

egk-ec-mixnet maths

John Caron *

5/20/2024

JC: Im wondering if you would be willing to be an author on this? I will certainly understand if you prefer not. Otherwise, I will just thank you as a reviewer.

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 [10].

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.

*jcaron1129@gmail.com

Section 3 provides performance comparisons between the Verificatum and egk-ec-mixnet libraries.

Contents

1	Definitions	4
1.1	Mathematical Groups	4
1.1.1	The ElectionGuard Integer Group	5
1.1.2	The P-256 Elliptical Curve Group	6
1.2	ElGamal Encryption and Reencryption	7
1.3	Permutations	7
1.3.1	Permutation Commitments	7
2	Terelius / Wikström Algorithm	9
2.1	Definitions	9
2.2	Non-interactive Calculations	9
2.2.1	Generators	10
2.2.2	BatchingVector	12
2.2.3	Challenge	15
2.3	Shuffle	17
2.4	Proof	18
2.4.1	Commitment to permutation	18
2.4.2	Commitment to shuffle	18
2.4.3	Commitment to exponents	19
2.4.4	Reply to challenge	20
2.4.5	The ProofOfShuffle Data Structure	20
2.5	Verification	21
3	Performance for the Integer Group	23
3.1	Operation counts	23
3.2	Regular vs accelerated exponentiation time	24
3.3	Timing results	25
3.3.1	GMP and simultaneous exponentiation	28
3.3.2	Parallelization	29
A	ElGamal Encryption and Reencryption	33
B	Alternative Calculation of B and B'	34
C	Implementation notes	36
	References	36

1 Definitions

The ElectionGuard 2.0 specification [9], and the ElectionGuard Kotlin library [10] 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

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

1.1.1 The ElectionGuard Integer Group

VT: It's not helpful that the literature tends to gloss over the difference between \mathbb{Z}_p^r and \mathbb{Z}_q . Here's the crucial fact: \mathbb{Z}_p^* is a cyclic group. That is, there's at least one element g (which is a number from 1 to $p - 1$) for which, if you list $L = \{g, g^2, g^3, \dots, g^{p-1}\}$ you get *all* the elements of \mathbb{Z}_p^* . Not all elements have this property: for some values of g , that list might repeat some elements and not mention others. But assume we've found a g that does indeed generate the whole set. (This is not supposed to be obvious, but it is a theorem, which frankly I can't remember the proof of, but I'm sure the Internet will know—look up “ \mathbb{Z}_p^* is a cyclic group”.) Now consider the following set:

$$X = \{g^r, g^{2r}, g^{3r}, \dots, g^{qr}\}$$

You'll notice it has q members, which must be distinct because they're different items in our list L —the last one is $g^{qr} = g^{p-1}$, so there are no repetitions of anything in L . You'll also notice that every element of X is an r -th residue. Here's the interesting bit: every element of \mathbb{Z}_p^r is in X , because if $y = x^r \bmod p$ is an r -th residue, we can write $x = g^i \bmod p$ for some i s.t. $1 \leq i \leq p - 1$ (because g is a generator of \mathbb{Z}_p^* —every element is a power of g). Then $y = g^{ir}$
 $= g^{(aq+b)r} \bmod p$ for some b s.t. $1 \leq b \leq q - 1$ (ordinary integer division)
 $= g^{arq} g^{br} \bmod p$
 $= g^{br}$ because $g^{rq} = g^{p-1} = 1$ by Fermat's little theorem.

Therefore y is in X .

So everything in \mathbb{Z}_p^r is in X and vice-versa. Hence the two sets are identical.

And finally, the multiplicative group on X is equivalent to the additive group of integers mod q - multiplying two elements of X , as $g^{ar} \times g^{br} \bmod p$, is just like adding $a + b \bmod q$. You can think about the operation happening in either domain (multiplication of elements in $X \bmod p$, or addition of elements of $\{1, 2, \dots, q - 1\} \bmod q$) and you always get the same answer.

So people tend to just equate (the multiplicative group of) the r -th residues of \mathbb{Z}_p^* (when $rq = p - 1$) with (the additive group of) integers mod q . They're not actually the *same* group, but they behave like the same group (the technical term is “isomorphic”).

One more fact: when g is a generator of \mathbb{Z}_p^* , it follows that g^r is a generator of the r -th residuosity group (which is sort of obvious if you look at the definition of X —we just proved that the powers of g^r give you exactly the members of the r -th residuosity group).

