# egk-ec-mixnet maths

John Caron *

6/03/2024

This is the mathematical description of the code in the egk-ec-mixnet library [12], which is an extraction of the Terelius / Wikström (TW) mixnet algorithm [1] [2] implementation from the Verificatum library [4], for use in conjunction with the electionguard-kotlin library [13].

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

The verification equations are well documented in [3]. The proof equations are reverse engineered from reading the Verificatum code, and are apparently not otherwise documented, in particular not in [3], although likely they are implied in [1] but using different notation.

The proof equations are implemented in the egk-ec-mixnet library class *ShuffleProver* and the verifier equations in *ShuffleVerifier*. The ShuffleVerifier has also been tested against the proofs output by Verificatum itself, leading to some confidence that these equations capture the TW algorithm as implemented in Verificatum.

Instead of providing pseudo-code, the Kotlin egk-ec-mixnet library [12] code is the implementation of the math described here. The code implementing the algorithm is separate from the workflow and the serialization code, and can act as a reference and comparison for ports to other languages and to other applications needing a mixnet.

Sections 1 and 2 describe the mathematics of the (TW) mixnet algorithm. Sections 3 and 4 provide performance comparisons between the Verificatum and egk-ec-mixnet libraries.

*jcaron1129@gmail.com

Thanks to Vanessa Teague for many corrections and clarifications in reviewing this paper, to Aresh Mirzaei for his review of the code and his helpful discussions, and to Michael Naehrig for his help with Elliptic Curves. I am grateful to Ben Adida for the opportunity to work with VotingWorks, to Josh Benolah and Michael Naehrig for all their work on ElectionGuard, and to Dan Wallach for his Kotlin cryptography implementation and sound advice.

Remaining errors in the math and the code are my own. Feedback and corrections would be very much appreciated.

# Contents

# 1 Definitions

The ElectionGuard 2.0 specification [9], and the ElectionGuard Kotlin library [13] are used for the cryptography primitives, in particular the parameters for $\mathbb{G}$, the variant of ElGamal exponential encryption, and the specifications for hashing with HMAC-SHA-256.

## 1.1 Mathematical Groups

The ElectionGuard Kotlin library implements both the ElectionGuard Integer Group and the P-256 Elliptic Curve Group. Everything in the math here and in the mixnet library itself works transparently with either Group, which is to say the details are hidden in the implementing code.

The mathematical Group in the rest of this document is denoted $\mathbb{G}$, which we describe specifically below for the two different cases. ElGamal ciphertexts are pairs of elements of $\mathbb{G}$, while other parts of the protocol require single elements of $\mathbb{G}$.

In both concrete groups, we also use

- $\mathbb{Z}_q = \{0, 1, 2, 3, ..., q-1\}$ the additive group modulo the prime q.

### 1.1.1 The ElectionGuard Integer Group

This is the ElectionGuard Integer Group $\mathbb{G} = \mathbb{Z}_p^r$ as described in [9]. Internally, elements of $\mathbb{G} = \mathbb{Z}_p^r$ use one 512 byte (4096 bit) integer, and $\mathbb{Z}_q$ uses one 32 byte (256 bit) integer. Currently the implementation uses pure JVM code.

### 1.1.2 The P-256 Elliptical Curve Group

In the elliptic curve case, $\mathbb{G}$ is the well-known *NIST Curve P-256*. The implementation is derived from the Verificatum VCR and VMN libraries, and will use the native Verificatum VECJ library and the GMP library directly when those are installed and enabled, for most of the performance-sensitive primitives. Otherwise the implementation uses pure JVM code.

The characteristics of the P-256 Group include:

- The group parameters are from the NIST standard [18].

- Internally, elements of $\mathbb{G}$ use two 32 byte (256 bit) integers to represent a point on the elliptic curve. Serialization uses point compression that reduces the byte count to 33 bytes. $\mathbb{Z}_q$ uses one 32 byte integer.

### 1.2 ElGamal Encryption and Reencryption

A variant of exponential ElGamal is used for encryption and reencryption (also see [11] and Appendix A), using the group generator g, and a public key K:

$$Encr(m, \xi) = (g^{\xi}, K^{m+\xi}) \tag{1}$$

$$Encr(0, \xi') = (g^{\xi'}, K^{\xi'}) \tag{2}$$

If $c = Encr(m, \xi)$ then

$$ReEncr(c, r) = Encr(0, r) * c = (g^{\xi+r}, K^{m+\xi+r}) \tag{3}$$

### 1.3 Permutations

A *permutation* is a bijective map $\psi : 1..n \to 1..n$. If $\vec{x}$ is a vector, $\psi(\vec{x})$ is the permutation of its elements, which we denote $\vec{x'}$, so that $x_i = x'_j$, where $i = \psi(j)$ and $j = \psi^{-1}(i)$. If M is a matrix, $\psi(M)$ is the permutation of its row vectors, to form a new matrix $M'$ of the same shape.

A permutation $\psi$ has a *permutation matrix* $B_\psi = (b_{ij})$ , where $b_{ij} = 1$ if $\psi(i) =$ j, otherwise 0, and where $\psi(\vec{x}) = B\vec{x}$ (matrix multiply).

