Performance of Shuffling: Taking it to the Limits

Rolf Haenni[1] and Philipp Locher[1]

Bern University of Applied Sciences, CH-2501 Biel, Switzerland
{rolf.haenni,philipp.locher}@bfh.ch

Abstract. In this paper, we propose further performance improvements for Wikström's shuffle proof. Compared to an implementation based on general-purpose exponentiation algorithms, we show that the overall shuffle operation can be accelerated by approximately one order of magnitude. The improvements result partly from applying special-purpose algorithms for fixed-based and product exponentiations in the right way and partly from other optimization techniques. Given that shuffling is often one of the most time-consuming tasks in cryptographic voting protocols, the achieved speed-up is significant for practical implementations of electronic voting systems.

1 Introduction

Current proposals for cryptographic voting protocols are often based on verifiable re-encryption mix-nets. At the core of this approach is a cryptographic shuffle process under encryption, which is usually used to unlink the decryption of ciphertext votes from their submission by the voters. It is therefore a method to establish vote secrecy under the assumption that the mix-net includes sufficiently many independent mix-nodes performing single shuffle steps in sequential order. In such a setting, only a coalition of all mix-nodes can break vote secrecy, i.e., a single non-colluding mix-node is sufficient for achieving the desired security.

Generating and verifying the cryptographic shuffles of a mix-net for a large number of encrypted votes is often the most time-consuming operation in a voting protocol. Performance improvements at the core of this method are therefore relevant for the overall performance of the voting protocol. As an example, consider the verification of the shuffle proofs for an input size of $N = 100\,000$ ElGamal ciphertexts and a mix-net with four mix-nodes. Verifying a single Wikström shuffle proof requires approximately $9N$ exponentiations, i.e., $36N = 3\,600\,000$ exponentiations are needed for verifying all four proofs. Assuming that computing modular exponentiations on 3072-bits integers lasts approximately 9 milliseconds on regular hardware (according to measurements conducted in [11] using the fastest available libraries), we obtain approximately 9 hours of computations as a rough overall estimate. This result shows that performance optimizations of one or more orders of magnitude are more than welcome for improving the shuffle proof performance in practical implementations.

1.1 Related Work

There are several competing proposals for non-interactive shuffle proofs[1] in the literature. Provably secure methods by Furukawa and Sako [8], by Wikström and Terelius [18, 19], and by Bayer and Groth [1] are among the most efficient ones in the *random oracle model* (ROM). Methods discovered more recently based on bilinear pairings are provably secure in the *common reference string model* (CRS) [5, 10] or the *generic bilinear group model* (GBGM) [6, 7]. While the performance of pairing-based methods has improved in recent years, they are still slightly less efficient than comparable ROM methods (see Table 1). Furthermore, their dependence to pairing-friendly elliptic curves may pose a restriction in voting protocols, which require the encoding of votes in groups or fields of integers. Nevertheless, reports on remarkable performance results have demonstrated their maturity and potential for practical applications [6].

Task	Operation	[8]	[18, 19]	[1]	[5]	[6]
Shuffling	Exponentiations	$2N$	$2N$	$2N$	$2N$	$2N$
Proof generation	Exponentiations	$8N$	$8N$	$2N \log m$	$16N$	$8N$
Proof verification	Exponentiations	$10N$	$9N$	$4N$	$2N$	$7N$
	Pairings	–	–	–	$18N$	$3N$
Security model		ROM	ROM	ROM	CRS	GBGM

Table 1: Performance comparison of shuffle proofs for ElGamal ciphertexts. N denotes the size of the shuffle and $m = N/n$ an algorithm parameter from Bayer and Groth's method for trading-off performance against proof size.

For achieving performance results similar to [6], it is important to implement optimization techniques in a systematic manner. Special-purpose algorithms for fixed-base and product exponentiations are two of the most obvious and most rewarding techniques. Corresponding algorithms such as the *comb method* by Lim and Lee [14, 15] have been available for quite some time, but they are still not available very frequently in common libraries for large number arithmetic. A systematic analysis and comparison of fixed-base and product exponentiation algorithms have been conducted in [11]. For a given use case, the presented results are useful for selecting the best algorithm and optimal algorithm parameters.

The particular use case of Wikström's shuffle proof is also briefly discussed in [11]. For the 2048-bits setting and an input size of $N = 100\,000$ ciphertexts, a speed-up by a factor of 12.5 is reported for the proof generation. The proof

[1] Some authors distinguish between *zero-knowledge proofs* with statistical soundness and *zero-knowledge arguments* with computational soundness. According to this definition, many existing methods for proving the correctness of a shuffle are actually shuffle arguments. This is also the case for Wikström's method, which depends on computationally binding Pedersen commitments and therefore offers computational soundness only under the discrete logarithm assumption. By calling it a shuffle proof throughout this paper, we adopt the terminology of Wikström's original publications.

verification also benefits from the optimizations, but the reported speed-up by a factor of 3.85 is much more moderate.

1.2 Contribution and Paper Overview

This paper takes the discussion of Wikström's shuffle proof from [11] as a starting point for a more detailed and systematic analysis of possible performance optimizations. In addition to the aforementioned exponentiation algorithms, we also use batch verification techniques and methods for performing membership tests in groups of quadratic residues with minimal overhead. The main goal is to further speed up the performance of the proof verification algorithm, which seems to offer the greatest potential for further improvements.

In Section 2, we summarize existing optimization techniques. With respect to exponentiation algorithms, we mainly refer to the recently published survey paper [11] and adopt its notation and results. We also give a summary of existing batch verification techniques and demonstrate their potential in combination with efficient group membership tests.[2] In Section 3, we apply these techniques in a systematic way to Wikström's shvffle proof. Our analysis, which demonstrates that regular exponentiations can be avoided almost entirely, is based on counting the number of necessary multiplications in a prime-order group, for which the decisional Diffie-Hellman (DDH) assumption is believed to hold. The results of our analysis are therefore applicable to all groups commonly used in combination with the ElGamal encryption scheme. We conclude the paper in Section 4 with a summary of the achieved results and outlook to future research.

2 Performance Optimization Techniques

Computing exponentiations $z = \mathsf{Exp}(b, e) = b^e$ in a given mathematical group is often the most time-consuming operation in applications of public-key cryptography. For 2048-bits or 3072-bits numbers, computing a single modular exponentiation natively on off-the-shelf hardware is a matter of a few milliseconds. Other execution environments such as JavaScript engines are up to 30 times less efficient [11]. While this is still sufficiently efficient for simple tasks such as signing or encrypting a message, it may lead to a bottleneck in more complex tasks such as shuffling a large list of encrypted votes. Computations on elliptic curves are about one order of magnitude more efficient, but they are less frequently used in voting protocols.

2.1 Product and Fixed-Base Exponentiation

The recommended general-purpose algorithm for computing $z = b^e$ for a base $b \in \mathcal{G}$ and exponent $e \in \mathbb{Z}_q$ in a multiplicative group $(\mathcal{G}, \cdot, ^{-1}, 1)$ of prime order q is the *sliding window method* [16, Alg.14.85]. By referring to it as HAC 14.85,

[2] The idea of applying batch verification to shuffle proofs is due to Groth [9].

we adopt the notation from [11]. It has a single parameter $1 \le k \le \ell$ (the *window size*), which can be maximized for a given bit length $\ell = \|e\|$ of the exponent. Table 2 shows the running time of HAC 14.85 as a function of ℓ and k and optimal values k for some typical values ℓ. Running times are measured in expected number $M_k(\ell)$ of group multiplications (thus assuming that squarings and general multiplications are equally expensive).