JC: Thanks thats really helpful. Ive been puzzling over what ElementModP and ElementModQ correspond to, and why it somehow works for both the Integer and the Elliptic groups. Now I see that they correspond to the isomorphism you describe, where ElementModQ is like the "log form" of an ElementModP, $y = g^i$, so $i = \log_g(y)$.

Or more generally we can say we have a non-invertible function E with domain Z_q (the finite field of order q , which are represented by ElementModQ), and range represented by ElementModP. For the Integer group, $E(Z_q) = g^i \bmod p$. And for Elliptic groups its the elliptic group function $E(Z_q) = \text{sqrt}(x^3 + ax + b) \bmod p$.

Thats why we can use the same ElementModQ in both cases, and its only the ElementModP that is different.

Does that sound right?

In any case Im going to simplify things by just referencing the defining documents.

The only thing needed below is \mathbb{G} and \mathbb{Z}_q .

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 [17].
- 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.

JC: Michael points out that we can save one byte in the serialization by always just choosing the smaller root.

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 :

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

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

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

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

1.3 Permutations

A *permutation* is a bijective map $\psi : 1..n \rightarrow 1..n$. 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 ψ has a *permutation matrix* $B_\psi = (b_{ij})$, where $b_{ij} = 1$ if $\psi(i) = j$, otherwise 0, and where $\psi(\vec{x}) = B_\psi \vec{x}$ (matrix multiply).

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

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

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

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_ψ , then the *permutation commitment to ψ* is defined as the vector of commitments to its columns:

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

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

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

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 ($n \times \text{width}$), 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 $\vec{w}_j, j = 1..\text{width}$
- 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
- ψ = permutation function
- ψ^{-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 [16], 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 arbitrary length by hashing the seed with incrementing integer values. See Appendix ? 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 length must be a multiple of 8 in the current implementation).

JC: TODO: Use more parameters in the seed? What should n_r , n_e , n_v (Algorithm 19 of [3]) be?

JC: AFAICT, VMN uses defaults of 100, 256, and 256 respectively. I am currently using 128, 256, 256. Note that n_r ("statistical distance") is added to the bit length of group.P when creating random elements of \mathbb{G} . But n_e , n_v are the bit lengths of the batching vectors and the challenge. Currently I cant use anything other than 256 for n_e , n_v .

VT: 128, 256, 256 sounds pretty sensible to me, especially as it matches (or exceeds) the verificatum default. As a general rule-of-thumb, 128 is fine for things where the attacker has to guess once (e.g. keys and El Gamal randomness), but 256 is needed for things where they just need to find a clash (typically hash outputs).

JC: It may be that VMN is using n_r whenever it creates a random value (note generators, batching vectors, and challenge are not random, but using a deterministic PRG). It does this by generating random bytes of $(\text{modulus.bitlength} + n_r)$ bits, then doing modular reduction. Im still tracking this down. If it is true, do we need to emulate that? Ive asked Michael Naehrig this question also.

VT: Hmmm this is tricky, and deserves some longer conversation. IIRC this was one of the things that doesn't "just work" when you go over to elliptic curves, though this is a bit outside my knowledge so I'm glad you've asked Michael. In the prime-field groups I think what's going on is this:

VT:

- We want to generate a random value in the range $[0, q - 1]$, but our source of randomness gives us bitstrings in the range $[0, 2^{q.\text{bitlength}} - 1]$, which is longer than we need (i.e. rounding up to the nearest power of 2). Assume it gives us a perfectly uniform distribution on this range.
- We can get a value inside the right range by modular reduction. Take r uniformly in $[0, 2^{q.\text{bitlength}} - 1]$, then return $r \bmod q$.
- But now we have a bias problem - some elements in $[0, q - 1]$ are more likely to occur than others, because the range $[0, 2^{q.\text{bitlength}} - 1]$ doesn't wrap an integer number of times around $[0, q - 1]$. (I *think* election guard just says 'oh well, it's a small bias' and doesn't worry about it in the prime-fields case, but I'm not certain.)
- I hadn't looked at this properly in Verificatum before, but I think Douglas is using n_r to reduce the bias: if we generate r from $[0, 2^{q.\text{bitlength} + n_r} - 1]$ instead, then we still have the same fundamental problem, but the difference between the most-likely and least-likely values is reduced. I haven't done the math, but I assume Douglas is right when he says this reduces the statistical error to n_r (probably $1/2^{n_r}$?).
- Short summary: I think we should just copy Douglas (though I also don't think it would be a disaster in practice if we ignored it). However, we definitely can't ignore the equivalent issue in the Elliptic Curve case, so I'm looking forward to hearing what Michael has to say.

