egk mixnet maths

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

We use the the ElectionGuard Kotlin library [7] for all the cryptography primitives. This library closely follows the ElectionGuard 2.0 specification [1].

Some of the prototype code in egk-mixlib is a port of code found in the OpenCHVote repository [8], and the appropriate license has been added. Please use any of this work in any way consistent with that.

The math here mostly recapitulates the work of Rolf Haenni et. al. [2], [3] in explaining the Terelius / Wikström mixnet algorithm [4], [5].

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

1. The ElectionGuard Group

- $\mathbb{Z} = \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$ is the set of integers.
- $\mathbb{Z}_n = \{0, 1, 2, \ldots, n-1\}$ 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, $\mathbb{Z}_p^* = \{1, 2, 3, \ldots, p-1\}$
- $\mathbb{Z}_p^r$ 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.

We use the ElectionGuard Kotlin library [7] and ElectionGuard 2.0 specification [1] for all 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.

2. 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 $j = \psi^{-1}(i)$.

A *permutation* ψ has a *permutation matrix* B_ψ , where b_{ij} = 1 if $\psi(i)$ = j, otherwise 0.

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

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

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

3. Pedersen Commitments

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

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

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

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

4. Proof of permutation (TW offline?)

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 c is a commitment to a permutation matrix.

5. ElGamal Encryption and Reencryption

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

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

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

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

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

$$\begin{aligned} & (2f) \\ \prod_{j=1}^{n} ReEncr(m_j, r_j)^{k_j} &= \prod_{j=1}^{n} Encr(0, r_j * k_j) * \prod_{j=1}^{n} Encr(m_j * k_j, \xi_j * k_j) \\ &= Encr(0, \sum_{j=1}^{n} (r_j * k_j)) * \prod_{j=1}^{n} Encr(m_j, \xi_j)^{k_j} \end{aligned}$$

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

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

6. TW Algorithm, proof of equal exponents (online?)

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.

7. Multitext mixing

Most of the literature assumes that each row to be mixed consists of a single ElGamalCiphertext. In our application we need the possibility that each row consists of ***width*** number of ElGamalCiphertexts. So for each row, we use:

```
    data class MultiText(val ciphertexts: List<ElGamalCiphertext>)
```

and it is the rows that are permuted, not the flat list of ciphertexts.

The changes to the standard algorithms needed for this are modest:

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

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

Here we just flatten the list of lists of ciphertexts for $\mathbf{e}, \mathbf{pe}$. This is used in both the proof construction and the proof verification.

2. In eq (2), we have $sumru = \sum_{j=1}^{n} (pr_j * pu_j)$. We need to modify this to

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

since each e_j has *width* ciphertexts, and all have pu_j applied.

Further research is needed to see if the restriction that all rows must have the same width could be relaxed, and if different pu_j could be used for different ciphertexts within a row.

8. Timings (preliminary)

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

operation counts

	proof	verify
regular exps	4*nrows + 2 * N	4 * nrows + 4 * N + 6
accelerated exps	3*nrows + 2 * N + 6	8

wallclock time (single threaded, fast CPU)

```
             shuffle: took 15910 msecs = 1.591 msecs/text (10000 texts) = 15910 msecs/shuffle for 1 shuffles
        shuffleProof: took 48261 msecs = 4.826 msecs/text (10000 texts) = 48261 msecs/shuffle for 1 shuffles
   checkShuffleProof: took 94875 msecs = 9.487 msecs/text (10000 texts) = 94875 msecs/shuffle for 1 shuffles
               total: took 159046 msecs = 15.90 msecs/text (10000 texts) = 159046 msecs/shuffle for 1 shuffles
```

Total time is 160 secs = 2.5 minutes to shuffle 100 ballots of 100 ciphertexts. = 56.5 * nrows + 15.24 * N msecs.

Not counting i/o or serialization.

Break into batches of 100 ballots each and do in parallel. Each batch would have 2 * 100 * 100 texts (input and shuffled). A ciphertext = 1K bytes. so 20 Mbytes. The proofs are small (4 * nrows ElementQ = 4 * 100 * 32 = 12K bytes).

Verificatum vs egk-mixnet

Vmn in pure Java mode, using BigInteger, shuffle twice:

100 rows, width = 34.

```
time ./scripts/runMixnetWorkflow.sh
...
real    1m30.294s
user    20m41.260s
sys 2m17.177s
```

TODO: is vmn parellel, what is user time?

egk-mixnet, shuffle trice:

```
testShuffleVerifyJson: nrows=100, width=34 N=3400
  shuffle1 took 29750
  verify1 took 33980
  shuffle2 took 24460
  verify1 took 33712
after 2 shuffles: 121903 msecs, N=3400 perN=35 msecs
```

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