Algorithm		Number of Multiplications	112	128	224	256	2048	3072
Plain	HAC 14.85	$M_k(\ell) = 2^{k-1} + \ell + \frac{\ell}{k+2}$	$k=3$		$k=4$		$k=6$	$k=7$
Product	HLG 2	$\widetilde{M}_m(\ell, N) = \frac{2^m + \ell}{m} + \frac{\ell}{N}$	$m=5$		$m=6$		$m=9$	
Fixed-base	HLG 3.2 HAC 14.117	$\widetilde{M}_{k,m}(\ell, N) = \frac{\ell}{N}\left(\frac{2^m}{km} + 1\right) + \frac{\ell}{m} + k$	maximize over $1 \le k \le \ell$ and $1 \le m \le \frac{\ell}{k}$					

Table 2: Expected number of multiplications and optimal algorithm parameters for plain, product, and fixed-based exponentiation algorithms. The expressions for $M_k(\ell)$, $\widetilde{M}_m(\ell, N)$, and $\widetilde{M}_{k,m}(\ell, N)$ have been slightly simplified for improved readability.

The particular task of computing the product $z = \mathsf{ProductExp}(\boldsymbol{b}, \boldsymbol{e}) = \prod_{i=1}^{N} b_i^{e_i}$ of exponentiations $z_i = b_i^{e_i}$ for $\boldsymbol{b} = (b_1, \ldots, b_N) \in \mathcal{G}^N$ and $\boldsymbol{e} = (e_1, \ldots, e_N) \in \mathbb{Z}_q^N$ can be computed much more efficiently than computing the N exponentiations individually using a general-purpose algorithm such as HAC 14.85. The most efficient *product exponentiation* (also called *simultaneous multi-exponentiation*) algorithm for small problem sizes N is the *interleaving method* from [17], but the precomputation table of size $O(2^N)$ prevents the algorithm from scaling.

For large problem instances, Algorithm 2 from [11] offers much better performance. In the remainder of this paper, we will refer to it as HLG 2. It has a single algorithm parameter $1 \le m \le N$, which denotes the size of the sub-tasks into which the problem is decomposed. If $M_m(\ell, N)$ denotes the total number of multiplications needed to solve a problem instance of size N and maximal exponent length $\ell = \max_{i=1}^{N} \|e_i\|$, then $\widetilde{M}_m(\ell, N) = M_m(\ell, N)/N$ denotes the *relative running time* of HLG 2. As shown in Table 2, $\widetilde{M}_m(\ell, N)$ depends on both ℓ and N, but the impact of N vanishes for large values N. Optimizing m is therefore largely independent of N. The parameters m shown in Table 2 are optimal for $N \ge 210$ (and nearly optimal for smaller values).

A second type of special-purpose exponentiation algorithms results from the problem of computing multiple exponentiations $z_i = b^{e_i}$ for a fixed base $b \in \mathcal{G}$. We denote this problem by $\boldsymbol{z} = \mathsf{FixedBaseExp}(b, \boldsymbol{e})$, where $\boldsymbol{z} = (z_1, \ldots, z_N) \in \mathcal{G}^N$ denotes the resulting exponentiations and $\boldsymbol{e} = (e_1, \ldots, e_N) \in \mathbb{Z}_q^N$ the given exponents. For solving this problem most efficiently, two *fixed-base exponentiation* algorithms exist with equivalent running times, the *comb method* by Lim and Lee [15] and Algorithm 3.2 from [11]. We refer to them as HAC 14.177 and HLG 3.2, respectively. Both of them are parametrized by two values, for example $1 \le k \le \ell$ (window size) and $1 \le m \le \ell/k$ (sub-task size) in the case of HLG 3.2.

The relative running time $\widetilde{M}_{k,m}(\ell, N)$ of HLG 3.2 is shown in Table 2. If N tends towards infinity, we get optimal parameters $k = 1$ and $m = \ell$, but otherwise the choice of k and m depends on both ℓ and N. For example, $k = 32$ and $m = 12$ are optimal for $\ell = 3072$ and $N = 1000$, i.e., $\widetilde{M}_{32,12}(3072, 1000) = 320$ group multiplications is the best possible performance in this particular case. Further exemplary performance results are depicted in Table 3, which also shows the benefits of special-purpose algorithms for product and fixed-base exponentiation. While HLG 2 performs between 5 to 9 times better than HAC 14.85 for product exponentiation, HLG 3.2/HAC 14.117 perform up to 25 times better than HAC 14.85 for fixed-base exponentiation.

	$\ell = 112$		$\ell = 128$		$\ell = 224$		$\ell = 256$		$\ell = 2048$		$\ell = 3072$		
HAC 14.85	138	1.00	157	1.00	269	1.00	306	1.00	2336	1.00	3477	1.00	
HLG 2	26	0.19	30	0.19	46	0.17	51	0.17	282	0.12	396	0.11	
HLG 3.2 HAC 14.117	22	0.16	27	0.17	43	0.16	49	0.16	313	0.13	449	0.13	N=100
	14	0.10	17	0.11	29	0.11	35	0.11	225	0.10	320	0.09	N=1 000
	11	0.08	12	0.08	22	0.08	25	0.08	176	0.08	255	0.07	N=10 000
	7	0.05	8	0.05	16	0.06	19	0.06	143	0.06	210	0.06	N=100 000
	5	0.04	6	0.04	12	0.04	15	0.05	120	0.05	176	0.05	N=1 000 000

Table 3: Comparison between exponentiation algorithms. For each exponent length $\ell \in \{112, 128, 224, 256, 2048, 3072\}$, the number of necessary group multiplications is shown in the left column, whereas the benefit of the optimization algorithm relative to HAC 14.85 is shown in the right column. All values are either taken or derived from [11].

2.2 Batch Verification

A particular task which sometimes appears in cryptographic protocols is testing for a batch of input triples $(z_1, b_1, e_1), \ldots, (z_N, b_N, e_N)$ whether $z_i = b_i^{e_i}$ holds for all N instances. We denote this problem by $\mathsf{BatchVerif}(\boldsymbol{z}, \boldsymbol{b}, \boldsymbol{e})$, where $\boldsymbol{z} \in \mathcal{G}^N$, $\boldsymbol{b} \in \mathcal{G}^N$, and $\boldsymbol{e} \in \mathbb{Z}_q^N$ denote respective vectors of input values. The trivial solution of computing all N exponentiations $b_i^{e_i}$ individually and comparing them with the given values z_i is not the most efficient one. The *small exponent test* (SET) from [2] solves $\mathsf{BatchVerif}(\boldsymbol{z}, \boldsymbol{b}, \boldsymbol{e})$ using a single equality test

$$\mathsf{ProductExp}(\boldsymbol{z}, \boldsymbol{s}) \stackrel{?}{=} \mathsf{ProductExp}(\boldsymbol{b}, \boldsymbol{s}') \tag{1}$$

between two product exponentiations of size N, where $\boldsymbol{s} = (s_1, \ldots, s_N) \in_R \mathbb{Z}_{2^s}^N$ is picked uniformly at random and $\boldsymbol{s}' = (s'_1, \ldots, s'_N)$ is derived from $\boldsymbol{s}$ and $\boldsymbol{e}$ by computing $s'_i = s_i e_i \bmod q$ over all N inputs.[3] The bit length s of the random exponents determines the failure probability 2^{-s} of the test. A failed test is one that returns true for a problem instance containing at least one $z_i \neq b_i^{e_i}$. Note

[3]Note that the algorithm given in [2] for solving $\mathsf{ProductExp}(\boldsymbol{z}, \boldsymbol{s})$ is not the most efficient one. Replacing it by HLG 2 improves the reported running times significantly.

that one product exponentiation deals with (short) exponents of length s and one with (long) exponents of length $\ell = \|q\|$. Therefore, the relative running time for solving $\mathsf{BatchVerif}(\boldsymbol{z}, \boldsymbol{b}, \boldsymbol{e})$ in this ways corresponds to

$$\widetilde{M}_{m_1,m_2}(s, \ell, N) = \widetilde{M}_{m_1}(s, N) + \widetilde{M}_{m_2}(\ell, N)$$

group multiplications per input (plus one modular multiplication in $\mathbb{Z}_q$ for computing $s'_i = s_i e_i \bmod q$). Optimal algorithm parameters m_1 and m_2 can be selected from Table 1 independently of N, for example $m_1 = 5$ and $m_2 = 9$ for solving a problem with parameters $s = 128$ and $\ell = 3072$. In this particular case, we obtain a total of $30 + 396 = 426$ multiplications in $\mathcal{G}$ (see Table 3) to perform the full batch verification. Compared to a naive implementation, this is approximately 8.5 times more efficient.

