and **pr** the permutation of **r** by ψ.

$$re_i = ReEncr(e_i, r_i) = Encr(0, r_i) * e_i$$

$$pre_j = ReEncr(pe_j, pr_j) = Encr(0, pr_j) * e_j$$

Let **u** be arbitrary values $\in \mathbb{Z}_q$ (to be specified later) and **pu** its permutation.

If the shuffle is valid, then it follows from $Equation$ 1 above that

$$\begin{aligned}
\prod_{j=1}^{n} pre_j^{pu_j} &= \prod_{j=1}^{n} (Encr(0, pr_j) * e_j)^{pu_j} \\
&= Encr(0, \sum_{j=1}^{n} (pr_j * pu_j)) * \prod_{j=1}^{n} e_j^{pu_j} \quad (Equation\ 1) \\
&= Encr(0, sumru) * \prod_{j=1}^{n} e_j^{pu_j}
\end{aligned}$$

where $sumru = \sum_{j=1}^{n} (pr_j * pu_j)$.

However, $e_j^{pu_j} = e_i^{u_i}$ for some i, so $\prod_{j=1}^{n} e_j^{pu_j} = \prod_{i=1}^{n} e_i^{u_j}$, and we have:

$$\prod_{j=1}^{n} pre_j^{pu_j} = Encr(0, sumru) * \prod_{i=1}^{n} e_i^{u_i} \quad (5.5)$$

Note that (5.5) from [2] and line 141 of the code in *GenShuffleProof* in [8] has

$$Encr(1, \tilde{r}),\ where\ \tilde{r} = \sum_{j=1}^{n} pr_j * u_j$$

whereas we have

$$Encr(0, \tilde{r}),\ where\ \tilde{r} = \sum_{j=1}^{n} pr_j * pu_j$$

The $Encr(0, . .)$ is because we use exponential ElGamal, so is fine. Their use of u_j instead of pu_j appears to be a mistake. Its also possible there is a difference in notation that I didnt catch.

Shuffling vectors

Simple

Much of the literature assumes that each row to be mixed consists of a single ciphertext. In our application we need the possibility that each row consists of a vector of ciphertexts.
So for each row i, we now have a vector of *w* = *width* ciphertexts:

$$\mathbf{e}_i = (e_{i,1}, . . e_{i,w}) = \{e_{i,k}\},\ k = 1 .. w$$

The main work is to modify the proof of equal exponents for this case.

Suppose we are looking for the simplest generalization of 5.5:

$$\prod_{j=1}^{n} pre_j^{pu_j} = Encr(0, sumru) \cdot \prod_{i=1}^{n} e_i^{u_i} \quad (5.5)$$

one could use the same nonce for all the ciphertexts in each row when reencrypting:

$$\mathbf{r} = \{r_j\}, j = 1 .. n$$
$$re_{j,k} = ReEncr(e_{j,k}, r_j) = Encr(0, r_j) \cdot e_{j,k} \quad (case1)$$

or generate N = nrows * width nonces, one for each ciphertext:

$$\mathbf{r} = \{r_{j,k}\},\ j = 1 .. n,\ k = 1 .. w$$
$$re_{j,k} = ReEncr(e_{j,k}, r_{j,k}) = Encr(0, r_{j,k}) \cdot e_{j,k} \quad (case2)$$

Then eq 5.5 is changed to

$$\prod_{j=1}^{n} \prod_{k=1}^{w} pre_{j,k}^{pu_j} = Encr(0, sumru') * \prod_{i=1}^{n} \prod_{k=1}^{w} e_{i,k}^{u_i}$$

where, now

$$\begin{aligned} sumru' &= \sum_{j=1}^{n} width * (pr_j * pu_j) \quad (case1) \\ &= \sum_{j=1}^{n} \sum_{k=1}^{n} (pr_{j,k} * pu_j) \quad (case2). \end{aligned}$$

In algorithms 8.4, 8.5 of [2], the challenge includes a list of all the ciphertexts and their reencryptions in their hash function:

$$\mathbf{u} = Hash(\dots, \mathbf{e}, \mathbf{pe}, pcommit, pkq, i, \dots)$$

Here we just flatten the list of lists of ciphertexts for $\mathbf{e}, \mathbf{pe}$, so that all are included in the hash. Since the hash is dependent on the ordering of the hash elements, this should preclude an attack that switches ciphertexts within a row.

Haines Proof of vector shuffling

Haines [9] gives a formal proof of security of TW when the shuffle involves vectors of ciphertexts.