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

$$\forall j \in 1..n : \sum_{i=1}^{n} b_{ij} = 1 \tag{4}$$

$$\prod_{i=1}^{n} \sum_{j=1}^{n} b_{ij} x_i = \prod_{i=1}^{n} x_i \tag{5}$$

This is Theorem 1 of [1], and Section 5.5.1 (first paragraph) of [5].

### 1.3.1 Permutation Commitments

For a vector $\vec{m} = (m_1..m_n) \in \mathbb{Z}_q$, the *Pedersen commitment* to $\vec{m}$ is

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

where $\vec{h}$ are random generators of $\mathbb{G}$ and $cr \in \mathbb{Z}_q$ is a randomization nonce.

If $\vec{b}_j$ is the $j^{th}$ column vector of a permutation matrix $B_\psi$, then the *permutation commitment to* $\psi$ is defined as the vector of commitments to its columns:

$$Commit(\psi, \vec{cr}) = (Commit(\vec{b}_1, cr_1), ..Commit(\vec{b}_n, cr_n)) \tag{6}$$

$$Commit(\vec{b}_j, cr_j) = g^{cr_j} * \prod_{i=1}^{n} h_i^{b_{ij}} = g^{cr_j} h_i, \text{ for } i = \psi^{-1}(j) \tag{7}$$

$$Commit(\psi, \vec{cr}) = \{g^{cr_j} h_{\psi^{-1}(j)}\}, \text{ j=1..n} \tag{8}$$

See section 5.2 of [5].

# 2 Terelius / Wikström 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 $\vec{w}_j, j = 1..width$
- $R$ = matrix of reencryption nonces, where $R_{ij}$ is the reencryption nonce for $W_{ij}$.
- $W'$ is the shuffle of $W$ = matrix of shuffled and reencrypted ciphertexts. Its entries, row vectors and column vectors are $w'_{i,j}, \vec{w'}_i, \vec{w'}_j$ respectively
- $\psi$ = permutation function
- $\psi^{-1}$ = inverse permutation function
- $\vec{h}$ = random generators of $\mathbb{G}$
- $h_0 = \vec{h}_1$
- $\vec{e}$ = batching vector.
- $\vec{e'} = \psi(\vec{e})$ = permuted batching vector.
- $v \in \mathbb{Z}_q$ = challenge.

We use one-based array indexing here for notational simplicity. The code itself uses zero based indexing.

## 2.2 Non-interactive Calculations

In the non-interactive version of the proof, the generators $\vec{h}$, batching vector $\vec{e}$, and challenge $v$ must be generated with a deterministic calculation that both the Prover and the Verifier can independently make, to prevent them from being carefully chosen by a corrupt Prover to subvert the proof.

### 2.2.1 Generators

The ProofOfShuffle requires random generators of $\mathbb{G}$, which are created by Verificatum [4] in a way that is highly idiomatic, and could only be created outside of the Verificatum library with difficulty. We use a simpler implementation, but follow the Verificatum algorithm described in section 6.8 of [3] and Section VIII of [17], modified in the elliptic curve case by using RFC9830.

A PRG is created based on the standard ElectionGuard SHA256 hashing, seeded with the *parameterBaseHash()* which includes all the parameters of $\mathbb{G}$, and an arbitrary string *mixName*. The PRG generates pseudo-random byte arrays of of arbitrary length by hashing the seed with incrementing integer values. See *org.cryptobiotic.mixnet.Generators.kt* in [12] for details.

Important parameters are $n_p$, the number of bytes used by the prime p, and $n_r$, the "auxiliary security parameter", currently 16 bytes (128 bits).

Note that bit lengths must be a multiple of 8 in the current implementation.

For the Integer Group:

1. Use PRG to create n positive integers $t_i$ of byte length $n_p + n_r$.
2. Calculate the generators as $h_i = t_i^{(p-1)/q} \bmod p$.

For the Elliptic Group:

1. Use PRG to create a positive integer of byte length $n_p$.
2. Feed that into the algorithm from RFC9380, section 5.3, to get uniform hashing with additional byte length $n_r = 16$ (128 bits). Then reduce by modulo p to give an integer $x$ of byte length $n_p = 32$.
3. Form $y = f(x)$ where f is the elliptic curve function.
4. Test if y is a quadratic residue, i.e. if $y^{((p-1)/2)} \bmod p = 1$. If so, add the elliptic curve point (x, sqrt(y)) to the list of generators.
5. Repeat steps 1-4 until n generators are found.

See *org.cryptobiotic.mixnet.getGenerators()* in [12] for the exact algorithm.

### 2.2.2 BatchingVector

- Create a seed by doing a recursive hash on all of the following elements:
  - the electionguard *parameterBaseHash()* on the group parameters
  - the generators $\vec{h}$
  - the permutation commitments $\vec{u}$
  - the election public key $K$
  - all the ciphertexts from $W$
  - all the ciphertexts from $W'$
- Use the egk library *Nonces* class to create a vector of nonces from the seed.
- Set $\vec{e}$, where $e_i = nonce_i$ (so $n_e = 256$)

In a recursive hash, each element's hash consists of the elements index, and a byte representation of the element. If the element is a collection, each of its components are hashed separately and fed into the element's hash. Then all the element hashes are fed into the final hash. We are using the SHA-256 MessageDigest for the hashing. See *org.cryptobiotic.maths.recursiveSHA256()* in [12]

See *org.cryptobiotic.mixnet.makeBatchingVector()* in [12] for the exact algorithm.

