MAYA: A Short Shuffle Argument With Fast Verification

Thi Van Thao Doan, Olivier Pereira, and Thomas Peters

UCLouvain, B-1348 Louvain-la-Neuve, Belgium

Abstract

Shuffle arguments account for the largest part of the audit data of end-to-end verifiable elections when simple homomorphic tallying cannot be applied. The mixnets that have been deployed in government or public elections (e.g., in Australia, Estonia, Israel, Norway, Spain or Switzerland), generate proofs with a size that scales linearly with the number of ciphertexts, and account for GB of data when millions of ciphertexts need to be shuffled.

We present MAYA, a shuffle argument with $O(\log n)$ communication complexity and a transparent setup requiring no trusted parameters, making the currently dominant cost of the shuffle argument a small fraction of the audit data size and verification time. Our construction instantiates a commitment-consistent shuffle framework using a succinct argument based on generalized k-ary folding that efficiently supports an arbitrary number of ciphertexts, while preserving the aggregation structure used in operational mixnet deployments.

We implement MAYA in Rust and compare it with Verificatum, the current state-of-the-art and heavily-optimized mixnet implementation. For $n = 10^6$ ciphertexts, our implementation produces arguments whose length is .002% of those of Verificatum, have a similar computation time, and require only 12% of the Verificatum argument verification time. Overall, MAYA considerably reduces the cost of hosting election verification data and speeds-up the election verification process, supporting effective election verification.

1 Introduction

End-to-end verifiable electronic voting systems aim to reconcile two fundamental requirements: privacy of individual votes and universal verifiability of the final tally. Mixnets, combined with publicly auditable bulletin boards (BBs), remain the predominant mechanism for achieving this goal. A central component of this audit trail is the *proof of shuffle*, which certifies that a mix server has correctly permuted and rerandomized the ciphertexts without altering, injecting, or removing votes.

Despite their strong security guarantees, existing mixnet deployments face a *scalability barrier*. In widely deployed systems, proof sizes and verification costs scale linearly in the number of ciphertexts. As elections approach national scale (e.g., $n \approx 10^6$ voters), these linear proofs become prohibitively large, placing severe strain on storage, bandwidth, and verification resources.

Verificatum [29], which is probably the mixnet implementation that has been the most largely used in government elections (used in Estonia, Israel, Norway and Spain for instance), exemplifies this. For a single shuffle of one million ECC256 ciphertexts, Verificatum produces a proof of approximately 304.2 MB (Section 4.3.3). And real-world deployments can bring substantially larger numbers: Estonia, for instance, uses a $\bmod p$ group with a 3072-bit prime p instead of ECC. In other applications, like STV, votes may need to be encrypted across multiple ciphertexts that need to be shuffled multiple times, after each round of candidate elimination. The shuffle-sum protocol [8] for STV tallying reports around 400GB of audit data and 10000h of verification time for 3 millions votes, essentially devoted to shuffle arguments. Recently, Swiss

Post, the provider of the Internet voting solution used in Switzerland, proposed with LORIA a new version of their protocol aiming at reducing trust in some elements of the system [12]. Their protocol requires 4 verifiable shuffles of a number of ciphertexts that is the product of the number of voters by the number of options on a ballot. For a single canton with around 100.000 voters and 100 options on the ballot, this requires 4 shuffles of 10^7 ciphertexts. This is a load that remains manageable on the proving side, as the proof computation is typically easy to parallelize and can be performed on multiple large workstations within the Swiss Post infrastructure. Proof verification, on the other hand, would be performed on the laptops of external auditors and require more than 5 hours of computation if we take numbers from Verificatum [30] for the P-256 curve, while these auditors typically need to operate under tight time constraints.

1.1 Related Work

The scalability limitations of linear-size proofs have motivated research into sub-linear shuffle arguments for almost 2 decades. The first sub-linear shuffle argument was proposed by Groth and Ishai in 2008 [21], achieving $O(\sqrt{n})$ communication complexity, at the cost of a prover computational complexity in $O(n^{3/2})$. Bayer and Groth [7] subsequently improved this to achieve practical efficiency with $O(\sqrt{n})$ proof size while maintaining linear prover time.

The Bulletproofs paper [11] outlined a shuffle argument of size $O(\log n)$ via recursive folding. However, their argument is designed for vector commitments of width n rather than lists of n ciphertexts, and the adaptation to the ciphertext setting is not direct because the ciphertexts are not computed w.r.t. bases that have unknown discrete logarithm relations, contrary to the ones used in vector commitments. Hoffmann et al. [22] addressed this gap with a *linear-map preimage argument* that proves knowledge of $\boldsymbol{a} \in \mathbb{Z}_p^n$ satisfying $\boldsymbol{G} \cdot \boldsymbol{a} = \boldsymbol{P}$ for an arbitrary public matrix $\boldsymbol{G} \in \mathbb{G}^{m \times n}$, achieving $O(\log n)$ communication via generalized k-ary folding. This argument is then the basis of a shuffle argument along the lines of the Bayer-Groth approach. Fauzi et al. proposed a pairing based shuffle argument of constant size [18]. However, soundness of their argument requires a structured common reference string (CRS) that needs to be generated in a trusted manner, which is something that is typically undesirable in an election setting: MAYA and all the other referenced solutions use a public-coin CRS.