The small exponent test as described in [2] is defined for the special-case of a single fixed base b, in which the right-hand side of (1) can be replaced by a single exponentiation $\mathsf{Exp}(b, s')$ for $s' = \sum_{i=1}^{N} s'_i \bmod q$. Another special case arises in problems with a fixed exponent e, where the right-hand side of (1) becomes equivalent to $\mathsf{Exp}(\mathsf{ProductExp}(\boldsymbol{b}, \boldsymbol{s}), e)$. As shown in [13], these techniques can also be used in combination, for example if input values

$$\begin{array}{llll} \boldsymbol{z} = (z_1, \ldots, z_n), & \boldsymbol{b} = (b_1, \ldots, b_N), & \hat{\boldsymbol{b}} = (\hat{b}, \ldots, \hat{b}), & \tilde{\boldsymbol{b}} = (\tilde{b}_1, \ldots, \tilde{b}_N), \\ & \boldsymbol{e} = (e_1, \ldots, e_N), & \hat{\boldsymbol{e}} = (\hat{e}_1, \ldots, \hat{e}_N), & \tilde{\boldsymbol{e}} = (\tilde{e}, \ldots, \tilde{e}), \end{array}$$

are given ($\hat{b}$ and $\tilde{e}$ are fixed) and the problem consists in testing $z_i = b_i^{e_i} \hat{b}^{\hat{e}_i} \tilde{b}_i^{\tilde{e}}$ for all $1 \leq i \leq N$. We denote this particular combination of batch verification problems, which we will encounter in Section 3 in exactly this form, by $\mathsf{BatchVerif}(\boldsymbol{z}, \boldsymbol{b}, \boldsymbol{e}, \hat{b}, \hat{\boldsymbol{e}}, \tilde{\boldsymbol{b}}, \tilde{e})$. Solving it using the small exponent test means to perform the following equality test:

$$\mathsf{ProductExp}(\boldsymbol{z}, \boldsymbol{s}) \stackrel{?}{=} \mathsf{ProductExp}(\boldsymbol{b}, \boldsymbol{s}') \cdot \mathsf{Exp}(\hat{b}, s') \cdot \mathsf{Exp}(\mathsf{ProductExp}(\tilde{\boldsymbol{b}}, \boldsymbol{s}), \tilde{e}).$$

Based on the algorithms described in Section 2.1, the relative running time of this combined test consists of

$$\widetilde{M}_{k,m_1,m_2}(s, \ell, N) = 2\widetilde{M}_{m_1}(s, N) + \widetilde{M}_{m_2}(\ell, N) + \frac{2M_k(\ell) + 2}{N}$$

multiplications per input, which converges towards $2\widetilde{M}_{m_1}(s, N) + \widetilde{M}_{m_2}(\ell, N)$ when N increases. In the example from above with $s = 128$ and $\ell = 3072$, we get a total of $2 \cdot 30 + 396 = 456$ multiplications, which is approximately 23 times more efficient than computing $3N$ exponentiations without optimization.

Applications of the small exponent test are based on two critical preconditions [3]. First, $\mathcal{G}$ must be a prime-order group, which implies for example that the small exponent test is not applicable to the group $\mathbb{Z}_p^*$ of integers modulo p, which is of order $p-1$. Second, every z_i from $\boldsymbol{z}$ must be an element of $\mathcal{G}$. For arbitrary prime-order groups $\mathcal{G}$, group membership $z_i \in \mathcal{G}$ can be tested by $z_i^q = 1$ using one exponentiation with an exponent of maximal length $\ell = \|q\|$. But by executing this test N times for all values $z_1, \ldots, z_N$, the potential performance benefit of the small exponent test is no longer available.

2.3 Efficient Group Membership Tests for Quadratic Residues

Group membership in elliptic curves can be tested efficiently by checking if a given point satisfies the curve equation. For the subgroup $\mathbb{G}_q \subset \mathbb{Z}_p^*$ of integers modulo a safe prime $p = 2q + 1$, which is the most commonly used cryptographic setting for the ElGamal encryption scheme in voting protocols, group membership $z_i \in \mathbb{G}_q$ can be tested more efficiently using the Jacobi symbol $(\frac{z_i}{p}) \in \{-1, 0, 1\}$. For $\ell = 2048$, common $O(\ell^2)$ time algorithms for computing the Jacobi symbol run approximately twenty times faster than modular exponentiation [16]. This largely solves the above-mentioned group membership problem of the short exponent test in this particular setting.

To avoid different sorts of attacks, group membership tests are also required in many other cryptographic primitives. Performing such tests in a systematic way over *all* input values is therefore a best practice in the design and implementation of cryptographic applications. For very large inputs, even efficient Jacobi symbol algorithms may then become a target for performance optimizations.

Since elements of $\mathbb{G}_q$ are quadratic residues modulo p, we can reduce the cost of the group membership test to a single modular multiplication in $\mathbb{Z}_p^*$. The improvement is based on the observation that every quadratic residue $x \in \mathbb{G}_q$ has two square roots in $\mathbb{Z}_p^*$, whereas quadratic non-residues $x \notin \mathbb{G}_q$ have no square roots in $\mathbb{Z}_p^*$. Group membership $x \in \mathbb{G}_q$ can therefore be demonstrated by presenting one of the two square roots $\sqrt{x} = \pm x^{\frac{q+1}{2}} \bmod p$ as a *membership witness* and by checking that $\sqrt{x}^2 \equiv x \pmod{p}$ holds. Thus, provided that such a membership witness is available "for free", group membership can be tested using a single modular multiplication.[4]

To implement this idea in practice, elements $x \in \mathbb{G}_q$ can be represented as pairs $\hat{x} = (\sqrt{x}, x)$, for which group membership can be tested as described above using a single multiplication.[5] For such pairs, multiplication $\hat{z} = \hat{x}\hat{y}$, exponentiation $\hat{z} = \hat{x}^e$, and computing the inverse $\hat{z} = \hat{x}^{-1}$ can be implemented based on corresponding computations on the square roots:

$$\sqrt{xy} \equiv \sqrt{x}\sqrt{y} \pmod{p}, \quad \sqrt{x^e} \equiv \sqrt{x}^{\,e} \pmod{p}, \quad \sqrt{x^{-1}} \equiv \sqrt{x}^{\,-1} \pmod{p}.$$

Thus, only a single additional multiplication $z = \sqrt{z}^2 \bmod p$ is needed in each case to obtain the group element itself. In such an implementation, it is even possible to compute groups elements only when needed, for example before decoding a decrypted ElGamal message or for equality tests. In this way, additional multiplications and can be avoided almost entirely during the execution

[4] While group membership testing based on square roots has been used in protocols for outsourcing modular exponentiations to malicious servers [4], we are not aware of any proposal or implementation of this technique as a general method for representing elements of $\mathbb{G}_q \subset \mathbb{Z}_p^*$. However, given the simplicity of the approach, we can not exclude it from being folklore.