### 2.2.3 Challenge

- Create a recursive hash (as above) from the following fields from section 2.4.2 and 2.4.3:
  - the seed that was made in makeBatchingVector() above
  - $A'$, $B$, $B'$, $C'$, $D'$, $F'$
- Then challenge = hash.digest().

This corresponds to $\tau^{pos}$ in Step 4 of Algorithm 19 of [3] and to the line "$y \leftarrow \mathsf{GetChallenge}(y, t)$" of Algorithm 8.45 in [5]. (Note that the actual value will be different in those implementations.)

See section 2.4.2 and 2.4.3 for definitions of $A'$, $B$, $B'$, $C'$, $D'$, $F'$. See *org.cryptobiotic.mixnet.makeChallenge()* in [12] for implementation.

## 2.3 Shuffle

Let R be a matrix of (n x width) randomly chosen reencryption nonces. For each ciphertext in W:

$$\begin{aligned} Reencrypt(w_{ij}) &= Reencrypt(w_{ij}, r_{ij}) = Encr(0, r_{ij}) * w_{ij} \\ &= Reencrypt(W, R) \end{aligned}$$

Then $W'$ is the permutation of the rows of $Reencrypt(W, R)$:

$$W' = \psi(Reencrypt(W, R))$$

The shuffle step chooses R and $\psi$, so Shuffle(W) $\rightarrow (R, \psi, W')$

## 2.4 Proof

This is a reverse engineering of the code in VMN. It corresponds to section 5.5.1 and Algorithm 8.45 of [5] and Algorithm 19 of [3]. The notation generally reflects that of [3].

### 2.4.1 Commitment to permutation

Choose a vector of random permutation nonces $\vec{cr}$. Form public permutation commitments $\vec{u}$, as described in Section 1.3.1:

$$u_j = g^{cr_j} h_i \quad \text{for } i = \psi^{-1}(j), \text{ j=1..n} \tag{9}$$

### 2.4.2 Commitment to shuffle

Choose vectors of $n$ random nonces $\vec{b}, \vec{\beta}, e\vec{p}s$ and random nonces $\alpha, \gamma, \delta$, all $\in \mathbb{Z}_q$.
Form the following values $\in \mathbb{G}$:

$$A' = g^{\alpha} \prod_{i=1}^{n} h_i^{eps_i} \tag{10}$$

$$B_1 = g^{b_1} h_0^{e'_1}, \ B_i = g^{b_i} (B_{i-1})^{e'_i}, \ i = 2..n \tag{11}$$

$$B'_1 = g^{\beta_1} h_0^{eps_1}, \ B'_i = g^{\beta_i} (B_{i-1})^{eps_i}, \ i = 2..n \tag{12}$$

$$C' = g^{\gamma} \tag{13}$$

$$D' = g^{\delta} \tag{14}$$

This corresponds to Algorithm 8.45 of [5]. Also see Appendix B for a variation on computing $B$ and $B'$.

### 2.4.3 Commitment to exponents

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

$$F'_j = Encr(0, -\phi_j) \cdot \prod_{i=1}^{n} (w'_{i,j})^{eps_i} \text{ for j=1..width} \tag{15}$$

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 $e\vec{p}s$. We can use any of the following notations to indicate this:

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

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

### 2.4.4 Reply to challenge

A challenge v $\in \mathbb{Z}_q$ is created as in 2.2.3, and the following values $\in \mathbb{Z}_q$ are made as reply:

$$k_A = v \cdot < \vec{cr}, \vec{e} > + \alpha \tag{16}$$

$$\vec{k_B} = v \cdot \vec{b} + \vec{\beta} \tag{17}$$

$$k_C = v \cdot \sum_{i=1}^{n} cr_i + \gamma \tag{18}$$

$$k_D = v \cdot d + \delta \tag{19}$$

$$\vec{k_E} = v \cdot \vec{e'} + e\vec{p}s \tag{20}$$

and, with $\vec{R}_j$ = jth column of reencryption nonces R:

$$k_{F,j} = v \cdot < \vec{R}_j, \vec{e'} > + \phi_j \text{ for j=1..width} \tag{21}$$

where $<,>$ is the inner product of two vectors, and $\cdot$ is scalar multiply.