The challenges of implementing shuffle arguments are important and, as far as we know, Verificatum [33] is by far the most complete and optimized implementation, so we use it as our base line for comparison with MAYA. As other examples, an implementation of the Bayer-Groth shuffle argument is maintained by Swiss Post,[1] and offer very slow running time compared to Verificatum, even accounting for the fact that it works in a 3072 bit MODP group. Hoffmann et al. [22] offer a proof of concept C++ implementation of their logarithmic argument, and deliver very low speed results, which they report as resulting from underlying libraries. Curdleproofs [28] also offers a logarithmic variant inspired from the Bayer-Groth protocol, but serving a different purpose: they show that a list of public keys $(g_i, g_i^{x_i})$ are all shuffled and exponentiated into $(g_i^k, g_i^{x_i k})$ for a unique secret k. Running times are also offered for an implementation of the constant-size argument of Fauzi et al.[18], but these are considerably slower than Verificatum, for proof computation and verification.

1.2 Our Contributions

We design, implement and benchmark MAYA, an efficient logarithmic-size shuffle argument with a transparent setup.

A comparison with the state-of-the-art Verificatum implementation [33] shows, for a million of ciphertexts (which is the point of reference chosen in the Verificatum benchmarks), that a MAYA argument has

[1] `https://gitlab.com/swisspost-evoting`

(i) a size of .002% of the one of Verificatum, (ii) a computation time almost equal to the one of Verificatum, and (iii) a verification time down to 9% of Verificatum. We also compare MAYA to the current best logarithmic-size argument [22]: our analysis shows (see Appendix I) that their approach is $\approx 43\%$ more demanding than MAYA for a million ciphertexts. Finally, MAYA efficiently handles the shuffling of lists of ciphertexts of arbitrary length, and not just powers of 2, addressing a technical hassle that led to important inefficiencies in the currently deployed implementation of the Bayer-Groth $O(\sqrt{n})$ complexity shuffle in the Swiss voting system, easing its integration within existing systems.

In order to achieve these results, we propose the following technical contributions.

(1) *Computational parameter optimization*: We optimize for computational throughput rather than proof size alone. Through extensive benchmarking, we identified the $k = 4$ folding factor as an effective operating point that reduces prover time by approximately 10% compared to standard binary fording ($k = 2$) by halving recursion depth while avoiding the arithmetic overhead of larger radices ($k \geq 8$).

(2) *Iterative padding with bounded overhead.* Standard k-ary folding requires padding inputs to k^d for some integer d, which can multiply the prover work by a factor k when the number of ciphertexts happens to be just over a power of k (e.g., $n = 2^{20} + 1$ for $k = 2, 4$). Instead of padding at the first layer only, we propose *iterative padding*, which pads incrementally at each recursion level to a multiple of k, bounding the total overhead to $O(\log_k n)$ additional elements regardless of how n aligns with powers of k.

(3) *Aggregated argument batching.* We apply batching techniques to inner product argument instances, which yields a $1.5\times$ reduction in prover time for the recursive phase compared to the non-batched baseline.

(4) *Bridged Consistency for Aggregated Commitments:* We provide a technique for verifying permutation-based arithmetic circuits using a single aggregated vector commitment, rather than the n separate commitments typically employed in succinct constraint systems [11, 27]. The construction relies on an auxiliary bridging argument that binds the circuit's linear constraints to the witness used in the ciphertext shuffle equations. This design addresses the soundness challenge of verifying hidden witness coordinates, allowing the construction to achieve $O(\log n)$ communication complexity without sacrificing the binding property of the commitment scheme.

(5) *Production-scale evaluation.* We provide two Rust instantiations: MAYA-Ristretto (Curve25519) and MAYA-P256 (P-256). For $n = 10^6$ ElGamal ciphertexts, MAYA (Ristretto) reduces proof size from 304.2 MB to 6.6 KB, verification time from 367 s to 32.7 s (single-threaded) and from 168.5 s to 17.1 s (multi-threaded), while the prover computation time is comparable to Verificatum, from 292.4 s to 248.3 s (single-threaded) and from 183.8 s to 120.3 s (multi-threaded), still in favor of MAYA.