JC: I think the statistical error is 2^{-n_r}

VT: Right!

JC: Im guessing this is the same issue as Michael recently sent me. I havent yet studied it enough to fully understand it. I am forwarding you our email conversation, and will include you on further.

JC: On Michael's advice, Im studying <https://www.rfc-editor.org/rfc/rfc9380.pdf>, section 5. Probably need to use 5.3 for mapping hash values to nonces, have a look when you can. Creating psuedo-random values for the generators is related, but I think different, but covered in the same RFC. Im leaning towards keeping Douglas's simple method for now, like you. But rfc9380 has a different algorithm, to avoid "probabilistic try-and-increment" (which Douglas's is), to avoid timing attacks. So we should evaluate that, esp if it ever goes into production. Maybe we should plan on replacing it at some point. TODO

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 $h_i = t_i^{(p-1)/2} \bmod p$.

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 $(x, \text{sqrt}(y))$ 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.

VT: Actually this is not a secure way to this, because the security of the Pedersen Commitments depends on the committer not knowing the discrete log of any of the values mod p . If you look carefully at the Pedersen commitment equation, you'll see that if you know the nonces, you can open it multiple different ways. (Olivier P and Thomas Haines and I wrote paper about how the Swiss/NSW Internet voting system got this wrong: <https://ieeexplore.ieee.org/document/9152765>) Section VIB(1) gives details about the commitment scheme, and how you can break it if you know the discrete logs of the generators. Section VIII describes some ways of securely generating the generators, with Verificatum as the example for Elliptic curves.

JC: Thanks, I knew I didnt understand this. Im going to add your paper as a reference. Also, since we are in principle enabling third party verifiers, I will document the (hopefully correct) algorithm in more detail.

JC: I've changed the algorithm as described above. Its provisional until I get more advice from Michael. Its a bit simpler by using an iterative generate and test, because we don't care about constant time algorithms. But it may be better in the long run to just use RFC9380 section 6.2.2.

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 $e_i = \text{nonce}_i$ (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.mixnet.makeBatchingVector()* and *org.cryptobiotic.maths.recursiveSHA256()* in [12] for the exact algorithm.

VT: I may be a little confused about how this fits in to the whole protocol, but when you're generating a challenge for a non-interactive zero knowledge proof, you need to include the proof statement into the hash. So for example if you've got an ElectionGuard ciphertext (a,b) and you want to prove that it's an encryption of 0 or 1, then you would include in the hash: a,b , the claim ("encryption of 0 or 1"), the group parameters (p,q,g) and all the other fixed election data you've described here.

JC: Im going to review what VMN does, now that I understand things better. Possibly Im missing something, for sure I should be using another PRG. The group parameters (p,q,g) are in *parameterBaseHash()*. I guess that W and W' correspond to the (a,b) of your example? Im not sure what the "the claim" might correspond to, but maybe the permutation commitments?

JC: Ok I think the missing thing that should be in the challenge is the "Proof of the Fiat-Shamir Proof", denoted τ^{pos} in 8.3, Algorithm 19 of [3]. I misinterpreted this as just the permutation commitments \vec{u} , but it actually includes a bunch more. So this is the missing "claim of proof". Since we use the seed from *makeBatchingVector()*, \vec{h} , \vec{u} , W and W' are also included, which I think is your "a,b".

VT: Right, exactly - the prover has to hash (a commitment to) the statement of what they're trying to prove, so they can't decide what to claim after they've seen the challenge.

VT: I'm afraid we just have to copy that hash calculation from Steps 2 and 4 of Algorithm 19 with exactly the same data. It doesn't matter if it's not in exactly the same form but it has to include hashes of all the right items. If you didn't already implement two different versions of the hash (RO-seed and RO-challenge) like Douglas has, I don't think that matters. What matters is that the challenge v depends on a hash of all the data that Douglas hashes into one or the other. In particular, you definitely need

- \vec{u} , W and W' , which do indeed correspond to (a,b) in my example, (this is the 'statement')
- τ^{pos} , which I forgot in my example, but which is crucial - this is the first message of the ZK proof,
- all the other auxiliary data about the group, the generators, the election public key etc.

VT: Added later: the above is wrong - I got confused between seed generation and challenge generation. No need to include τ^{pos} here.