### 2.4.5 The ProofOfShuffle Data Structure

ShuffleProver$(R, \psi, W') \rightarrow$ ProofOfShuffle

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

    // Commitment of the Fiat-Shamir proof.
    val Ap: ElementModP,
    val B: VectorP,
    val Bp: VectorP,
    val Cp: ElementModP,
    val Dp: ElementModP,
    val Fp: VectorCiphertext, // size 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, // size width
)
```

## 2.5 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 *commitment to exponents* section 2.4.3 above.

The Verifier is provided with:

- $W$ = rows of ciphertexts (n x width)
- $W'$ = shuffled and reencrypted rows of ciphertexts (n x width)
- the ProofOfShuffle

The $\vec{h}$ (generators), $\vec{e}$ (batching vector), and challenge $v$ are deterministically recalculated from the algorithms described in section 2.2.

The following values $\in \mathbb{G}$ are computed:

$$A = \prod_{i=1}^{n} u_i^{e_i} \tag{22}$$

$$C = (\prod_{i=1}^{n} u_i)/(\prod_{i=1}^{n} h_i) \tag{23}$$

$$D = B_n \cdot h_0^{\prod_{i=1}^{n} e_i} \tag{24}$$

and

$$F_j = \prod_{i=1}^{n} (w_{i,j})^{e_i} \text{ for j=1..width} \tag{25}$$

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

$$A^v \cdot A' = g^{k_A} \prod_{i=1}^{n} h_i^{k_{E,i}} \tag{26}$$

$$B_i^v \cdot B_i' = g^{k_{B,i}} (B_{i-1})^{k_{E,i}}, \ where \ B_0 = h_0, \ i = 1..n \tag{27}$$

$$C^v \cdot C' = g^{k_C} \tag{28}$$

$$D^v \cdot D' = g^{k_D} \tag{29}$$

and

$$F_j^v F_j' = Encr(0, -k_{F,j}) \prod_{i=1}^{n} (w_{i,j}')^{k_{E,i}} \text{ for j=1..width} \tag{30}$$

# 3 Performance for the Integer Group

This section presents results for the ElectionGuard Integer Group. See below for the Elliptic Group.

Environments used for measuring times:

**Workstation**

- Ubuntu 22.04.3
- HP Z440 Workstation, Intel Xeon CPU E5-1650 v3 @ 3.50GHz
- 6-cores, two threads per core.

**Server**

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

**Laptop**

- Windows 10 Pro
- Dell Precision M3800, Intel i7-4712HQ CPU @ 2.30GHz
- 4-cores, two threads per core.

## 3.1 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 + n - 1 | 4N + 4n + 1 |
| accelerated exps | 2N | 6n + 2width + 4 | 2n + 2width + 6 |

Table 1: Exponent operation count

### 3.2 Regular vs accelerated exponentiation time

When the same base is used many times for exponentiation, it can be optimized with Pereira's "pow-radix" precomputation and Montgomery forms. See [11], [14] for more details. Here we measure the performance difference between regular and accelerated exponentiation, after JIT warmup:

**Workstation**

```
acc took 12551 msec for 20000 = 0.62755 msec per acc
exp took 40998 msec for 20000 = 2.0499 msec per exp
exp/acc took 3.266
```

**Server**

```
acc took 15288 msec for 20000 = 0.7644 msec per acc
exp took 46018 msec for 20000 = 2.3009 msec per exp
exp/acc = 3.010
```

**Laptop**