[5] To disallow the encoding of an additional bit of information into the square root representation of a quadratic residue, we suggest normalizing the representation by taking always either the smaller or the larger of the two values.

of a cryptographic protocol, during which all computations are conducted on the square roots. Restricting the representation to the square root is also useful for avoiding additional memory and communication costs. In other words, group membership in $\mathbb{G}_q \subset \mathbb{Z}_p^*$ can be guaranteed at almost no additional cost. This maximizes the benefit of the small exponent test in batch verification.

3 Optimizing the Performance of Shuffling

A *cryptographic shuffle* transforms a list of input ciphertexts $\boldsymbol{e} = (e_1, \ldots, e_N)$ into a permuted list $\tilde{\boldsymbol{e}} = (\tilde{e}_1, \ldots, \tilde{e}_N)$ of re-encrypted output ciphertexts, in which every $\tilde{e}_j = \mathsf{ReEnc}_{pk}(e_i, \tilde{r}_i)$, $j = \psi(i)$, is a re-encryption of exactly one e_i under the given public key pk. The whole shuffle operation can be denoted by

$$\tilde{\boldsymbol{e}} = \mathsf{Shuffle}_{pk}(\boldsymbol{e}, \tilde{\boldsymbol{r}}, \psi),$$

where $\tilde{\boldsymbol{r}} = (\tilde{r}_1, \ldots, \tilde{r}_N)$ denotes the vector of re-encryption randomizations and $\psi \in \Psi_N$ the randomly selected permutation (which determines the order of the elements in $\tilde{\boldsymbol{e}}$). For proving the correctness of $\tilde{\boldsymbol{e}}$ relative to $\boldsymbol{e}$, a non-interactive zero-knowledge proof of knowledge of $\tilde{\boldsymbol{r}}$ and ψ must be generated along with $\tilde{\boldsymbol{e}}$. Such a *shuffle proof*

$$\pi = \mathit{NIZKP}\left[(\tilde{\boldsymbol{r}}, \psi) : \tilde{\boldsymbol{e}} = \mathsf{Shuffle}_{pk}(\boldsymbol{e}, \tilde{\boldsymbol{r}}, \psi)\right]$$

can be constructed in various ways (see Table 1). In this paper, we only focus on the shuffle proof by Wikström and Terelius [18, 19], which is one of the most prominent and efficient approaches in the literature. Based on the pseudo-code algorithms from [12], we first provide a detailed list of all the exponentiations required for generating and verifying the shuffle. We will then discuss possible improvements in the light of the optimization techniques presented in Section 2. The goal is to reduce the overall running times to the greatest possible extent.

3.1 Performance Analysis

The shuffle proof by Wikström and Terelius is very flexible in various aspects. It supports different types of encryption schemes, different mathematical groups, and different operations to be performed on each ciphertext. The *Verificatum Mix-Net* implementation, for example, supports a combination of re-encryption and decryption with a shared private key [20]. In this particular setting, the mix-net outputs a list of decrypted plaintext messages, which correspond to the plaintext messages included in the input ciphertexts.

In our analysis, we restrict ourselves to the classical case of performing a re-encryption shuffle of ElGamal ciphertexts. Each ciphertext included in $\boldsymbol{e}$ is therefore a pair $e_i = (a_i, b_i) \in \mathcal{G}^2$ of two group elements $a_i = m_i \cdot pk^{r_i}$ and $b_i = g^{r_i}$ and re-encrypting e_i with a fresh randomization $\tilde{r}_i \in \mathbb{Z}_q$ means to multiply it with an encryption of the identity element $1 \in \mathcal{G}$:

$$\mathsf{ReEnc}_{pk}(e_i, \tilde{r}_i) = e_i \cdot \mathsf{Enc}_{pk}(1, \tilde{r}_i) = (a_i, b_i) \cdot (pk^{\tilde{r}_i}, g^{\tilde{r}_i}) = (a_i \cdot pk^{\tilde{r}_i}, b_i \cdot g^{\tilde{r}_i}).$$

On the other hand, we do not restrict the analysis to a particular mathematical group, i.e., the presented results are applicable to any DDH secure group. As in Section 2, the performance is measured in number of group multiplications, i.e., without making a distinction between general multiplication and squaring.

Slightly modified versions of the pseudo-code algorithms GenShuffle, GenProof, and CheckProof from [12] are included in Appendix A. For better readability, two sub-algorithms have been merged into GenProof and some variables have been renamed for better consistency with the rest of the paper. To avoid negative exponents $-c$ in CheckProof, we have swapped all appearances of c and $-c$ in both algorithms. The benefit of this modification is computations with smaller exponents. If λ denotes the security parameter and $\ell = \|q\|$ an appropriate group size in bits, for example $\lambda = 128$ and $\ell = 3072$, we get much smaller exponents of length $\|c\| = \lambda$ instead of $\|-c\| = \ell$. This improves the running time of CheckProof independently of any optimization techniques.

Based on the algorithms as given in the appendix, Table 4 provides a complete list of all exponentiations required for generating a cryptographic shuffle of size N and the corresponding shuffle proof. The lengths of the involved exponents are indicated in each case. As a general rule, randomizations such as r_i or ω_i are of length $\ell = \|q\|$, whereas challenges such as $\tilde{u}_i$ are of length λ, where λ is the security parameter of the shuffle proof. The rightmost column of Table 4 shows the number of necessary group membership tests to conduct on the input values. It is assumed that independent generators $g, h, h_1, \ldots, h_N \in \mathcal{G}$ are publicly known.

In Table 5, a similar overview of exponentiations and group membership tests is given for algorithm CheckProof, again by distinguishing between exponents of length ℓ and λ. By comparing Table 5 with Table 4, it seems that generating and verifying a shuffle is almost equally expensive. In each case, there are exactly N plain exponentiations and $3N$ product exponentiations with large ℓ-bits exponents, and roughly N plain exponentiations with small λ-bits exponents. The main difference lies in the number of product exponentiations with small exponents and fixed-base exponentiations with large exponents, but the total sum of all exponentiations is almost identical ($10N + 5$ for generating the shuffle vs. $9N + 11$ for verifying the shuffle). A major difference lies in the number of group membership tests ($2N + 1$ vs. $7N + 6$).