By reducing proof size from hundreds of megabytes to kilobytes and verification time from minutes to seconds, our implementation removes the proof itself as a scalability bottleneck. While, in currently deployed mixnet based voting systems, shuffle arguments represent the large majority of the audit data, our proof of shuffle makes that argument negligible compared to the actual size of the ciphertexts.

1.3 MAYA at a Glance

MAYA makes a careful use of logarithmic-size inner product argument (IPA) and extended Chaum-Perdersen argument (eCP) from compressed sigma-protocols to make them smoothly compatible within the commitment-consistent shuffle framework due to Wikström. The underlying calls to IPA and eCP are optimized in two ways. First, we observe that random linear combinations can be applied to some internal equations, for which we need to prove satisfiability, to aggregate them without any computational loss in soundness

thanks to the usual Schwartz-Zippel lemma. Second, and more importantly, we improve these base sub-protocols with the iterative padding and early termination strategy. These optimizations are detailed in Section 2.5 and Section 2.6 in the preliminaries as they consist of an adaptation of existing protocols to which we keep the recursive k-ary folding structure [10, 11, 6].

The core description of MAYA is given in Section 3. The main steps follow [29] to ensure that the output ciphertexts $\mathbf{C}'$ are a re-encryption of a permutation of the input ciphertexts $\mathbf{C}$ as

$$C'^{\boldsymbol{e}'} \cdot (g, f)^{-r'} = C^{\mathrm{e}},$$

where $\boldsymbol{e}$ is a public challenge vector and $\boldsymbol{e}' = (e_{\pi(1)}, \ldots, e_{\pi(n)})$ encodes the secret permutation π. Prior Wikström-style shuffle arguments [29, 27] represent $\boldsymbol{e}'$ using n separate commitments while we encode it into a single vector commitment $V = h^s \cdot \boldsymbol{g}^{\boldsymbol{e}'}$. We then want to apply the commitment-consistent shuffle techniques to V. That is, (phase 1) $\boldsymbol{e}'$ is indeed a permutation of $\boldsymbol{e}$, and (phase 2) the list of input-output ciphertext $\mathbf{C}$ and $\mathbf{C}'$ follows the same order-relation as $\boldsymbol{e}$ and $\boldsymbol{e}'$, named ciphertext consistency. To apply these two phases, we have to overcome some difficulties that arise from the compressed representation V that makes it harder to evaluate linear functions over the individual coordinates (even with Bulletproof), and from the fact that ciphertexts do not form a list of independent generator basis for which no one knows a discrete logarithm relation, which prevents us from naively relying on standard folding techniques.

We deal with Phase 1 in Section 3.2, where we show the permutation relation by the standard polynomial identity technique [25, 20, 29], which can be reduced to checking that

$$\textstyle\prod_i (e'_i - u) = \prod_i (e_i - u)$$

for a random challenge u. Following [10, 11, 6] we express this condition as an arithmetic circuit satisfiability instance, compatible with IPA. To address the first difficulty, the prover first computes a bridging commitment to the linear terms that the verifier cannot evaluate. Then, one IPA proves circuit satisfiability with the newly committed terms, and another IPA certifies that V contains the appropriate $\boldsymbol{e}'$.

As for phase 2 that we develop in Section 3.3, we have to show that the same $\boldsymbol{e}'$ committed in V satisfies the aggregated ciphertext relation. To avoid providing a linear-size response for that proof, we first compress it by following eCP proof technique to show a *linear-map preimage* relation $\boldsymbol{P} = \boldsymbol{G}^{\boldsymbol{a}}$. Inspired by [22], we circumvent the second aforementioned difficulty by adding one-more *anchoring* row to $\boldsymbol{G}$ consisting of random generators to "erase" any knowledge of discrete logarithm relation. Our choice of these generators is simply the $\boldsymbol{g}$ used to compute V. Our final protocol manages to compute a single IPA and a single eCP using the batching techniques mentioned earlier.

In Section 4, we provide an experimental evaluation of MAYA for a variety of practical settings (single and multi-core) and various election sizes. We first analyse the impact of the folding parameter k and compare MAYA against Verificatum at national scale. We also study the effect of the padding strategies, the trade-off between speed and proof size, and deployment recommendations based on this analysis. Our conclusion is in Section 5 and all our detailed security proofs are provided in the appendices.

2 Preliminaries

2.1 Notation

Let $\mathbb{G}$ be a cyclic group of prime order p, and let $\mathbb{Z}_p$ denote the field of integers modulo p. We denote by $\mathbb{Z}_p^n$ the n-dimensional vector space over $\mathbb{Z}_p$, and by $\mathbb{G}^n$ the module of n-tuples over $\mathbb{Z}_p$ (or simply the set of vectors of length n in $\mathbb{G}$). Furthermore, let $\mathbb{Z}_p^*$ denote the multiplicative group of nonzero elements of $\mathbb{Z}_p$.