```
acc took 16910 msec for 20000 = 0.8455 msec per acc
exp took 55654 msec for 20000 = 2.7827 msec per exp
exp/acc took 3.291
```

From which we conclude that accelerated exponentiation is about 3 times faster.

We can estimate the performance gain of this acceleration using the operation counts from above, and the factor of 3 for the acceleration speedup. Then

$$speedup = noacc/withacc = totalcount/(expcount + (acccount/3)$$

When width = 34, nrows greater than 50, ShuffleAndProof gets around a 50 percent speedup, while Verify gets only a few percent.

### 3.3 Timing results for Integer Group

All results shown are for the workstation (6 cores, 12 threads), but results are similar for the other test environments.

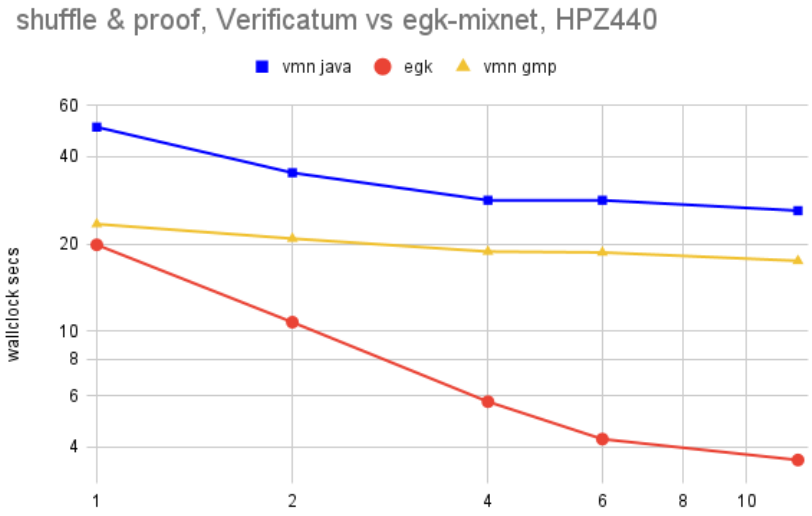


Figure 1: ShuffleAndProof on 100 ballots of width 34

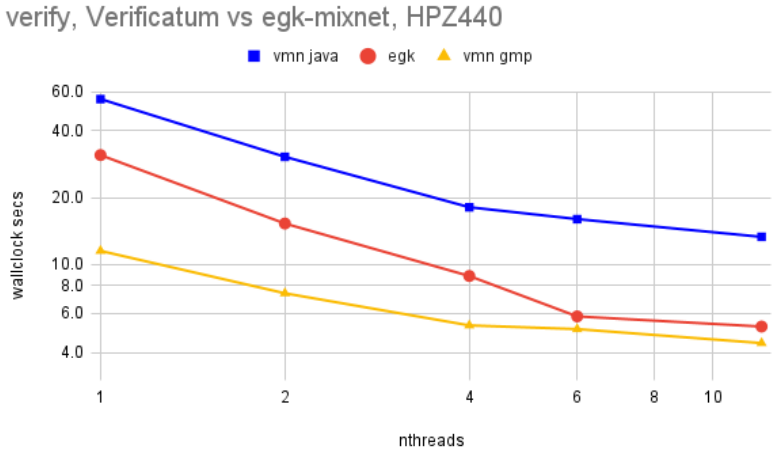


Figure 2: Verify on 100 ballots of width 34

This seemed acceptable until I tested on larger numbers of ballots:

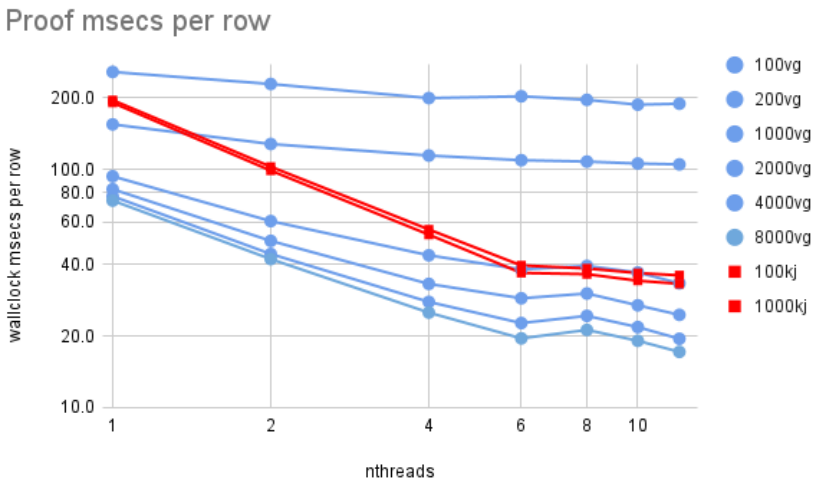


Figure 3: ShuffleAndProof for differing number of ballots

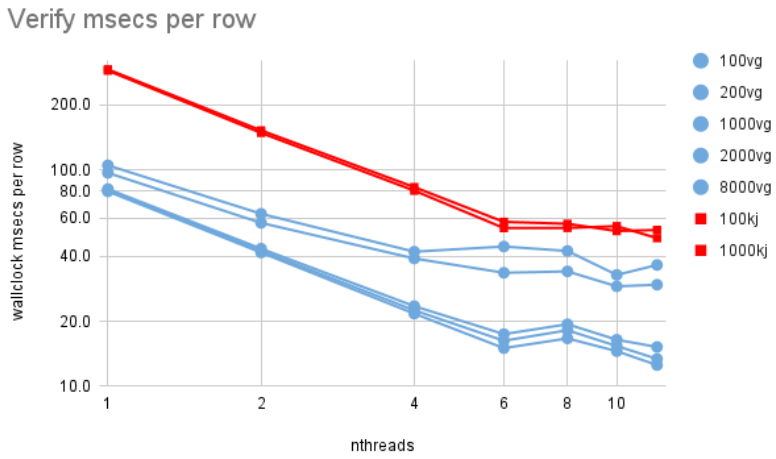


Figure 4: Verify for differing number of ballots

| | |
|---|---|
| 100vg | 100 ballots, with VMN and GMP |
| 200vg | 200 ballots, with VMN and GMP |
| 1000vg | 1000 ballots, with VMN and GMP |
| 2000vg | 2000 ballots, with VMN and GMP |
| 4000vg | 4000 ballots, with VMN and GMP |
| 8000vg | 8000 ballots, with VMN and GMP |
| 100kj | 100 ballots, with egk java |
| 1000kj | 1000 ballots, with egk java |

Table 2: Figures 3 and 4 captions

VMN with GMP (vg) does much better than EGK-Java (kj) for Verify (3-4x), especially when the number of ballots is larger than 100. Vg Proof does much better when the number of ballots is over 500-1000 (1.5-2.5x), and it keeps getting better the more ballots there are, up to at least 8000. In contrast, vg Verify reaches maximum speedup by 1000, and kj is completely independent of the number of ballots for both Proof and Verify.

### 3.3.1 GMP and simultaneous exponentiation

The operation count from table 1 is dominated by the 2N term for Proof and the 4N term for Verify, where N = total number of ciphertexts = nballots x width. This comes from eq 15 (Proof) and eq 25 and 30 (Verify), when calculating products of the column vectors of W or W' raised to an exponent. Egk-Java does these exponentiations separately with the Java BigInteger library function *modPow*. Its performance is surprisingly good for pure Java (with some intrinsic inline code).

VMN with GMP has carefully crafted C code (spowm.c in the GMPEE library) to optimize this operation, using a variant of the simultaneous exponentiation algorithm, see 14.88 of [14]. If t is the number of bits in the exponent (256 in our case), then w can be chosen to minimize the number of multiplications needed, where w is the number of terms to be simultaneously exponentiated. For t=256, 7 is the optimum value for w. The standard algorithm then is called N/w times, and each call takes $(2^w + 2t)$. The cost per term is $(2^w + 2t)/w$. For t=256 and w=7, this is 91 multiply operations. This algorithm is what VMN-Java uses.