3.2 Performance Improvements

The analysis of the previous subsection give us a precise map of how to apply the special-purpose exponentiation algorithms from Section 2.1 for improving the performance of generating and verifying a cryptographic shuffle. Based on this map, we can compute the total number of multiplications required for an input size of N ElGamal ciphertext. By dividing this number by N, we obtain *relative running times* for generating and verifying the shuffle, which measures the average number of multiplications *per* input ciphertext. The columns in the middle of Tables 6 and 7 show corresponding numbers for $\lambda = 128$ and $\ell = 3072$ and $\lambda = 128$ and $\ell = 256$, which are typical settings today for modular

Algorithm	Line	Computation	PLE		PRE	FBE		GMT
			ℓ	λ	ℓ	ℓ	b	
GenShuffle	1a	$(a_i, b_i) \in \mathcal{G}^2$	–	–	–	–	–	$2N$
	1b	$pk \in \mathcal{G}$	–	–	–	–	–	1
	5	$\tilde{a}_i \leftarrow a_i \cdot pk^{\tilde{r}_i}$	–	–	–	N	pk	–
	6	$\tilde{b}_i \leftarrow b_i \cdot g^{\tilde{r}_i}$	–	–	–	N	g	–
GenProof	4	$c_{j_i} \leftarrow h_i \cdot g^{r_{j_i}}$	–	–	–	N	g	–
	11	$\hat{c}_i \leftarrow g^{\hat{r}_i} \cdot \hat{c}_{i-1}^{\tilde{u}_i}$	–	N	–	N	g	–
	15	$\hat{t}_i \leftarrow g^{\hat{\omega}_i} \cdot \hat{c}_{i-1}^{\tilde{\omega}_i}$	N	–	–	N	g	–
	17	$t_1 \leftarrow g^{\omega_1}$	–	–	–	1	g	–
	18	$t_2 \leftarrow g^{\omega_2}$	–	–	–	1	g	–
	19	$t_3 \leftarrow g^{\omega_3} \cdot \prod_{i=1}^{N} h_i^{\tilde{\omega}_i}$	–	–	N	1	g	–
	20	$t_{4,1} \leftarrow pk^{-\omega_4} \cdot \prod_{i=1}^{N} \tilde{a}_i^{\tilde{\omega}_i}$	–	–	N	1	pk	–
	21	$t_{4,2} \leftarrow g^{-\omega_4} \cdot \prod_{i=1}^{N} \tilde{b}_i^{\tilde{\omega}_i}$	–	–	N	1	g	–
Total			$10N + 5$					$2N + 1$

Table 4: Overview of exponentiations and group membership tests in the shuffle and shuffle proof generation algorithms. The column PLE lists the number of plain exponentiations, the column PRE the number of product exponentiations, the column FBE the number of fixed-based exponentiations, and the column GMT the number of group membership tests for an input size of N ciphertexts, a group $\mathcal{G}$ of size $\ell = \|q\|$ bits, and a security parameter λ.

groups respectively elliptic curves. Compared to the numbers for an unoptimized implementation shown in the left hand columns, generating the shuffle becomes up to 5.5 times and verifying the shuffle up to 3.5 times more efficient. Generally, the speed-up for $\ell = 256$ is slightly smaller than for $\ell = 3072$, and it grows moderately for an increasing N.

Given the potential of product and fixed-base exponentiation algorithms (between 10 and 20 times more efficient for $\ell = 3072$, see Table 3), the maximum performance improvement has not yet been achieved. The most problematic exponentiations in Tables 4 and 5 are the $2N$ plain exponentiations, and among them especially those with exponents of length ℓ. Apparently, computing them without optimizations creates a significant bottleneck that prevents even better performances. Here is a proposal for removing the bottleneck in all cases:

- The second exponentiation in the assignment $\hat{t}_i \leftarrow g^{\hat{\omega}_i} \cdot \hat{c}_{i-1}^{\tilde{\omega}_i}$ in Line 15 of GenProof is based on the commitment chain $\hat{c}_0, \hat{c}_1, \ldots, \hat{c}_N$, which is defined recursively by $\hat{c}_0 \leftarrow h$ in Line 8 and $\hat{c}_i \leftarrow g^{\hat{r}_i} \cdot \hat{c}_{i-1}^{\tilde{u}_i}$ in Line 11. By raising the recursion to the exponent, we can reformulated this definition into $\hat{c}_i \leftarrow g^{R_i} \cdot h^{U_i}$, where exponents $R_i = \hat{r}_i + \tilde{u}_i R_{i-1} \bmod q$ and $U_i = \tilde{u}_i U_{i-1} \bmod q$ are computed recursively from $R_0 = 0$ and $U_0 = 1$ in time linear to N. By changing Line 11 accordingly, we obtain two fixed-base exponentiations—one for base g and one for base h—with exponents of length $\ell = \|q\|$.

- Based on the same exponents R_i and U_i, we can also change Line 15 of GenProof into $\hat{t}_i \leftarrow g^{R'_i} \cdot h^{U'_i}$ with exponents $R'_i = \hat{\omega}_i + \tilde{\omega}_i R_{i-1} \bmod q$ and

Algorithm	Line	Computation	PLE		PRE		FBE		GMT
			ℓ	λ	ℓ	λ	ℓ	b	
CheckProof	1a	$t \in \mathcal{G} \times \mathcal{G} \times \mathcal{G} \times \mathcal{G}^2 \times \mathcal{G}^N$	–	–	–	–	–	–	$N+5$
	1b	$\boldsymbol{c} \in \mathcal{G}^N$, $\hat{\boldsymbol{c}} \in \mathcal{G}^N$	–	–	–	–	–	–	$2N$
	1c	$(a_i, b_i) \in \mathcal{G}^2$, $(\tilde{a}_i, \tilde{b}_i) \in \mathcal{G}^2$	–	–	–	–	–	–	$4N$
	1d	$pk \in \mathcal{G}$	–	–	–	–	–	–	1
	7	$\hat{c} \leftarrow \hat{c}_N \cdot h^{-u}$	–	–	–	–	1	h	–
	8	$\tilde{c} \leftarrow \prod_{i=1}^{N} c_i^{u_i}$	–	–	–	N	–	–	–
	9	$\tilde{a} \leftarrow \prod_{i=1}^{N} a_i^{u_i}$	–	–	–	N	–	–	–
	10	$\tilde{b} \leftarrow \prod_{i=1}^{N} b_i^{u_i}$	–	–	–	N	–	–	–
	13	$\hat{t}'_i \leftarrow \hat{c}_i^{\,c} \cdot g^{\hat{s}_i} \cdot \hat{c}_{i-1}^{\tilde{s}_i}$	N	N	–	–	N	g	–
	14	$t'_1 \leftarrow \bar{c}^{\,c} \cdot g^{s_1}$	–	1	–	–	1	g	–
	15	$t'_2 \leftarrow \hat{c}^{\,c} \cdot g^{s_2}$	–	1	–	–	1	g	–
	16	$t'_3 \leftarrow \tilde{c}^{\,c} \cdot g^{s_3} \cdot \prod_{i=1}^{N} h_i^{\tilde{s}_i}$	–	1	N	–	1	g	–
	17	$t'_{4,1} \leftarrow \tilde{a}^{\,c} \cdot pk^{-s_4} \cdot \prod_{i=1}^{N} \tilde{a}_i^{\tilde{s}_i}$	–	1	N	–	1	pk	–
	18	$t'_{4,2} \leftarrow \tilde{b}^{\,c} \cdot g^{-s_4} \cdot \prod_{i=1}^{N} \tilde{b}_i^{\tilde{s}_i}$	–	1	N	–	1	g	–
Total			$9N+11$						$7N+6$

Table 5: Overview of exponentiations and group membership tests in the shuffle verification algorithm. The column PLE lists the number of plain exponentiations, the column PRE the number of product exponentiations, the column FBE the number of fixed-based exponentiations, and the column GMT the number of group membership tests for an input size N, a group $\mathcal{G}$ of size $\ell = \|q\|$ bits, and a security parameter λ.

$U'_i = \tilde{\omega}_i U_{i-1} \bmod q$, which again consists of two fixed-base exponentiations with exponents of length ℓ. Therefore, all plain exponentiations from algorithm GenProof can be replaced by fixed-base exponentiations. Together with GenShuffle, we obtain a total of $3N$ product exponentiations and $7N+5$ fixed-base exponentiations ($N+1$ for pk, $4N+4$ for g, $2N$ for h) for generating the shuffle and its proof. All exponents are of length $\ell = \|q\|$.