VT: I like your high-level idea of incorporating ElectionGuard's parameterBaseHash into Verificatum's challenges. There's nothing wrong with hashing extra stuff, as long as we don't omit anything that Douglas hashed. I'm not quite sure how you're using makeBatchingVector, but one option is to put the ElectionGuard baseHash into the "prefix to random oracles", which [3] calls ρ . For example, we could have a "version" that incorporates the ElectionGuard baseHash, and is incorporated into ρ in Step 4 of Algorithm 28 of [3]. This is certainly not wrong, though I'm not really sure it achieves a lot apart from linking the "ElectionGuard" part of the protocol with the "Verificatum" part of the protocol. Incorporating the ElectionGuard baseHash in this way (assuming that its inclusion is verified) obviates the need to include the same values again explicitly when computing the challenge, so in principle we could go through the data hashed in step 2 of Algorithm 19 and carefully omit the ones that are already incorporated into the baseHash. However, I think it's probably easier to just include exactly what Douglas has included - there's nothing wrong with including something twice, but it can be bad to leave things out.

JC: So you're saying its not enough to use the seed from makeBatchingVector(), we need to explicitly add everything to the hash? I believe that τ^{pos} is exactly what I already have below, so I could add everything in the Batching Vector seed, rather than using the seed.

JC: The ElectionGuard hashing function (see section 5.5 of [9]) uses the form
- $H(key, domain\ separator, values) \rightarrow UInt256$.

Under the covers, it uses the HmacSHA256 function: "HmacSHA256 takes as input a key k and a message m of arbitrary length and returns a bit string of length 256 bits". Not sure exactly why the key is distinguished?

The PRG just hashes everything into the key, then uses sequential integers as the message to generate a sequence of hashes. This seems to be a standard way to do it, meaning thats how Dan Wallach implemented it, and I think thats how VMN does it too. Its trivial to create as many separate PRGs as we want. TODO review.

VT: Don't mind me - I got confused between the initial seed and the challenge generation. I think what you've done is fine - now I'll look at the Challenge section below. And no I don't know why ElectionGuard uses a keyed HMAC instead of a plain hash function. And no there's no need to add τ^{pos} here - that was my confusion - it gets added later on the fly, as you've done.

2.2.3 Challenge

- Create a recursive hash (as above) from the following fields in the Proof-Committment structure:
 - the seed that was made in makeBatchingVector() above
 - \vec{A}_p
 - \vec{B}
 - \vec{B}_p
 - \vec{C}_p
 - \vec{D}_p
 - \vec{F}_p
- Then challenge = hash.digest(). (so $n_v = 256$)

See *org.cryptobiotic.mixnet.makeChallenge()* and *org.cryptobiotic.maths.recursiveSHA256()* in [12] for the exact algorithm.

This challenge corresponds to τ^{pos} in Step 4 of Algorithm 19 of [3] and to the

line “ $y \leftarrow \text{GetChallenge}(y, t)$ ” of Algorithm 8.45 in [5]. Note that the actual value will be different from the corresponding libraries.

VT: Actually now this is perfectly clear. My only remaining question about this (the continuation of the high-level “does this match what Douglas did?” question) is to ask whether you followed his very careful structuring of the data into nodes. So you’ve definitely got both the seed s and the right data τ^{pos} - I haven’t looked recently at Douglas’s code but IIRC he has some careful structuring of combined values - it’s basically so that you can’t hash “anant” and later choose between “an ant” and “a nant”. That is, $\text{node}(A, B, C)$ makes a different hash from $\text{node}(A, \text{node}(B, C))$. I forget exactly how this implementation is done in Verificatum. The CHVote spec goes into great detail about these issues in Ch 4: in 4.4.1 they talk about prepending a type indicator when they hash things of different types. But the part that corresponds to Douglas’s node thing is CHVote 4.4.2. They say “it is important to ensure that collisions can not occur in a trivial manner. This happens for example when applying the hash algorithm directly to the concatenated byte arrays.” The point is that, when we write the math, we think of hashing as a single step, but it’s actually two different steps: (1) encode your data in the form that the hash function (or HMAC) can input; (2) apply the hash function (or HMAC). There’s a hash collision that occurs in a “trivial manner” if the encoding function causes two different inputs to be encoded in the same way. This breaks the essential collision-resistance property of the hash. So both CHVote and Douglas have gone to great lengths to structure their data encoding so that it’s not possible for two different (possibly complex) data inputs to produce the same hash. I’m not sure whether CHVote did exactly the same thing that Douglas did, or if they achieved an equivalent result in a slightly different way, but we should do something similar too (if you haven’t already).