VMN with GMP uses a variation of this algorithm, which divides N into b batches, then further divides each batch into w slices. This allows the squarings to be done once per batch, at the cost of storing the $b/w$ exponentiation tables in memory at the same time. This reduces the cost

per N to $(2^w + t)/w + t/b$, with a memory cost of $2^w * b/7 * p$ bytes, where p is the size of the base terms in bytes, in our case = 512. We chose b=84, so that the extra memory is less than a megabyte, and then the cost per term is 58 multiplies, which is only 2 more than the minimum acheivable with this algorithm.

Implementation in Kotlin executes about the same as using the built-in BigInteger modPow function, which agrees with our measurements that one modPow operation costs about the same as around 59 multiplies. See *org.cryptobiotic.maths.VmnProdPowW.kt* in [13] for the relevant code.

There are a number of libraries, (eg [15]), that allow GMP to be called by Java using JNA (verificatum-vmgj uses JNI which JNA is based on). However, the overhead of using JNI to call into C code destroys most of the speedup. In order to be competitive with VMN-GMP, we need to replicate the verificatum-gmpmee algorithms in C. One can then do a large amount of computation for a single call into C code, so the overhead is minimal.

To test the performance of this option, a fork of the repo [13] was used that called the GMP library directly, not using verificatum-vmgj. To call GMP I used the new FFM feature in Java [16] to call into my own C code, which ports some parts of verificatum-gmpmee. This was mostly a desire to try out FFM, which replaces JNI with a much safer and less brittle mechanism. The downside to this decision to use FFM is that one needs Java 21, and there is always delay in adopting new versions of the JVM. However, Java 21 is an LTS (a stable release with long-term support), and our intention is to upgrade to the latest LTS (every two years) as a matter of policy.

This fork [13] is called egk-mixnet with GMP (kg), and is used in the timing for the Integer group in Figures 5 and 6 below. It predates the development of the Elliptic Curve Group in egk-ec-mixnet [12]. The option to use GMP for the Integer group is not yet present in egk-ec-mixnet, and will be added later.

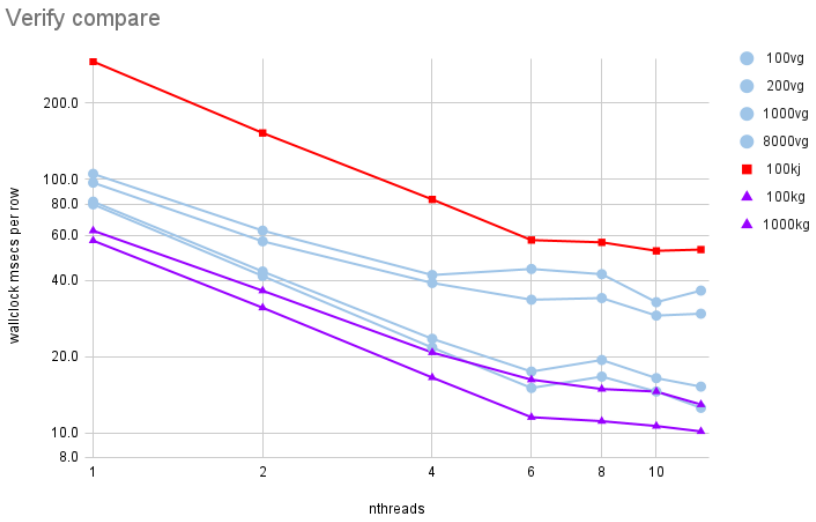


Figure 6: Verify compare

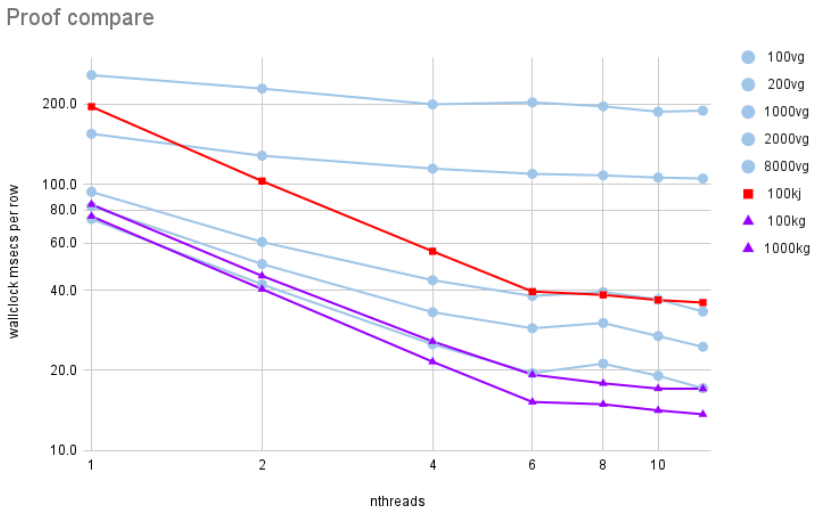


Figure 5: ShuffleAndProof compare

For Proof, at high numbers of ballots, kg and vg are identical when nthreads = 1, then kg has a modest advantage of 30-40% at higher thread

count. At smaller ballot counts, for example 1000, kg starts at 20% better for nthreads = 1, and rises to 2.5x better for higher threads. For Verify and nballots = 1000 or more, kg is 20-50% better than vg, and increasingly better for smaller nballots.

### 3.3.2 Parallelization

We use Kotlin coroutines to parallelize all operations that can be calculated independently on row or column vectors or matrices. This parallelization is done at a high level within the algorithm code, so as to control the amount of work put into each thread. Verificatum appears to place its parallelization code lower down in the stack, in individual vector operations.