We use bold font to denote vectors and matrices. For a field $\mathbb{Z}_p$, we write $\boldsymbol{a} \in \mathbb{Z}_p^n$ for a vector with components $a_1, \ldots, a_n \in \mathbb{Z}_p$. For a group $\mathbb{G}$, we write $\boldsymbol{g} \in \mathbb{G}^n$ to denote an n-tuple of group elements.

Similarly, $\boldsymbol{A} \in \mathbb{Z}_p^{n \times m}$ denotes a matrix with n rows and m columns, and $a_{i,j}$ denotes the (i,j)-th entry of $\boldsymbol{A}$. For a scalar $c \in \mathbb{Z}_p$ and a vector $\boldsymbol{a} \in \mathbb{Z}_p^n$, we write $c \cdot \boldsymbol{a} \in \mathbb{Z}_p^n$ for component-wise scalar multiplication, i.e., $(c \cdot \boldsymbol{a})_i = c \cdot a_i$. When sampling uniformly at random, we write $x \xleftarrow{\$} \mathbb{Z}_p$ or $g \xleftarrow{\$} \mathbb{G}$, depending on the domain.

Given two vectors $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{Z}_p^n$, we define the inner product as $\langle \boldsymbol{a}, \boldsymbol{b} \rangle = \sum_{i=1}^n a_i \cdot b_i \in \mathbb{Z}_p$, and the Hadamard (component-wise) product as $\boldsymbol{a} \circ \boldsymbol{b} = (a_1 \cdot b_1, \ldots, a_n \cdot b_n) \in \mathbb{Z}_p^n$. For a vector of group elements $\boldsymbol{g} = (g_1, \ldots, g_n) \in \mathbb{G}^n$ and exponents $\boldsymbol{a} \in \mathbb{Z}_p^n$, we define the multi-exponentiation as $\boldsymbol{g}^{\boldsymbol{a}} = \prod_{i=1}^n g_i^{a_i} \in \mathbb{G}$. We write $\boldsymbol{P} = \boldsymbol{G}^{\boldsymbol{a}}$ for $\boldsymbol{G} \in \mathbb{G}^{m \times n}, \boldsymbol{P} \in \mathbb{G}^m, \boldsymbol{a} \in \mathbb{Z}_p^n$ to denote $P_i = \boldsymbol{g}_i^{\boldsymbol{a}}$ where each row of matrix $\boldsymbol{G}$ is a vector $\boldsymbol{g}_i \in \mathbb{G}^n$ with $i \in [m]$. For $k \in \mathbb{Z}_p^*$, we denote by $\boldsymbol{k}^n \in (\mathbb{Z}_p^*)^n$ the vector of the first n powers of k: $\boldsymbol{k}^n = (1, k, k^2, \ldots, k^{n-1})$.

We use the notation $\mathcal{R} = \{(\text{Public Input}; \text{Witness}) : \text{Relation}\}$ to denote a relation $\mathcal{R}$ defined over a given public input and private witness.

2.2 Assumptions

Let Gen be an algorithm that, on input 1^λ, outputs $(\mathbb{G}, p, g)$, where $\mathbb{G}$ is a cyclic group of prime order p with $|p| = \lambda$, and g is a generator of $\mathbb{G}$. The *Discrete Logarithm Relation* (DLR) assumption informally states that it is computationally infeasible to find a non-trivial algebraic relation $g^{a_0} \prod_{i=1}^n g_i^{a_i} = 1$ among independently sampled group elements $g, g_1, \ldots, g_n$. For polynomially bounded n (in λ), it is equivalent to the *Discrete Logarithm* (DL) assumption [11].

Definition 1 (Discrete Logarithm Relation (DLR) Assumption [11])**.** For all $n \geq 2$ and all non-uniform polynomial-time adversaries $\mathcal{A}$,

$$\Pr\left[\begin{array}{l}(\mathbb{G}, p, g) \leftarrow \mathsf{GGen}(1^\lambda);\ g_1, \ldots, g_n \xleftarrow{\$} \mathbb{G}; \\ a_0, \ldots, a_n \leftarrow \mathcal{A}(\mathbb{G}, p, g, \{g_i\}_i) : \\ \quad \exists i \text{ s.t. } a_i \neq 0 \text{ and } g^{a_0} \prod_{i=1}^n g_i^{a_i} = 1\end{array}\right] \approx 0.$$

2.3 Security Definitions

We consider an interactive argument system $\Pi = (\mathsf{Setup}, \mathcal{P}, \mathcal{V})$ for a polynomial-time decidable relation $\mathcal{R} \subseteq \{0,1\}^* \times \{0,1\}^*$. On input the security parameter 1^λ, the setup algorithm Setup outputs a common reference string (or public parameters) pp. Given pp, a statement x, and a witness w such that $(x, w) \in \mathcal{R}$, the prover $\mathcal{P}$ and verifier $\mathcal{V}$ interact to produce a transcript $tr \leftarrow \langle \mathcal{P}(pp, x, w), \mathcal{V}(pp, x) \rangle$. The verifier outputs a bit for acceptance or rejection. Our security definitions follow those established by Bünz et al. [11].

