egk-mixnet-maths DRAFT

John Caron

1/17/2024

Preliminary explorations of mixnet implementations to be used with the ElectionGuard Kotlin library [10].

Derived mostly from [4] and [3].

The math here mostly recapitulates the work of Wikström [3]; Haenni et. al. [5] [6], in explaining the Terelius / Wikström (TW) mixnet algorithm[1][2]; and the work of Haines [8] that gives a formal proof of security of TW when the shuffle involves vectors of ciphertexts.

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

Contents

1 Definitions

1.1 The ElectionGuard Group

- $Z = \{..., -3, -2, -1, 0, 1, 2, 3, ...\}$ is the set of integers.
- $Z_n = \{0, 1, 2, ..., n-1\}$ is the ring of integers modulo n.
- Z_n^* is the multiplicative subgroup of Z_n that consists of all invertible elements modulo n. When p is a prime, $Z_p^* = \{1, 2, 3, ..., p-1\}$
- Z_p^r is the set of r-th-residues in Z_p^* . Formally, $Z_p^r = \{y \in Z_p^*$ for which there exists $x \in Z_p^*$ 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 Z_p^r is an order-q cyclic subgroup of Z_p^* , and for any $y \in Z_p^*$, $y \in Z_p^r$ if and only if y^q mod p = 1.

The ElectionGuard Kotlin library [10] and ElectionGuard 2.0 specification [9] is used for the cryptography primitives, in particular the parameters for Z_p^r, the variant of ElGamal encryption described next, and the use of HMAC-SHA-256 for hashing

1.2 ElGamal Encryption and Reencryption

$$Encr(m,\xi) = (g^{\xi}, K^{m+\xi}) = (a,b) \tag{1.2a}$$
$$Encr(0,\xi') = (g^{\xi'}, K^{\xi'})$$

$$(a,b) * (a',b') = (a*a', b*b') \tag{1.2b}$$
$$Encr(m,\xi) * Encr(m',\xi') = (g^{\xi+\xi'}, K^{m+m'+\xi+\xi'}) = Encr(m+m', \xi+\xi')$$

$$(a,b)^k = (a^k, b^k) \tag{1.2c}$$
$$Encr(m,\xi)^k = (g^{\xi*k}, K^{(m*k+\xi*k)}) = Encr(m*k, \xi*k)$$

$$\prod_{j=1}^{n} Encr(m_j,\xi_j) = (g^{\sum_{j=1}^{n}\xi_j}, K^{\sum_{j=1}^{n} m_j + \sum_{j=1}^{n}\xi_j}) = Encr(\sum_{j=1}^{n} m_j, \sum_{j=1}^{n}\xi_j) \tag{1.2d}$$
$$\prod_{j=1}^{n} Encr(m_j,\xi_j)^{k_j} = Encr(\sum_{j=1}^{n}(m_j*k_j), \sum_{j=1}^{n}(\xi_j*k_j))$$

$$ReEncr(m,r) = (g^{\xi+r}, K^{m+\xi+r}) = Encr(0,r) * Encr(m,\xi) \tag{1.2e}$$
$$ReEncr(m,r)^k = Encr(0, r*k) * Encr(m*k, \xi*k)$$

$$\prod_{j=1}^{n} ReEncr(e_j,r_j) = (g^{\sum_{j=1}^{n}(\xi_j+r_j)}, K^{\sum_{j=1}^{n}(m_j+\xi_j+r_j)}) \tag{1.2f}$$
$$= ReEncr(\prod_{j=1}^{n} e_j, \sum_{j=1}^{n} r_j)$$

$$\prod_{j=1}^{n} ReEncr(m_j,r_j)^{k_j} = \prod_{j=1}^{n} Encr(0, r_j*k_j) * \prod_{j=1}^{n} Encr(m_j*k_j, \xi_j*k_j) \tag{1.2g}$$
$$= Encr(0, \sum_{j=1}^{n}(r_j*k_j)) * \prod_{j=1}^{n} Encr(m_j,\xi_j)^{k_j}$$

Let

- $e_j = Encr(m_j, \xi_j)$
- $re_j = ReEncr(m_j, r_j) = ReEncr(e_j, r_j) = Encr(0, r_j) * e_j$

then

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

1.3 Permutations

A *permutation* is a bijective map $\psi : 1..N \to 1..N$. We use **px** to mean the permutation of a vector **x**, $\mathbf{px} = \psi(\mathbf{x})$, 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 $\psi(\mathbf{x}) = \mathbf{px} =$ B$\mathbf{x}$ (matrix multiply).