JC: Ok, I missed the need for recursive hashing instead of concatenation. I will work on that, and also see what VMN and ElectionGuard are doing. Whew, am I glad you’re reviewing this!

VT: I’m really enjoying reviewing this. It’s interesting and I’m (re)learning a lot of interesting details as I go. Sorry it’s slow going. Also sorry for leaving you on your own for so long.

JC: Electionguard spec section 5.5, “Domain separation” often has byte concatenations, but each hash has a different “domain separator” byte. The key is almost always H_P or H_E , which ties it to a specific group or election respectively.

JC: VMN indeed uses a recursive hashing algorithm using an ordered tree. AFAICT, at the nodes, it hashes a byte representation that is very specific to the way VMN does serialization, using a bespoke serialization that isnt particularly well specified. This makes it unlikely that CHVote replicates that exactly, and is why I wasnt keen on doing it either. My impression is that much of what CHVote did is to clean up these dark corners of the VMN implementation, much like what we are doing.

VT: Yeah. The CHVote paper is a *lot* easier to read than the original.

JC: Its simple enough to use a hash of each value above, and hash that, rather than hash the concatenated bytes. But CHVote also hashes each component of an array or matrix separately, and then hashes those. I suppose I should just follow their strategy? Now Im wondering if thats how the original code I got from Dan worked, and I changed it without realizing? Ugh. The pitfalls of working alone, without a reference implementation. So, I'm inclined to follow Algorithm 4.14 of CHVote, if you agree.

VT: Yes I agree - let's just use 4.14 of CHVote. No reason not to.

JC: Tests show that recursive SHA-256 is similar in time to the EG hash using HmacSHA256 and byte array concatenation, See <https://github.com/JohnLCaron/egk-ec-mixnet/issues/29>. So Im now using recursive SHA-256. I threw in the element index number for collision resistance. Seems like its equivleant to how CHVote does it.

JC: I feel like this implementation is more or less interim until someone else (hopefully electionguard) makes a more "official" specification, that we will then want to copy in order to get interoperability. So, Im not working all that hard to specify it for now.

VT: That seems like a fine approach to me, especially since we're not really adding much when both the ElectionGuard team and the CHVote team have written such detailed specifications (or are supposed to be writing detailed specifications)

2.3 Shuffle

Let R be a matrix of $(n \times \text{width})$ randomly chosen reencryption nonces. For each ciphertext in W :

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

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

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

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

2.4 Proof

VT: I'm having a bit of trouble lining this up with [1] - is this an equivalent to Protocol 3, but with different notation? I was expecting the permutation commitments \vec{u} to turn up in the commitment to the shuffle, but probably I'm just confused...

JC: This is a reverse engineering of the code in VMN. Its not in [1], and I wasnt able to decipher [2]. The code does correspond to section 5.5.1 and Algorithm 8.45 of [5] "CHVote Protocol Specification Version 3.5". That paper has been the most useful for me. I wouldnt be surprised if CHVote also reverse engineered the code. Perhaps we can ask them?

This is a reverse engineering of the code in VMN. It corresponds to section 5.5.1 and Algorithm 8.45 of [5].

2.4.1 Commitment to permutation

Choose a vector of random permutation nonces $\vec{c\tau}$. Form public permutation commitments \vec{u} :

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

2.4.2 Commitment to shuffle

Choose vectors of n random nonces $\vec{b}, \vec{\beta}, e\vec{p}s$ and random nonces α, γ, δ , all $\in \mathbb{Z}_q$.

Form the following values $\in \mathbb{G}$:

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

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

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

$$C' = g^\gamma \quad (13)$$

$$D' = g^\delta \quad (14)$$

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:

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

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 \vec{eps} . We can use any of the following notations to indicate this:

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

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:

$$k_A = v \cdot \langle \vec{c}, \vec{e} \rangle + \alpha \quad (16)$$

$$\vec{k}_B = v \cdot \vec{b} + \vec{\beta} \quad (17)$$

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

$$k_D = v \cdot d + \delta \quad (19)$$

$$\vec{k}_E = v \cdot \vec{e'} + e\vec{p}s \quad (20)$$

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

$$k_{F,j} = v \cdot \langle \vec{R}_j, \vec{e'} \rangle + \phi_j \text{ for } j=1..\text{width} \quad (21)$$

where \langle, \rangle is the inner product of two vectors, and \cdot is scalar multiply.