Definition 2 (Perfect Completeness)**.** An argument system Π has *perfect completeness* if for all non-uniform polynomial-time adversaries $\mathcal{A}$, the probability that an honest verifier rejects a valid proof generated by an honest prover is zero:

$$\Pr\left[\begin{array}{l|l}pp \leftarrow \mathsf{Setup}(1^\lambda); & (x, w) \in \mathcal{R} \Rightarrow \\ (x, w) \leftarrow \mathcal{A}(pp) & \langle \mathcal{P}(pp, x, w), \mathcal{V}(pp, x) \rangle = 1\end{array}\right] = 1.$$

Definition 3 (Computational Witness Extended Emulation)**.** An interactive argument system Π has *computational witness extended emulation (WEE)* if for every deterministic polynomial-time prover $\mathcal{P}^*$, there exists an expected polynomial-time emulator $\mathcal{E}$ such that for all non-uniform polynomial-time adversaries $\mathcal{A}$, the following condition holds: Let the adversary $\mathcal{A}$ generate a statement x and state st based on the

public parameters $pp \leftarrow \mathsf{Setup}(1^\lambda)$: $(x, st) \leftarrow \mathcal{A}(pp)$. We require that the probability difference between the real view and the emulated view is negligible:

$$\left| \Pr[\mathcal{A}(\mathsf{tr}) = 1 \mid \mathsf{tr} \leftarrow \langle \mathcal{P}^*(x, st), \mathcal{V}(x) \rangle] - \Pr[\mathcal{A}(\mathsf{tr}) = 1 \mid (\mathsf{tr}, w) \leftarrow \mathcal{E}^{O}(x)] \right| \leq \mu(\lambda),$$

where the emulator $\mathcal{E}$ has oracle access O to rewinding $\mathcal{P}^*$. Furthermore, if the emulated transcript tr is accepting, the extracted witness w must be valid for the statement x (i.e., $(x, w) \in \mathcal{R}$) with all but negligible probability. We use witness-extended emulation to define knowledge-soundness, following the formalisms in [21, 24] and the methodology in [10, 11].

Definition 4 (Public-Coin)**.** An interactive argument system Π is *public coin* if all messages sent by the verifier are chosen uniformly at random and independently of the prover's messages. We denote the verifier's randomness by ρ.

Definition 5 (Perfect SHVZK)**.** A public-coin interactive argument Π is *Perfect Special Honest Verifier Zero Knowledge (SHVZK)* if there exists a probabilistic polynomial-time simulator $\mathcal{S}$ that can reproduce the view of the protocol without access to the witness. Formally, for all adversaries $\mathcal{A}$ choosing valid statement-witness pairs, and for any set of fixed verifier challenges ρ:

$$\Pr\left[\begin{array}{l} pp \leftarrow \mathsf{Setup}(1^\lambda); \\ (x, w) \leftarrow \mathcal{A}(pp) \\ \text{s.t. } (x, w) \in \mathcal{R} \end{array} : \begin{array}{l} \mathcal{S}(pp, x, \rho) = \\ \langle \mathcal{P}(pp, x, w), \mathcal{V}(pp, x; \rho) \rangle \end{array}\right] = 1.$$

2.4 Cryptographic Primitives

Pedersen Commitments. A non-interactive commitment scheme consists of a pair of probabilistic polynomial-time (PPT) algorithms $(\mathsf{Gen}_{\mathsf{ck}}, \mathsf{Com}_{\mathsf{ck}})$. The key generation algorithm $\mathsf{Gen}_{\mathsf{ck}}(1^\lambda)$ outputs a public commitment key ck, which specifies a message space $\mathcal{M}_{\mathsf{ck}}$, randomness space $\mathcal{R}_{\mathsf{ck}}$, and commitment space C_{ck}. The commitment algorithm is a function $\mathsf{Com}_{\mathsf{ck}} : \mathcal{M}_{\mathsf{ck}} \times \mathcal{R}_{\mathsf{ck}} \to C_{\mathsf{ck}}$.