If $B_\psi = (b_{ij})$ is an n-by-n matrix overZ_q and $\mathbf{x} = (x_1, ..., x_n)$ a vector of N independent variables, then B_ψ is a permutation matrix if and only

$$\begin{aligned} \sum_{i=1}^{n} b_{ij} = 1, \text{ for all j (Equation 1)} \\ \sum_{i=1}^{n}\sum_{j=1}^{n} b_{ij}x_i = \sum_{i=1}^{n} x_i \text{ (Equation 2)} \end{aligned}$$

1.4 Pedersen Commitments

For a set of messages $\mathbf{m} = (m_1..m_n) \in Z_q$, the *Pedersen committment* to **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 Z_p^r with randomization nonce $cr \in Z_q$. (section 1.2 of)

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

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

2 TW Algorithm

2.1 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 $\in Z_q$ (unpermuted)
- ψ = permutation function
- ψ^{-1} = inverse permutation function
- $\vec{h}$ = generators of $Z_p^r, h_0 = \vec{h}_1$

We use one-based array indexing for notational simplicity.

2.2 Shuffle

Choose R = (n x width) matrix of reencryption random nonces, ie separate nonces for each ciphertext. The the shuffle of W is:

$$W' = \psi^{-1}(Reencrypt(W, R))$$

2.3 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 [3], although likely they are implied in [1] 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.

2.3.1 Commitment to permutation

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

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

2.3.2 Commitment to shuffle

Compute n nonces $\vec{e}$ that will be public. Let $e' = \psi^{-1}(\vec{e})$.
Choose vectors of n random nonces $\vec{b}, \vec{\beta}, \vec{eps}$.
Choose random nonces α, γ, δ .
Form the following values $\in Z_p^r$:

$$\begin{aligned}
&A' = g^{\alpha} \prod_{i=1}^{n} h_i^{eps_i} \\
&B_i = g^{b_i}(B_{i-1})^{e'_i},\ where\ B_0 = g^{b_0} h_0^{e'_0},\ i = 1..n \\
&B'_1 = g^{\beta_1} h_0^{eps_1},\ B'_i = g^{\beta_i}(B_{i-1})^{eps_i},\ i = 2..n \\
&C' = g^{\gamma} \\
&D' = g^{\delta}
\end{aligned}$$

Commitment to exponents

Choose *width* random nonces $\vec{\phi}$.
Form the following ciphertext values:

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

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 component-wise product of it exponentiated with $\vec{eps}$. We can use any of the following notations:

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

This disambiguates the equations in Algorithm 19 of [3], for example: $\prod w_i^{e_i}$. and $\prod (w'_i)^{k_{E,i}}$.

2.3.3 Reply to challenge v:

A challenge v $\in Z_q$ is given, and the following values $\in Z_q$ are made as reply:

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

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.

2.4 ProofOfShuffle Data Structure

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

    // 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

    // 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
)
```

2.5 Proof Verification

The following equations are taken from Algorithm 19 of [3] and checked against the Verificatum implementation. The main ambiguity is in the meaning of $\prod_{i=1}^{n} w_i^{e_i}$ and $\prod_{i=1}^{n} (w_i')^{k_{E,i}}$ in steps 3 and 5. These are interpreted as a short hand for *width* equations on the column vectors of W and W', respectively, as detailed in *committment to exponents* section above.

The Verifier is provided with:

- n = number of rows
- width = number of ciphertexts in each row
- W = rows of ciphertexts (n x width)
- W' = shuffled and reencrypted rows of ciphertexts (n x width)
- the ProofOfShuffle

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 $\in Z_p^r$ 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:

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

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 $h_i = g^{x_i}$"

3 Timings

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.

3.1 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
```