– The two plain exponentiations of algorithm CheckProof are both contained in the assignment $\hat{t}'_i \leftarrow \hat{c}_i^{\,c} \cdot g^{\hat{s}_i} \cdot \hat{c}_{i-1}^{\tilde{s}_i}$ of Line 13. The purpose of computing the values $\hat{t}'_i$ is to compare them in Line 19 with the values $\hat{t}_i$ included in the proof. They must all be equal for the proof to succeed. Instead of conducting explicit equality tests as suggested in algorithm CheckProof, it is also possible to apply the batch verification method from Section 2.2. Note that the given use case in Line 19 corresponds precisely to the particular combination of batch verification problems discussed at the end of Section 2.2, which tests three exponentiations simultaneously (one with a fixed base g, one with a fixed exponent c, and one general case). For $\hat{\boldsymbol{t}} = (\hat{t}_1, \ldots, \hat{t}_N)$, $\hat{\boldsymbol{c}} = (\hat{c}_1, \ldots, \hat{c}_N)$, and $\hat{\boldsymbol{c}}_0 = (\hat{c}_0, \ldots, \hat{c}_{N-1})$, we can therefore execute the combined small exponent test from Section 2.2 to perform $\mathsf{BatchVerif}(\hat{\boldsymbol{t}}, \hat{\boldsymbol{c}}_0, \tilde{\boldsymbol{s}}, g, \hat{\boldsymbol{s}}, \hat{\boldsymbol{c}}, c)$, which requires three product exponentiations of size N (one with exponents of length ℓ and

Not optimized				Partly optimized				Fully optimized				
Generate		Verify		Generate		Verify		Generate		Verify $s = \lambda$		N
31622	1.00	18221	1.00	6742	0.21	5406	0.30	3908	0.12	1861	0.10	100
31465	1.00	18027	1.00	6264	0.20	5233	0.29	3230	0.10	1740	0.10	1 000
31450	1.00	18007	1.00	5971	0.19	5162	0.29	2817	0.09	1730	0.10	10 000
31448	1.00	18007	1.00	5782	0.18	5117	0.28	2546	0.08	1729	0.10	100 000
31448	1.00	18005	1.00	5640	0.18	5083	0.28	2346	0.07	1729	0.10	1 000 000

Table 6: Comparison of relative running times for generating and verifying a cryptographic shuffle in a typical setting for modular groups with $\lambda = 128$ and $\ell = 3072$.

Not optimized				Partly optimized				Fully optimized				
Generate		Verify		Generate		Verify		Generate		Verify $s = \lambda$		N
2926	1.00	2184	1.00	822	0.28	768	0.35	445	0.15	374	0.17	100
2913	1.00	2161	1.00	763	0.26	742	0.34	362	0.12	356	0.16	1 000
2911	1.00	2158	1.00	725	0.25	731	0.34	308	0.11	354	0.16	10 000
2911	1.00	2158	1.00	699	0.24	725	0.33	270	0.09	354	0.16	100 000
2911	1.00	2158	1.00	683	0.23	721	0.33	248	0.09	354	0.16	1 000 000

Table 7: Comparison of relative running times for generating and verifying a cryptographic shuffle in a typical setting for elliptic curves with $\lambda = 128$ and $\ell = 256$.

two with exponents of length s), and two single plain exponentiations. This implies that CheckProof can be implemented using nine product exponentiations of size N (four with exponents of lengths ℓ, three with exponents of length λ, and two with exponents of length s) and 13 single exponentiations (which become negligible for large problem instances). When implementing batch verification into the non-interactive verification algorithm in this way, care has to be taken that the process of picking $\boldsymbol{s}$ uniformly at random can not be influenced by the prover.

Rewriting GenProof and CheckProof using these optimization leads to the performance results shown in the right hand columns of Tables 6 and 7. Note that the change in CheckProof increases the probability for an invalid proof to pass the verification by 2^{-s} (see Section 2.2). To preserve the soundness of the proof, it is therefore important to select s in accordance with the security parameter λ. For $s = \lambda = 112$ and $\ell = 3072$, a speed-up by a factor of 10 and more can be observed for both generating and verifying the shuffle. For $s = \lambda = 128$ and $\ell = 256$, the speed-up is slightly more moderate, but still significant.

With the above optimizations, it seems that the potential for improvements based on special-purpose exponentiations algorithms has been exhausted. In groups $\mathbb{G}_q \subseteq \mathbb{Z}_p^*$ of integers modulo a safe prime $p = 2q + 1$, an area for further improvements are the group membership tests, which need to be conducted on all group elements included in the algorithm inputs (we mentioned earlier that

membership tests in elliptic curves are almost for free). Recall from Tables 4 and 5 and from Section 2.3 that an ElGamal shuffle of size N requires $2N + 1$ such membership tests for generating and $7N + 6$ tests for verifying the shuffle.

If membership testing $z \in \mathbb{G}_q$ is implemented naïvely by computing $z^q \bmod p$ using plain modular exponentiation, then the added cost of this test completely outweighs the performance improvements achieved so far. Therefore, we assume that any practical implementation at leasts includes an algorithm for computing the Jacobi symbol, which is up to 20 times faster than plain modular exponentiation for 3072-bits integers. Given that HAC 14.85 requires 3477 modular multiplications, we can estimate the cost of computing the Jacobi symbol as equivalent to approximately 175 multiplications. Therefore, $2{\cdot}175 + 1 = 351$ and $7{\cdot}175 + 6 = 1231$ multiplications need to be added to the cost of generating and verifying the shuffle, respectively. Compared to the numbers from Table 6, this demonstrates that the cost for computing Jacobi symbols is not negligible, especially for verifying a proof. For large N, we obtain a total of $1729 + 1231 = 2960$ multiplications per input ciphertext, which is approximately 1.7 less efficient than without performing the membership test. This loss can be avoided by implementing the membership test based on the membership witness method from Section 2.3, which reduces the relative cost to a single multiplication.

4 Conclusion

Based on recent work on special-purpose algorithms for computing exponentiations, we have shown in this paper that generating and verifying a cryptographic shuffle can be accelerated by approximately one order of magnitude. A combination of optimization techniques is necessary to obtain the best possible performance. Given the importance of shuffling in voting protocols and the high computational costs of the available methods, this improvement is significant for practical implementations. We have shown how to achieve this benefit for Wikström's shuffle proof, but we expect similar benefits for other methods.

Many of the algorithms and methods discussed in this paper are not yet available in libraries for large integer arithmetic. We were therefore not able to evaluate the performance of the proposed method on real machines. But recent work on similar topics has shown that theoretical performance estimations based on counting multiplications can often be confirmed rather easily in practical experiments. Implementing all algorithms and conducting such experiments is an area for further research.

Further performance improvements can be achieved by executing the exponentiation tasks in parallel on multiple cores or multiple machines. There are many ways of implementing parallelization into a shuffling procedure, but finding a clever way of distributing the total cost optimally to all available resources is a complex problem. This is another area for further research.