Let $(\mathbb{G}, p, g)$ be a cyclic group of prime order p generated by g. Given an additional generator $h \in \mathbb{G}$, the *Pedersen commitment* to a message $m \in \mathbb{Z}_p$ is $c = \mathsf{Com}_{\mathsf{ck}}(m; r) = h^r g^m \in \mathbb{G}$, where $r \xleftarrow{\$} \mathbb{Z}_p$. For commitments to vectors $\boldsymbol{m} \in \mathbb{Z}_p^n$, the key is $\mathsf{ck} = (\mathbb{G}, p, h, g_1, \ldots, g_n)$, where $h, g_1, \ldots, g_n \xleftarrow{\$} \mathbb{G}$ are sampled independently. The commitment is:

$$c = \mathsf{Com}_{\mathsf{ck}}(\boldsymbol{m}; r) = h^r \cdot \boldsymbol{g}^{\boldsymbol{m}} = h^r \cdot \textstyle\prod_{i=1}^n g_i^{m_i} \in \mathbb{G}.$$

Pedersen commitments are *perfectly hiding* and *computationally binding* under the DLR assumption. They satisfy an additive homomorphism: for all $m_0, m_1 \in \mathbb{Z}_p$ and $r_0, r_1 \in \mathbb{Z}_p$,

$$\mathsf{Com}_{\mathsf{ck}}(m_0; r_0) \cdot \mathsf{Com}_{\mathsf{ck}}(m_1; r_1) = \mathsf{Com}_{\mathsf{ck}}(m_0 + m_1; r_0 + r_1).$$

Homomorphic and Randomizable Encryption. Informally, an encryption scheme is said to be homomorphic if, for a public key pk, messages M_1, M_2, and randomness r_1, r_2, the encryption algorithm guarantees $\mathsf{Enc}(\mathsf{pk}, M_1 M_2; r_1 + r_2) = \mathsf{Enc}(\mathsf{pk}, M_1; r_1) \cdot \mathsf{Enc}(\mathsf{pk}, M_2; r_2)$. Furthermore, the scheme is randomizable if there exists an algorithm Rand that takes a ciphertext and fresh randomness r' to produce a new ciphertext distributed identically to a fresh encryption of the same message with updated randomness. Our analysis applies to different homomorphic encryption schemes over message spaces of large prime order p. For clarity, we focus on the ElGamal encryption scheme.

ElGamal Encryption. We recall the ElGamal encryption scheme [17]. $\mathsf{Gen}(1^\lambda)$ generates a cyclic group $(\mathbb{G}, p, g)$ with $|p| = \lambda$, samples $x \xleftarrow{\$} \mathbb{Z}_p$, sets $f = g^x$, and returns $(\mathsf{pk} = \{g, f\}, \mathsf{sk} = x)$. $\mathsf{Enc}(\mathsf{pk}, M; r)$ returns $\boldsymbol{C} = (C_1, C_2) = (g^r, M \cdot f^r) \in \mathbb{G}^2$. $\mathsf{Dec}(\mathsf{sk}, \boldsymbol{C})$ returns $M = C_2 \cdot C_1^{-x}$. Eventually, $\mathsf{Rand}(\mathsf{pk}, \boldsymbol{C}; r')$ returns $\boldsymbol{C}' = \boldsymbol{C} \cdot (g, f)^{r'} = (C_1 \cdot g^{r'},\ C_2 \cdot f^{r'})$.

When handling multiple ciphertexts $\{\boldsymbol{C}_i\}_{i\in[n]}$, we write $\boldsymbol{C}_i = (C_{i1}, C_{i2})$, and denote by $\boldsymbol{c}_1 = (C_{11}, \ldots, C_{n1})$ and $\boldsymbol{c}_2 = (C_{12}, \ldots, C_{n2})$ the vectors of first and second ciphertext components, respectively.

2.5 Inner Product Argument with Padding Strategies

We review the succinct inner product argument (IPA) introduced by Bünz et al. [11], which allows a prover to demonstrate knowledge of vectors $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{Z}_p^n$ satisfying a single inner-product constraint. In the context of MAYA, this protocol provides the mathematical foundation for the Permutation Phase (Section 3.2). Formally, the IPA protocol is an argument of knowledge for the relation

$$\mathcal{R}_{\mathsf{IPA}} = \left\{ \begin{array}{l} (\boldsymbol{g}, \boldsymbol{h} \in \mathbb{G}^n, u, P \in \mathbb{G}, c \in \mathbb{Z}_p; \quad \boldsymbol{a}, \boldsymbol{b} \in \mathbb{Z}_p^n) : \\ P = \boldsymbol{g}^{\boldsymbol{a}} \boldsymbol{h}^{\boldsymbol{b}} u^c \wedge c = \langle \boldsymbol{a}, \boldsymbol{b} \rangle \end{array} \right\}, \tag{1}$$

where the discrete logarithm relations among $\boldsymbol{g}, \boldsymbol{h}$, and u are assumed to be unknown. We denote the interactive execution between a prover $\mathcal{P}$ and verifier $\mathcal{V}$ by

$$\mathcal{P} \leftrightarrow \mathcal{V} : \mathsf{IPA}(\boldsymbol{g}, \boldsymbol{h}, P, c\ ;\ \boldsymbol{a}, \boldsymbol{b}).$$