In Figures 5 and 6, kg shows only modest dependence on the number of ballots, and stops improving by 1000 ballots, while vg has a much larger dependence on the number of ballots, especially for Proof.

Both variants of ekg-mixnet show good parallelization speedup up to the number of cores, after which presumably it starts competing with the system and other background processes. Vmn also shows this behavior, esp when the ballot count is over 100.

Its likely that the main difference in performance between vg and kg is better parallelization in kg. Vg Proof in particular shows poor parallelization, such that even the egk-ec-mixnet Java version is competitive with it at higher thread counts and lower nballots.

The percent of unparallel code (lower is better) can be roughly estimated from the formula $time = (unparallel) + (parallel)/nthreads$, running with various values of nthreads, and solving the linear equations. Doing so gives:

| | proof | verify |
|---|---|---|
| 100g | 62 % | 25 % |
| 200g | 54 % | 18 % |
| 1000g | 24 % | 4.7 % |
| 2000g | 18 % | 3.5 % |
| 8000g | 10 % | 2.2 % |
| 100k | 3.6 % | 3.1 % |
| 1000k | 2.6 % | 2.1 % |
| 2000k | 2.7 % | 2.1 % |
| 100kg | 6.3 % | 9.2 % |
| 1000kg | 3.4 % | 3.4 % |
| 4000kg | 3.2 % | 3.0 % |

Table 3: Percent unparallel code

These numbers support some of our conclusions above: that VMN-GMP has poor parallelization at low values of nballots, especially in the Proof; that larger numbers of ballots get better parallelization; and that egk-mixnet (both Java and GMP) is mostly "as good as it will get" for nballots greater than 1000. For large number of ballots and low thread count, the two libraries have roughly equivalent performance.

# 4 Performance for the Elliptic Group P-256

This section presents performance results using the Elliptic Group P-256, comparing egk-ec-mixnet with Verificatum VMN. In both cases the underlying multiprecision math calculations are done by GMP, and called from the JVM with Verificatum-vecj using JNI.

Testing is all done on the "server machine", with 24 cores and 48 threads. The number of threads varies from 1 and 48, and the number of ballots (aka rows) varies from 100 to 4000.

The ballots are uniform with a width of 34 ciphertexts. In Figures 7 and 8, "e100" for example means egk-ec-mixnet with 100 rows, and "v1000" is VMN with 1000 rows. Time is wallclock time. Graphs are log-log.

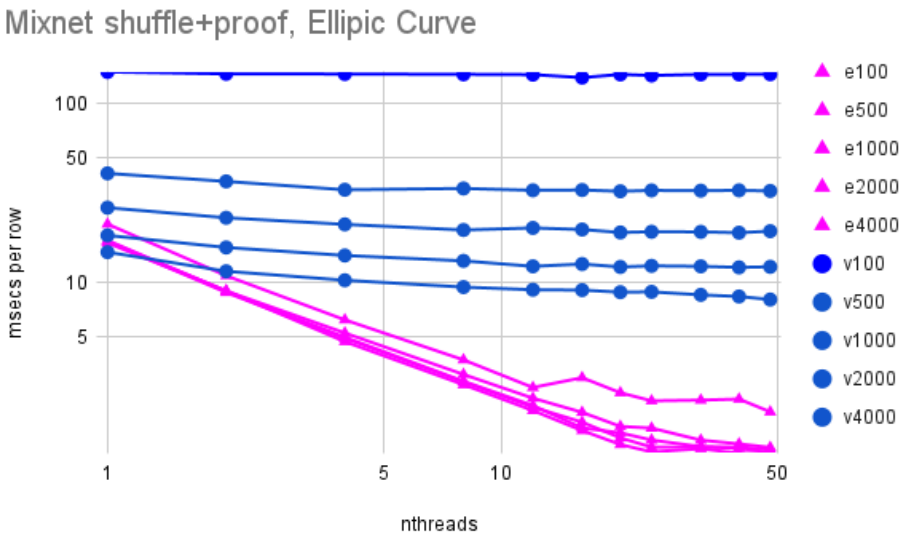


Figure 7: ShuffleAndProof, Elliptic Curves

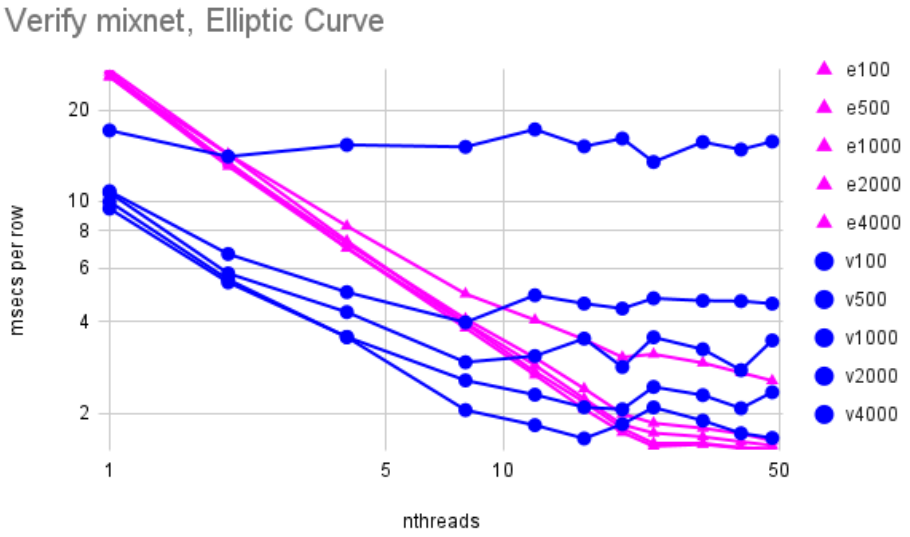


Figure 8: Verify, Elliptic Curves

VMN shuffle and proof shows little to no parallelization, but gets better performance at higher number of rows, as in the integer case. egk-ec-mixnet has good parallelization down to around the number of cores (24), and very little dependence on the number of rows.

VMN verify has poor parallelization at small number of ballots, but quite good performance, especially at higher number of rows, along with somewhat mediocre parallelization. Verify in egk-ec-mixnet starts off with performance about half of VMN at low threads, but with good parallelization, catches up to VMN at higher threads, with very little dependence on the number of rows.