We will use the notation above for case 2, using a separate nonce for each ciphertext:

$$\mathbf{r} = \{r_{j,k}\},\ j = 1..n,\ k = 1..w$$
$$re_{j,k} = ReEncr(e_{j,k}, r_{j,k}) = Encr(0, r_{j,k}) \cdot e_{j,k} \quad (case2)$$

This gives an nrows x width matrix R of reencryption nonces. The vector notation is a shorthand for component-wise operations:

$$R = (\mathbf{r}_1, .. \mathbf{r}_n)$$
$$Encr(\mathbf{e}_i) = (Encr(e_{i,1}), .. Encr(e_{i,w}))$$
$$ReEncr(\mathbf{e}_i, \mathbf{r}_i) = (ReEncr(e_{i,1}, r_{i,1}), .. ReEncr(e_{i,1}, r_{i,w}))$$

so now we have vector equations for rencryption:

$$\mathbf{re}_i = ReEncr(\mathbf{e}_i, \mathbf{r}_i) = Encr(0, \mathbf{r}_i) * \mathbf{e}_i$$

and the permuted form, as is returned by the shuffle:

$$\mathbf{pre}_j = ReEncr(\mathbf{pe}_j, \mathbf{pr}_j) = Encr(0, \mathbf{pr}_j) * \mathbf{e}_j$$

which corresponds to ntnu equation (p 3) of [9]:

$$\mathbf{e}'_i = ReEnc(\mathbf{e}_{\pi(i)}, R_{\pi(i)}), \pi = \pi_M$$

Let **ω** be width random nonces, **ω'** = permuted **ω**, and $\mathbf{pe}_i$ = permuted $\mathbf{e}_i = \mathbf{e}'_i$ as before. Then the t_4 equation (p 3, paragraph 2 of [9]) is a vector of width components:

$$\mathbf{t}_4 = ReEnc(\prod_i^n \mathbf{pe}_i^{\boldsymbol{\omega}'_i}, -\boldsymbol{\omega}_4)$$
$$= (ReEnc(\prod_i^n \mathbf{pe}_i^{\boldsymbol{\omega}'_i}, -\boldsymbol{\omega}_{4,1}), .. (ReEnc(\prod_i^n \mathbf{pe}_i^{\boldsymbol{\omega}'_i}, -\boldsymbol{\omega}_{4,w}))$$

where

$$\prod_i^n \mathbf{pe}_i^{\boldsymbol{\omega}'_i}$$

must be the product over rows of the k_{th} ciphertext in each row:

$$(\prod_i^n \mathbf{pe}_{i,1}^{\boldsymbol{\omega}_i'}, .. \prod_i^n \mathbf{pe}_{i,w}^{\boldsymbol{\omega}_i'})$$

$$= \{\prod_i^n \mathbf{pe}_{i,k}^{\boldsymbol{\omega}_i'}\}, k = 1..width$$

$$\mathbf{t}_4 = \{Rencr(\prod_i^n \mathbf{pe}_{i,k}^{\boldsymbol{\omega}_i'}, -\boldsymbol{\omega}_4)\}, k = 1..width$$

(quite a bit more complicated than "our simplest thing to do" above)

extra

to go back to (2f) and unravel this:

$$\prod_{j=1}^n ReEncr(e_j, r_j) = ReEncr(\prod_{j=1}^n e_j, \sum_{j=1}^n r_j) \quad (2f)$$

$$\prod_{j=1}^n ReEncr(\mathbf{pe}_i^{\boldsymbol{\omega}_i'}, r_j) = ReEncr(\prod_{j=1}^n \mathbf{pe}_i^{\boldsymbol{\omega}_i'}, \sum_{j=1}^n r_j)$$

Timings vs Verificatum (preliminary)

Environment used for testing:

- Ubuntu 22.04.3
- HP Z840 Workstation, Intel Xeon CPU E5-2680 v3 @ 2.50GHz
- 24-cores, two threads per core.

Regular vs accelerated exponentiation time

Regular exponentiation is about 3 times slower after the acceleration cache warms up:

```
acc took 15288 msec for 20000 = 0.7644 msec per acc
exp took 46018 msec for 20000 = 2.3009 msec per exp
exp/acc = 3.01007326007326
```

VMN

Operation counts

- *n* = number of rows, eg ballots or contests
- *width* = number of ciphertexts per row
- *N* = nrows * width = total number of ciphertexts to be mixed

	shuffle	proof of shuffle	proof of exp	verify
regular exps	0	4 * n	2 * N	4*N + 4 * n + 4
accelerated exps	2 * N	3 * n + 2 * width + 4	0	n + 2*width + 3

Even though N dominates, width is bound but nrows can get arbitrarily big.