2.4.5 The ProofOfShuffle Data Structure

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

```
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:

$$A = \prod_{i=1}^n u_i^{e_i} \quad (22)$$

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

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

and

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

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

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

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

$$C^v \cdot C' = g^{k_C} \quad (28)$$

$$D^v \cdot D' = g^{k_D} \quad (29)$$

and

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

3 Performance for the Integer Group

This section refers only to the Integer Group. We will add performance measurements for the Elliptic Group later.

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], [13] 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, n rows greater than 50, ShuffleAndProof gets around a 50 percent speedup, while Verify gets only a few percent.

3.3 Timing results

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

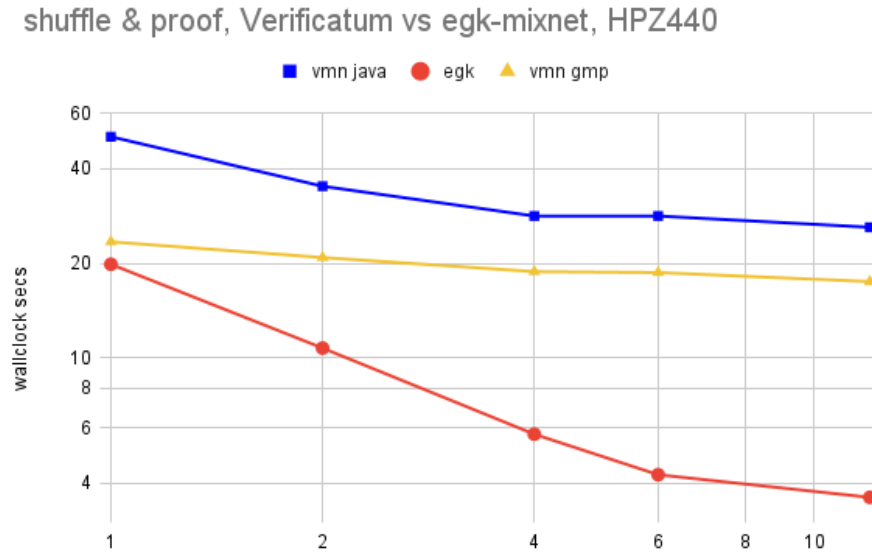


Figure 1: ShuffleAndProof on 100 ballots of width 34

verify, Verificatum vs egk-mixnet, HPZ440

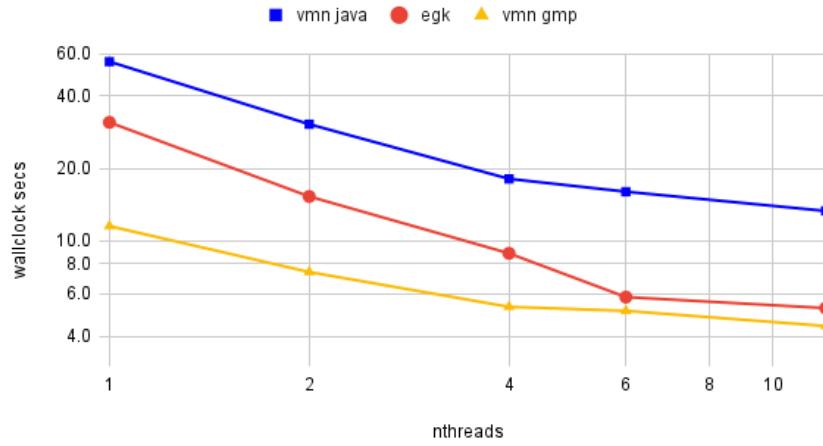


Figure 2: Verify on 100 ballots of width 34

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

Proof msec per row

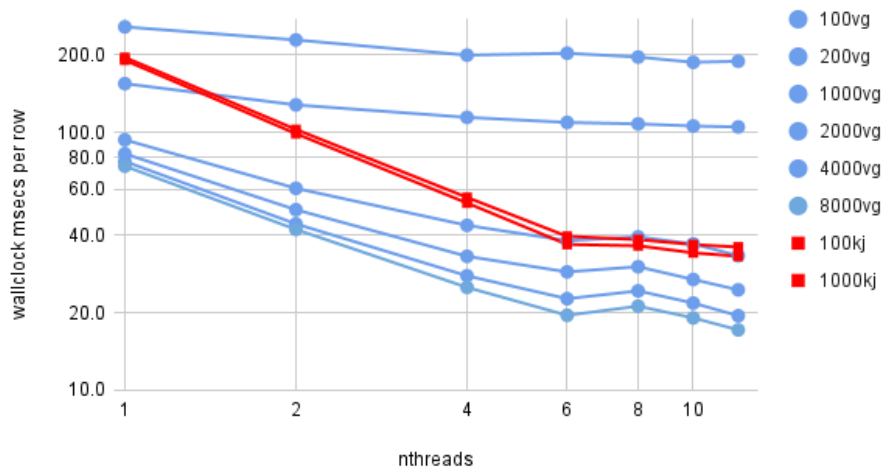


Figure 3: ShuffleAndProof for differing number of ballots

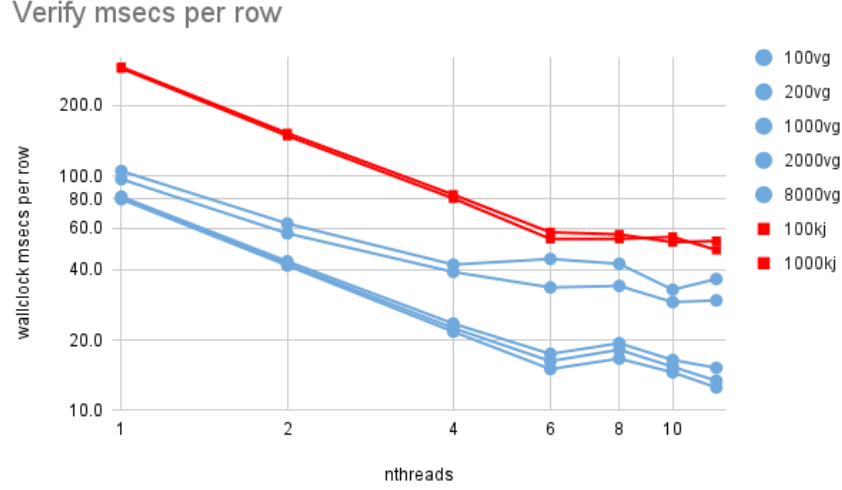


Figure 4: Verify for differing number of ballots

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 = nb ballots \times 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 [13]. 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 achievable 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 [12] for the relevant code.