A naïve proof would send both witness vectors ($2n$ field elements). The key insight, due to Bootle et al. [10] and refined by Bünz et al. [11], is that the commitment P and the generators $(\boldsymbol{g}, \boldsymbol{h})$ can be *folded* together, replacing an n-dimensional claim by an $(n/2)$-dimensional one at the cost of transmitting just 2 group elements. Consider $n = 4$, and write the witness vectors as $\mathbf{a} = (a_1, a_2, a_3, a_4)$ and $\mathbf{b} = (b_1, b_2, b_3, b_4)$. The prover splits each vector into a left and right half, $\mathbf{a}_L = (a_1, a_2)$ and $\mathbf{a}_R = (a_3, a_4)$, doing the same for $\mathbf{b}$ and the generator vectors $\mathbf{g}, \mathbf{h}$. The prover then sends two cross-term commitments, L and R, whose exponents encode the "off-diagonal" contributions $\langle \mathbf{a}_L, \mathbf{b}_R \rangle$ and $\langle \mathbf{a}_R, \mathbf{b}_L \rangle$. The verifier replies with a uniformly random challenge $x \in \mathbb{Z}_p^*$. Both parties then fold: $\mathbf{a}' = x \cdot \mathbf{a}_L + x^{-1} \cdot \mathbf{a}_R, \quad \mathbf{b}' = x^{-1} \cdot \mathbf{b}_L + x \cdot \mathbf{b}_R$, $\mathbf{g}' = \mathbf{g}_L^{x^{-1}} \circ \mathbf{g}_R^{x}, \quad \mathbf{h}' = \mathbf{h}_L^{x} \circ \mathbf{h}_R^{x^{-1}}$. After folding, one verifies the identity $P' = \mathbf{g}'^{\mathbf{a}'} \mathbf{h}'^{\mathbf{b}'} u^{\langle \mathbf{a}', \mathbf{b}' \rangle}$, where $P' = L^{x^2} \cdot P \cdot R^{x^{-2}}$. This results in the exact same algebraic structure, but now in dimension $n/2 = 2$. Any inconsistency, by the Schwartz–Zippel argument, yields a non-trivial discrete-log relation among the generators.

Repeating this halving process $\log_2 n$ times reduces the claim to a single pair of scalars, which the prover reveals directly. The total proof consists of $2 \log_2 n$ group elements (one L and one R per round) plus two final scalars, exponentially smaller than the $2n$ field elements of the naïve proof.

2.5.1 Generalized k-ary Folding.

While the standard IPA uses binary folding, where the statement is recursively halved at each round, the protocol can be generalized to a k-ary split-and-fold mechanism [22] for any folding factor $k \geq 2$. At each recursion round, the prover splits the witness vectors into k equal-sized chunks and transmits $2(k-1)$ cross-term commitments. This reduces the recursion depth from $\log_2 n$ to $\log_k n$ at the cost of increased communication per round. We denote this variant by $\mathcal{P} \leftrightarrow \mathcal{V} : \mathsf{IPA}_k(\boldsymbol{g}, \boldsymbol{h}, P, c\ ;\ \boldsymbol{a}, \boldsymbol{b})$. The binary protocol used in the original construction is recovered as the special case $k = 2$.

2.5.2 The Padding Challenge.

Standard folding arguments require the input size n to satisfy $n = k^d$ for some integer d. When inputs do not align with this structure, the standard approach is to pad vectors to the next power of k at initialization (e.g., [11, 22]). In the worst case, when $n = k^{d-1}+1$, this forces padding to $n_{\text{padded}} = k^d \approx k \cdot n$, resulting in an important computational and memory overhead, especially when k increases. For national-scale elections with $n \approx 10^6$ votes, such overhead can be substantial. To address this, we develop two complementary strategies.

1. *Early Termination*: Standard recursive arguments (e.g., Bulletproofs with $k = 2$) terminate when $n = 1$, sending a single witness element. We modify this by introducing a base-case parameter $m \geq 1$. The protocol stops recursing when $n \leq m$, at which point the prover transmits the remaining (at most) m witness elements directly. This serves two purposes. First, it reduces structural overhead. Since each recursion round incurs fixed costs (e.g., challenge generation, cross-term computation, etc.), terminating at $m = k^2$ or $m = k^3$ eliminates several rounds at the cost of transmitting $O(m)$ instead of $O(1)$ final elements. Second, it enables *flexible input sizes* by varying m. The protocol can accommodate any input size n without forcing strict alignment to k^d. The parameter m thus enables a tunable trade-off between proof size (linear in m) and recursion depth (logarithmic in n/m). We evaluate this trade-off experimentally in Section 4.4.