Could break into batches of 100-1000 ballots each and do each batch in parallel. The advantage here is that there would be complete parallelization.

Timing results

See VMN spreadsheets for graphs of timing results (work in progress).

OpenChVote

operations count

	shuffle	proof	verify
regular exps	0	2*N* + 5n	4*N* + 4n + 6
accelerated exps	2 * N	2*N* + 3n	8

wallclock time vs verificatum

nrows = 1000, width = 34, N=3400

```
Time verificatum as used by rave

RunMixnet elapsed time = 67598 msecs
RunMixnet elapsed time = 67853 msecs)
RunMixnetVerifier elapsed time = 68855 msecs
RunMixnetVerifier elapsed time = 68738 msecs
```

```
nrows=1000, width= 34 per row, N=34000, nthreads=24

shuffle: took 5511 msecs
proof: took 12944 msecs
verify: took 27983 msecs
total: took 46438 msecs
```

nrows = 100, width = 34, N=3400

```
Time verificatum as used by rave

RunMixnet elapsed time = 27831 msecs
RunMixnet elapsed time = 26464 msecs)
RunMixnetVerifier elapsed time = 12123 msecs
RunMixnetVerifier elapsed time = 12893 msecs

total = 79.311 secs
```

```
Time egk-mixnet

  shuffle1 took 5505
  shuffleProof1 took 17592
  shuffleVerify1 took 33355
  shuffle2 took 5400
  shuffleProof2 took 17213
  shuffleVerify1 took 33446

  total: 119.711 secs, N=3400 perN=35 msecs
```

Vmn proof 27/(17.4+5.4) = 1.18 is 18% slower

Vmn has verifier 33355/12123 = 2.75 faster, TODO: investigate if theres an algorithm improvement there. Possibly related to the "wide integer" representation, eg see

```
LargeInteger.modPowProd(LargeInteger[] bases, LargeInteger[] exponents, LargeInteger
modulus)
```

More likely there are parallelization being done, eg in the same routine. So to compare, we have to run vmn and see what parelization it gets.

Also note LargeInteger.magic that allows use of VMGJ.

Vmn in pure Java mode, using BigInteger. TODO: Find out how much speedup using VMGJ gets.

SO why doesnt same speedup apply to proof?

References

1. Josh Benaloh and Michael Naehrig, *ElectionGuard Design Specification, Version 2.0.0*, Microsoft Research, August 18, 2023, https://github.com/microsoft/electionguard/releases/download/v2.0/EG_Spec_2_0.pdf
2. Rolf Haenni, Reto E. Koenig, Philipp Locher, Eric Dubuis. *CHVote Protocol Specification Version 3.5*, Bern University of Applied Sciences, February 28th, 2023, https://eprint.iacr.org/2017/325.pdf
3. R. Haenni, P. Locher, R. E. Koenig, and E. Dubuis. *Pseudo-code algorithms for verifiable re-encryption mix-nets*. In M. Brenner, K. Rohloff, J. Bonneau, A. Miller, P. Y. A.Ryan, V. Teague, A. Bracciali, M. Sala, F. Pintore, and M. Jakobsson, editors, FC'17, 21st International Conference on Financial Cryptography, LNCS 10323, pages 370–384, Silema, Malta, 2017.
4. B. Terelius and D. Wikström. *Proofs of restricted shuffles*, In D. J. Bernstein and T. Lange, editors, AFRICACRYPT'10, 3rd International Conference on Cryptology inAfrica, LNCS 6055, pages 100–113, Stellenbosch, South Africa, 2010.
5. D. Wikström. *A commitment-consistent proof of a shuffle.* In C. Boyd and J. González Nieto, editors, ACISP'09, 14th Australasian Conference on Information Security and Privacy, LNCS 5594, pages 407–421, Brisbane, Australia, 2009.
6. D. Wikström. *How to Implement a Stand-alone Verifier for the Verificatum Mix-Net VMN Version 3.1.0*, 2022-09-10, https://www.verificatum.org/files/vmnv-3.1.0.pdf
7. John Caron, Dan Wallach, *ElectionGuard Kotlin library*, https://github.com/votingworks/electionguard-kotlin-multiplatform
8. E-Voting Group, Institute for Cybersecurity and Engineering, Bern University of Applied Sciences, *OpenCHVote*, https://gitlab.com/openchvote/cryptographic-protocol
9. Thomas Haines, *A Description and Proof of a Generalised and Optimised Variant of Wikström's Mixnet*, arXiv:1901.08371v1 [cs.CR], 24 Jan 2019