3.2 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	verify
regular exps	0	2N + 4n	4N + 5n + 4
accelerated exps	2N	3n + 2width + 4	n + 2width + 4

Table 1: Operation count

	shuffle	proof	verify
regular exps	0	2N + n - 1	4N + 4n + 1
accelerated exps	2N	6n + 2width + 4	2n + 2width + 6

Table 2: Operation count with alternate B, B'

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

The proof of shuffle could be done "offline", though intermediate values would have to be kept private (I think).

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.

3.3 Timing results

See https://docs.google.com/spreadsheets/d/1Sny1xXxU9vjPnqo2K1QPeBHQwPVWhJOHdlXocMimt88VMI
spreadsheets for graphs of timing results (work in progress).

4 Bibliography

References

[1] B. Terelius and D. Wikström. *Proofs of restricted shuffles*, In D. J. Bernstein and T. Lange, editors, AFRICACRYPT'10, 3rd International Con-

ference on Cryptology inAfrica, LNCS 6055, pages 100–113, Stellenbosch, South Africa, 2010.

[2] 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.

[3] 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`.

[4] D. Wikström. *Verificatum Mix-Net*, `https://github.com/verificatum/verificatum-vmn`.

[5] 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`

[6] 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.

[7] *OpenCHVote* E-Voting Group, Institute for Cybersecurity and Engineering, Bern University of Applied Sciences, `https://gitlab.com/openchvote/cryptographic-protocol`

[8] Thomas Haines, *A Description and Proof of a Generalised and Optimised Variant of Wikström's Mixnet*, arXiv:1901.08371v1 [cs.CR], 24 Jan 2019.

[9] 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`.

[10] John Caron, Dan Wallach, *ElectionGuard Kotlin library*, `https://github.com/votingworks/electionguard-kotlin-multiplatform`.

5 Appendix

5.1 Alternative Calculation of B and B'

The calculation of B (use e instead of e' here for notational simplicity):

$$
\begin{aligned}
B_0 &= g^{b_0} h_0^{e_0} \\
B_i &= g^{b_i} (B_{i-1})^{e_i}, \ i = 1..n
\end{aligned}
$$

can be done to only use accelerated exponents . Expand the series:

$$
\begin{aligned}
B_0 &= g^{b_0} h_0^{e_0} \\
B_1 &= g^{b_1} (B_0)^{e_1} = g^{b_1 + b_0 e_1} \cdot h_0^{e_0 \cdot e_1} \\
B_2 &= g^{b_2} (B_1)^{e_2} = g^{b_2 + (b_1 + b_0 e_1) e_2} \cdot h_0^{e_0 \cdot e_1 \cdot e_2} \\
&... \\
B_i &= g^{b_i} (B_{i-1})^{e_i} = g^{gexps_i} \cdot h_0^{hexps_i}
\end{aligned}
$$

where

$$
\begin{aligned}
gexps_i &= b_i + (gexps_{i-1}) \cdot e_i \\
hexps_i &= \prod_{j=1}^{i} e_j
\end{aligned}
$$

Then each row has 2 accelerated exponentiations.
Similarly for B':

$$
\begin{aligned}
B'_1 &= g^{\beta_1} h_0^{eps_1} \\
B'_i &= g^{\beta_i} (B_{i-1})^{eps_i}, \ i = 2..n
\end{aligned}
$$

can be done with only accelerated exponentiations:

$$
\begin{aligned}
B'_1 &= g^{\beta_1} h_0^{eps_1} \\
B'_i &= g^{beta_i} (B_{i-1})^{eps_i} = g^{gpexps_i} \cdot h_0^{hpexps_i}
\end{aligned}
$$

where

$$gpexps_i = \beta_i + (gexps_{i-1}) \cdot eps_i$$
$$hpexps_i = hexps_{i-1} \cdot eps_i$$

5.1.1 Operation Count

	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

Table 3: Operation count