There are a number of libraries, (eg [14]), that allow GMP to be called by Java using JNA (GMPEE 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, egk-ec-mixnet needs to replicate the GMPEE algorithm in C and call the GMP library directly. One can then do a large amount of computation for a single call into C code, so the overhead is minimal. I chose to use the new FFM feature in Java [15] which

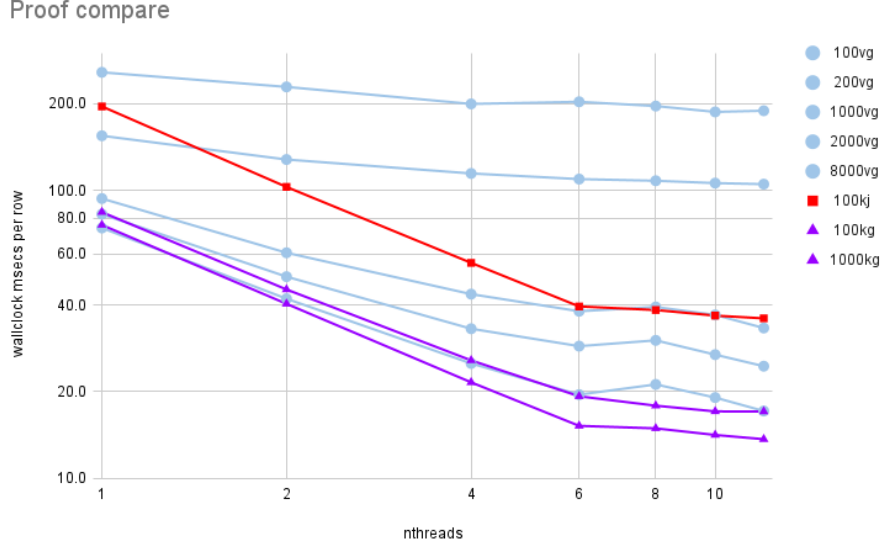


Figure 5: ShuffleAndProof compare

replaces JNI with a much safer and less brittle mechanism. The downside to this decision 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 egk-ec-mixnet will upgrade to the latest LTS (every two years) as a matter of policy.

Adding the option egk-ec-mixnet with GMP (kg) gives Figures 5 and 6 below.

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

egk-ec-mixnet uses 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

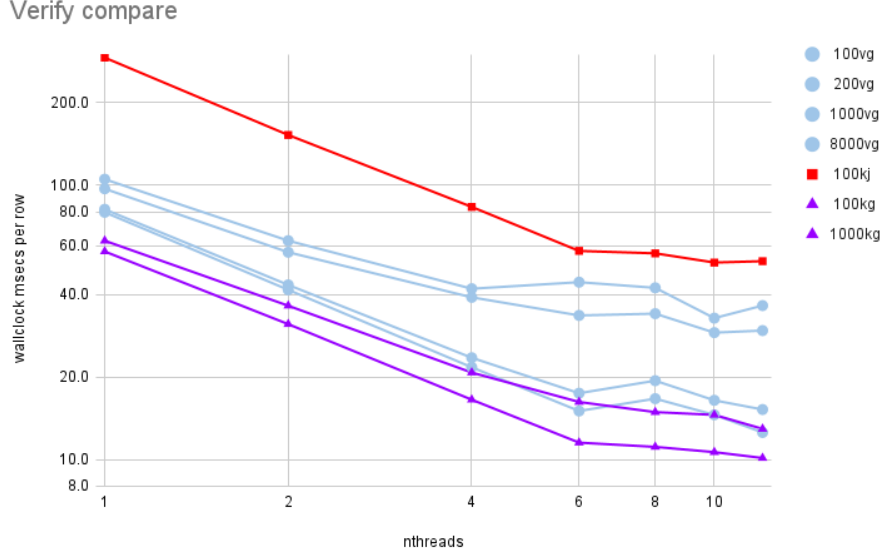


Figure 6: Verify compare

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 ekg-ec-mixnet Java version is competitive with it at higher thread counts and lower nb ballots.

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-ec-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.

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)

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

Let

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

then

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

B Alternative Calculation of B and B'

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

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

can be done to only use accelerated exponents.

Expand the series:

$$\begin{aligned}
B_1 &= g^{b_1} (h_0)^{e_1} \\
B_2 &= g^{b_2} (B_1)^{e_2} = g^{b_2 + b_1 e_2} \cdot h_0^{e_1 \cdot e_2} \\
B_3 &= g^{b_3} (B_2)^{e_3} = g^{b_3 + (b_2 + b_1 e_2) e_3} \cdot h_0^{e_1 \cdot e_2 \cdot e_3} \\
&\dots \\
B_i &= g^{b_i} (B_{i-1})^{e_i} = g^{geps_i} \cdot h_0^{heps_i}
\end{aligned}$$

where

$$\begin{aligned}
geps_1 &= b_1 \\
geps_i &= b_i + (geps_{i-1}) \cdot e_i, \quad i > 1 \\