References

[1] Bayer, S., Groth, J.: Efficient zero-knowledge argument for correctness of a shuffle. EUROCRYPT'12, 31st Annual International Conference on Theory and Applications of Cryptographic Techniques. pp. 263–280. LNCS 7237, Cambridge, UK (2012)

[2] Bellare, M., Garay, J.A., Rabin, T.: Fast batch verification for modular exponentiation and digital signatures. EUROCRYPT'98, 17th International Conference on the Theory and Application of Cryptographic Techniques. pp. 236–250. LNCS 1403, Espoo, Finland (1998)

[3] Boyd, C., Pavlovski, C.: Attacking and repairing batch verification schemes. ASIACRYPT'00, 6th International Conference on the Theory and Application of Cryptology and Information Security. pp. 58–71. LNCS 1976, Kyoto, Japan (2000)

[4] Di Crescenzo, G., Khodjaeva, M., Kahrobaei, D., Shpilrain, V.: Practical and secure outsourcing of discrete log group exponentiation to a single malicious server. CCSW'17, 9th ACM Cloud Computing Security Workshop. pp. 17–28. Dallas, USA (2017)

[5] Fauzi, P., Lipmaa, H.: Efficient culpably sound NIZK shuffle argument without random oracles. CT–RSA'16, The Cryptographers' Track at the RSA Conference. pp. 200–216, LNCS 9610, San Francisco, USA (2016)

[6] Fauzi, P., Lipmaa, H., Siim, J., Zając, M.: An efficient pairing-based shuffle argument. ASIACRYPT'17, 23rd International Conference on the Theory and Application of Cryptology and Information Security. pp. 97–127. LNCS 10625, Hongkong, China (2017)

[7] Fauzi, P., Lipmaa, H., Zając, M.: A shuffle argument secure in the generic model. ASIACRYPT'16, 22nd International Conference on the Theory and Application of Cryptology and Information Security. pp. 841–872. LNCS 10032, Hanoi, Vietnam (2016)

[8] Furukawa, J., Sako, K.: An efficient scheme for proving a shuffle. CRYPTO'01, 21st Annual International Cryptology Conference on Advances in Cryptology. pp. 368–387. LNCS 2139, Santa Barbara, USA (2001)

[9] Groth, J.: A verifiable secret shuffle of homomorphic encryptions. *Journal of Cryptology*, 23(4):546–579, 2010.

[10] Groth, J., Lu, S.: A non-interactive shuffle with pairing based verifiability. ASIACRYPT'08, 13th International Conference on the Theory and Application of Cryptology and Information Security. pp. 51–67. LNCS 4833, Kuching, Malaysia (2007)

[11] Haenni, R., Locher, P., Gailly, N.: Improving the performance of cryptographic voting protocols. Voting'19, 4th Workshop on Advances in Secure Electronic Voting. Basseterre, St. Kitts and Nevis (2019)

[12] Haenni, R., Locher, P., Koenig, R.E., Dubuis, E.: Pseudo-code algorithms for verifiable re-encryption mix-nets. Voting'17, 2nd Workshop on Advances in Secure Electronic Voting. pp. 370–384. LNCS 10323, Silema, Malta (2017)

[13] Hoshino, F., Abe, M., Kobayashi, T.K.: Lenient/strict batch verification in several groups. ICISC'01, 4th International Conference on Information Security and Cryptology. pp. 81–94. LNCS 2200, Seoul, South Korea (2001)
[14] Lee, P.J., Lim, C.H.: Method for exponentiation in a public-key cryptosystem. United States Patent No. 5999627 (December 1999)
[15] Lim, C.H., Lee, J.P.: More flexible exponentiation with precomputation. CRYPTO'94, 14th Annual International Cryptology Conference on Advances in Cryptology. pp. 95–107. LNCS 839, Santa Barbara, USA (1994)
[16] Menezes, A.J., van Oorschot, P.C., Vanstone, S.A.: Handbook of Applied Cryptography. CRC Press, Boca Raton, USA (1996)
[17] Möller, B.: Algorithms for multi-exponentiation. SAC'01, 8th Annual International Workshop on Selected Areas in Cryptography. pp. 165–180. LNCS 2259, Toronto, Canada (2001)
[18] Terelius, B., Wikström, D.: Proofs of restricted shuffles. AFRICACRYPT'10, 3rd International Conference on Cryptology in Africa. pp. 100–113. LNCS 6055, Stellenbosch, South Africa (2010)
[19] Wikström, D.: A commitment-consistent proof of a shuffle. ACISP'09, 14th Australasian Conference on Information Security and Privacy. pp. 407–421. LNCS 5594, Brisbane, Australia (2009)
[20] Wikström, D.: User Manual for the Verificatum Mix-Net – VMN Version 3.0.3. Verificatum AB, Stockholm, Sweden (2018)

A Pseudo-Code Algorithms

1 **Algorithm:** $\mathsf{GenShuffle}(\boldsymbol{e}, pk)$
Input: ElGamal ciphertexts $\boldsymbol{e} = (e_1, \ldots, e_N)$, $e_i = (a_i, b_i) \in \mathcal{G}^2$
Encryption key $pk \in \mathcal{G}$
2 $\psi \leftarrow \mathsf{GenPermutation}(N)$
3 **for** $i = 1, \ldots, N$ **do**
4 $\quad \tilde{r}_i \in_R \mathbb{Z}_q$
5 $\quad \tilde{a}_i \leftarrow a_i \cdot pk^{\tilde{r}_i}$
6 $\quad \tilde{b}_i \leftarrow b_i \cdot g^{\tilde{r}_i}$
7 $\quad \tilde{e}_i \leftarrow (\tilde{a}_i, \tilde{b}_i)$
8 $\tilde{\boldsymbol{e}} \leftarrow (\tilde{e}_{j_1}, \ldots, \tilde{e}_{j_N})$
9 $\tilde{\boldsymbol{r}} \leftarrow (\tilde{r}_1, \ldots, \tilde{r}_N)$
10 **return** $(\tilde{\boldsymbol{e}}, \tilde{\boldsymbol{r}}, \psi)$ // $\tilde{\boldsymbol{e}} \in (\mathcal{G}^2)^N$, $\tilde{\boldsymbol{r}} \in \mathbb{Z}_q^N$, $\psi \in \Psi_N$

Algorithm A.1: Performs a re-encryption shuffle to a given list of ElGamal ciphertexts.

Algorithm: $\mathsf{GenProof}(\boldsymbol{e}, \tilde{\boldsymbol{e}}, \tilde{\boldsymbol{r}}, \psi, pk)$

Input: ElGamal ciphertexts $\boldsymbol{e} = (e_1, \ldots, e_N)$, $e_i = (a_i, b_i) \in \mathcal{G}^2$
Shuffled ElGamal ciphertexts $\tilde{\boldsymbol{e}} = (\tilde{e}_1, \ldots, \tilde{e}_N)$, $\tilde{e}_i = (\tilde{a}_i, \tilde{b}_i) \in \mathcal{G}^2$
Re-encryption randomizations $\tilde{\boldsymbol{r}} = (\tilde{r}_1, \ldots, \tilde{r}_N)$, $\tilde{r}_i \in \mathbb{Z}_q$
Permutation $\psi = (j_1, \ldots, j_N) \in \Psi_N$
Encryption key $pk \in \mathcal{G}$