Given that both use GMP for most of the multiprecision calculations, the differences are likely due to the software architecture. The source of the performance advantage of VMN at small threads could be investigated.

# A ElGamal Encryption and Reencryption

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

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

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

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

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

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

(1.2g)

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

Let

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

then

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

# B Alternative Calculation of B and B'

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

$$B_1 = g^{b_1} h_0^{e_1}$$
$$B_i = g^{b_i} (B_{i-1})^{e_i},\ i = 2..n$$

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

$$B_1 = g^{b_1} (h_0)^{e_1}$$
$$B_2 = g^{b_2} (B_1)^{e_2} = g^{b_2 + b_1 e_2} \cdot h_0^{e_1 \cdot e_2}$$
$$B_3 = g^{b_3} (B_2)^{e_3} = g^{b_3 + (b_2 + b_1 e_2) e_3} \cdot h_0^{e_1 \cdot e_2 \cdot e_3}$$
$$...$$
$$B_i = g^{b_i} (B_{i-1})^{e_i} = g^{gexps_i} \cdot h_0^{hexps_i}$$

where

$$\begin{aligned} gexps_1 &= b_1 \\ gexps_i &= b_i + (gexps_{i-1}) \cdot e_i, \quad i > 1 \\ hexps_i &= \prod_{j=1}^{i} e_j \end{aligned}$$

Then each row has exactly 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:

$$B'_i = g^{gpexps_i} \cdot h_0^{hpexps_i}$$

where

$$\begin{aligned} gpexps_1 &= \beta_1 \\ gpexps_i &= \beta_i + (gexps_{i-1}) \cdot eps_i, \quad i > 1 \\ hpexps_1 &= eps_1 \\ hpexps_i &= hexps_{i-1} \cdot eps_i, \quad i > 1 \end{aligned}$$

The net result is that this shifts 2n operations from exp to acc in the proof.

## References


[1] 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.

[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`.

[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 OpenCHVote Internet Voting System* 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/JohnLCaron/egk-ec.

[11] Dan Wallach, *ElectionGuard Kotlin cryptography notes*, https://github.com/votingworks/electionguard-kotlin-multiplatform/blob/main/docs/CryptographyNotes.md

[12] John Caron, *egk-ec-mixnet library*, https://github.com/JohnLCaron/egk-ec-mixnet.

[13] John Caron, *egk-ec-mixnet library*, https://github.com/JohnLCaron/egk-mixnet.

[14] Menezes, A.J., Van Oorschot, P.C. and Vanstone, S.A, *Handbook of Applied Cryptography*, CRC Press, Boca Raton, FL,

[15] Gianluca Amato and Francesca Scozzari, *JGMP: Java bindings and wrappers for the GMP library*, https://www.softxjournal.com/article/S2352-7110(23)00124-3.

[16] JEP442, *Foreign Function And Memory API*, https://openjdk.org/jeps/442.

[17] T. Haines, S. J. Lewis, O. Pereira and V. Teague, *How not to prove your election outcome*, 2020 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 2020, pp. 644-660, doi: 10.1109/SP40000.2020.00048, https://ieeexplore.ieee.org/document/9152765.

[18] Mehmet Adalier and Antara Teknik, *Efficient and Secure Elliptic Curve Cryptography Implementation of Curve P-256*, 2015, https://csrc.nist.gov/csrc/media/events/workshop-on-elliptic-curve-cryptography-standards/documents/papers/session6-adalier-mehmet.pdf