heps_i &= \prod_{j=1}^i e_j
\end{aligned}$$

Then each row has exactly 2 accelerated exponentiations.

Similarly for B' :

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

can be done with only accelerated exponentiations:

$$B'_i = g^{gpeps_i} \cdot h_0^{hpeps_i}$$

where

$$\begin{aligned}
gpeps_1 &= \beta_1 \\
gpeps_i &= \beta_i + (gpeps_{i-1}) \cdot eps_i, \quad i > 1 \\
hpeps_1 &= eps_1 \\
hpeps_i &= hpeps_{i-1} \cdot eps_i, \quad i > 1
\end{aligned}$$

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

C Implementation notes

Permutation The permutation function is defined in Verificatum as the inverse of the definition here, and the operations of permute and invert are therefore switched. This is really a notational difference, and does not affect the mathematics. Also note that only the Shuffle/Prover works with the permutation, and the Verifier is not affected by the switch.

Side-channel attacks Verificatum has this warning: "Our code is not intended to be secure against side-channel attacks, since we do not need it for the applications we have in mind. It is the responsibility of the user to make sure that this is the case in their application." This needs to be evaluated.

Random number generation When generating random values, VMN has a "statDist" parameter like this, then does a modulus to restore the bitlength:

```
final int bitLength = modulus.bitLength() + statDist;
final BigInteger tmp = new BigInteger(bitLength, randomSource);
value = tmp.mod(modulus).value;
```

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

- [14] Gianluca Amato and Francesca Scozzari, *JGMP: Java bindings and wrappers for the GMP library*, [https://www.softxjournal.com/article/S2352-7110\(23\)00124-3](https://www.softxjournal.com/article/S2352-7110(23)00124-3).
- [15] JEP442, *Foreign Function And Memory API*, <https://openjdk.org/jeps/442>.
- [16] 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>.
- [17] 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>