for $i = 1, \ldots, N$ **do**
  $r_{j_i} \in_R \mathbb{Z}_q$
  $c_{j_i} \leftarrow h_i \cdot g^{r_{j_i}}$
$\boldsymbol{c} = (c_1, \ldots, c_N)$
for $i = 1, \ldots, N$ **do**
  $u_i \leftarrow \mathsf{Hash}((\boldsymbol{e}, \tilde{\boldsymbol{e}}, \boldsymbol{c}), i)$
$\hat{c}_0 \leftarrow h$
for $i = 1, \ldots, N$ **do**
  $\hat{r}_i \in_R \mathbb{Z}_q$, $\tilde{u}_i \leftarrow u_{j_i}$
  $\hat{c}_i \leftarrow g^{\hat{r}_i} \cdot \hat{c}_{i-1}^{\tilde{u}_i}$
$\hat{\boldsymbol{c}} = (\hat{c}_1, \ldots, \hat{c}_N)$
for $i = 1, \ldots, N$ **do**
  $\hat{\omega}_i \in_R \mathbb{Z}_q$, $\tilde{\omega}_i \in_R \mathbb{Z}_q$
  $\hat{t}_i \leftarrow g^{\hat{\omega}_i} \cdot \hat{c}_{i-1}^{\tilde{\omega}_i}$
$\omega_1 \in_R \mathbb{Z}_q$, $\omega_2 \in_R \mathbb{Z}_q$, $\omega_3 \in_R \mathbb{Z}_q$, $\omega_4 \in_R \mathbb{Z}_q$
$t_1 \leftarrow g^{\omega_1}$
$t_2 \leftarrow g^{\omega_2}$
$t_3 \leftarrow g^{\omega_3} \cdot \prod_{i=1}^{N} h_i^{\tilde{\omega}_i}$
$t_{4,1} \leftarrow pk^{-\omega_4} \cdot \prod_{i=1}^{N} \tilde{a}_i^{\tilde{\omega}_i}$
$t_{4,2} \leftarrow g^{-\omega_4} \cdot \prod_{i=1}^{N} \tilde{b}_i^{\tilde{\omega}_i}$
$t \leftarrow (t_1, t_2, t_3, (t_{4,1}, t_{4,2}), (\hat{t}_1, \ldots, \hat{t}_N))$
$c \leftarrow \mathsf{Hash}(\boldsymbol{e}, \tilde{\boldsymbol{e}}, \boldsymbol{c}, \hat{\boldsymbol{c}}, pk, t)$
$v_N \leftarrow 1$
for $i = N, \ldots, 1$ **do**
  $v_{i-1} \leftarrow \tilde{u}_i v_i \bmod q$
$r \leftarrow \sum_{i=1}^{N} r_i \bmod q$, $s_1 \leftarrow \omega_1 - c \cdot r \bmod q$
$\hat{r} \leftarrow \sum_{i=1}^{N} \hat{r}_i v_i \bmod q$, $s_2 \leftarrow \omega_2 - c \cdot \hat{r} \bmod q$
$\bar{r} \leftarrow \sum_{i=1}^{N} r_i u_i \bmod q$, $s_3 \leftarrow \omega_3 - c \cdot \bar{r} \bmod q$
$\tilde{r} \leftarrow \sum_{i=1}^{N} \tilde{r}_i u_i \bmod q$, $s_4 \leftarrow \omega_4 - c \cdot \tilde{r} \bmod q$
for $i = 1, \ldots, N$ **do**
  $\hat{s}_i \leftarrow \hat{\omega}_i - c \cdot \hat{r}_i \bmod q$, $\tilde{s}_i \leftarrow \tilde{\omega}_i - c \cdot \tilde{u}_i \bmod q$
$s \leftarrow (s_1, s_2, s_3, s_4, (\hat{s}_1, \ldots, \hat{s}_N), (\tilde{s}_1, \ldots, \tilde{s}_N))$
$\pi \leftarrow (t, s, \boldsymbol{c}, \hat{\boldsymbol{c}})$
return $\pi \in (\mathcal{G} \times \mathcal{G} \times \mathcal{G} \times \mathcal{G}^2 \times \mathcal{G}^N) \times (\mathbb{Z}_q \times \mathbb{Z}_q \times \mathbb{Z}_q \times \mathbb{Z}_q \times \mathbb{Z}_q^N \times \mathbb{Z}_q^N) \times \mathcal{G}^N \times \mathcal{G}^N$

Algorithm A.2: Generates an ElGamal shuffle proof.

Algorithm: $\mathsf{CheckProof}(\pi, \boldsymbol{e}, \tilde{\boldsymbol{e}}, pk)$

Input: Shuffle proof $\pi = (t, s, \boldsymbol{c}, \hat{\boldsymbol{c}})$
- $t = (t_1, t_2, t_3, (t_{4,1}, t_{4,2}), (\hat{t}_1, \ldots, \hat{t}_N)) \in \mathcal{G} \times \mathcal{G} \times \mathcal{G} \times \mathcal{G}^2 \times \mathcal{G}^N$
- $s = (s_1, s_2, s_3, s_4, (\hat{s}_1, \ldots, \hat{s}_N), (\tilde{s}_1, \ldots, \tilde{s}_N)) \in \mathbb{Z}_q \times \mathbb{Z}_q \times \mathbb{Z}_q \times \mathbb{Z}_q \times \mathbb{Z}_q^N \times \mathbb{Z}_q^N$
- $\boldsymbol{c} = (c_1, \ldots, c_N) \in \mathcal{G}^N$, $\hat{\boldsymbol{c}} = (\hat{c}_1, \ldots, \hat{c}_N) \in \mathcal{G}^N$

ElGamal ciphertextes $\boldsymbol{e} = (e_1, \ldots, e_N)$, $e_i = (a_i, b_i) \in \mathcal{G}^2$
Shuffled ElGamal ciphertexts $\tilde{\boldsymbol{e}} = (\tilde{e}_1, \ldots, \tilde{e}_N)$, $\tilde{e}_i = (\tilde{a}_i, \tilde{b}_i) \in \mathcal{G}^2$
Encryption key $pk \in \mathcal{G}$

for $i = 1, \ldots, N$ **do**
  $u_i \leftarrow \mathsf{Hash}((\boldsymbol{e}, \tilde{\boldsymbol{e}}, \boldsymbol{c}), i)$
$\hat{c}_0 \leftarrow h$
$\bar{c} \leftarrow \prod_{i=1}^{N} c_i / \prod_{i=1}^{N} h_i$
$u \leftarrow \prod_{i=1}^{N} u_i \bmod q$
$\hat{c} \leftarrow \hat{c}_N \cdot h^{-u}$
$\tilde{c} \leftarrow \prod_{i=1}^{N} c_i^{u_i}$
$\tilde{a} \leftarrow \prod_{i=1}^{N} a_i^{u_i}$
$\tilde{b} \leftarrow \prod_{i=1}^{N} b_i^{u_i}$
$c \leftarrow \mathsf{Hash}(\boldsymbol{e}, \tilde{\boldsymbol{e}}, \boldsymbol{c}, \hat{\boldsymbol{c}}, pk, t)$
for $i = 1, \ldots, N$ **do**
  $\hat{t}'_i \leftarrow \hat{c}_i^{\,c} \cdot g^{\hat{s}_i} \cdot \hat{c}_{i-1}^{\tilde{s}_i}$
$t'_1 \leftarrow \bar{c}^{\,c} \cdot g^{s_1}$
$t'_2 \leftarrow \hat{c}^{\,c} \cdot g^{s_2}$
$t'_3 \leftarrow \tilde{c}^{\,c} \cdot g^{s_3} \cdot \prod_{i=1}^{N} h_i^{\tilde{s}_i}$
$t'_{4,1} \leftarrow \tilde{a}^{c} \cdot pk^{-s_4} \cdot \prod_{i=1}^{N} \tilde{a}_i^{\tilde{s}_i}$
$t'_{4,2} \leftarrow \tilde{b}^{c} \cdot g^{-s_4} \cdot \prod_{i=1}^{N} \tilde{b}_i^{\tilde{s}_i}$
return

$$(t_1 = t'_1) \wedge (t_2 = t'_2) \wedge (t_3 = t'_3) \wedge (t_{4,1} = t'_{4,1}) \wedge (t_{4,2} = t'_{4,2}) \wedge \left[\bigwedge_{i=1}^{N} (\hat{t}_i = \hat{t}'_i)\right]$$

Algorithm A.3: Checks the correctness of an ElGamal shuffle proof.