2. *Iterative Padding*: Rather than padding the entire input to $n_{\text{padded}} = k^d$ at initialization, so-called *Global Pre-Padding*, we pad *incrementally* at each recursion step. If the current vector size is not divisible by k, we append just enough identity elements to make it divisible. Over $d = \log_k n$ rounds, the total padding overhead is bounded by $(k-1) \cdot \log_k(n)$ elements. While iterative padding has been informally suggested in community discussions [1], our contribution provides a systematic treatment: we formally define the technique and analyze its effect on computational cost (Section 2.5.3); present empirical benchmarks (Section 4.4); and provide security proofs within the context of k-ary shuffle arguments (Appendix B).

2.5.3 IPA with Iterative Padding.

We present the complete k-ary IPA protocol incorporating early termination, denoted $\text{IPA}_{k,m}$. The global pre-padding variant, which pads to $n_{\text{padded}} = k^d \cdot m$ at initialization, is deferred to Appendix A as it is primarily of theoretical interest for understanding worst-case behavior (Section 4.4).

Communication Complexity. Let $d = \lceil \log_k(n/m) \rceil$ denote the recursion depth. The protocol transmits $(2k-2) \cdot d$ group elements (cross-term commitments per round) and at most $2m$ field elements (final witness vectors in the base case). The total communication is $O(k \cdot \log_k(n/m))$ group elements plus $O(m)$ field elements.

Theorem 1. *Protocol 2.5 for the relation* (1) *has perfect completeness and computational witness-extended emulation, assuming the hardness of the Discrete Logarithm Relation (DLR) assumption.*

The proof for Theorem 1 is given in Appendix B.

Batch Optimization. To minimize communication overhead when proving multiple independent statements, we leverage the additive homomorphism of Pedersen commitments. By taking random linear combinations of μ instances, we aggregate verification into a single IPA execution. This amortizes both communication and computation cost. In particular, proof size reduces from $O(\mu \log n)$ to $O(\mu + \log n)$ group elements; prover and verifier work in the recursive phase reduces by a factor of μ, as both parties execute

Public Input: $\boldsymbol{g}, \boldsymbol{h} \in \mathbb{G}^n$, $u, P \in \mathbb{G}$, $c \in \mathbb{Z}_p$
Prover's Witness: $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{Z}_p^n$ such that $P = \boldsymbol{g}^{\boldsymbol{a}} \boldsymbol{h}^{\boldsymbol{b}} u^c$ and $c = \langle \boldsymbol{a}, \boldsymbol{b} \rangle$
Parameters: Folding factor $k \geq 2$, base case size $m \geq 1$

1. Base Case ($n \leq m$):
$\mathcal{P} \to \mathcal{V}$: Sends $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{Z}_p^n$.
$\mathcal{V}$ checks if $P = \boldsymbol{g}^{\boldsymbol{a}} \boldsymbol{h}^{\boldsymbol{b}} u^{\langle \boldsymbol{a}, \boldsymbol{b} \rangle}$, accept; otherwise, reject.

2. Recursion ($n > m$):
(a) **Iterative Padding:** If $n \not\equiv 0 \pmod{k}$:

- Compute $r = n \bmod k$ and extend:

$$\boldsymbol{g} \leftarrow \boldsymbol{g} \| \boldsymbol{g}_{\text{pad}},\ \boldsymbol{h} \leftarrow \boldsymbol{h} \| \boldsymbol{h}_{\text{pad}},\ \boldsymbol{a} \leftarrow \boldsymbol{a} \| \boldsymbol{0}^{k-r},\ \boldsymbol{b} \leftarrow \boldsymbol{b} \| \boldsymbol{0}^{k-r}$$

  where $\boldsymbol{g}_{\text{pad}}, \boldsymbol{h}_{\text{pad}} \in \mathbb{G}^{k-r}$ are derived deterministically from public parameters (e.g., via hash-to-curve from the current transcript).

- Update $n \leftarrow n + (k - r)$.

(b) Let $n' = n/k$. Partition vectors into k contiguous chunks of size n':

$$\boldsymbol{g} = (\boldsymbol{g}_1, \ldots, \boldsymbol{g}_k) \text{ where } \boldsymbol{g}_i = (g_{(i-1)n'+1}, \ldots, g_{in'})$$

and similarly for $\boldsymbol{h}, \boldsymbol{a}, \boldsymbol{b}$.
(c) $\mathcal{P}$ computes cross terms for $\ell \in \{-(k-1), \ldots, k-1\} \setminus \{0\}$:

$$U'_\ell = \prod_{j-i=\ell} \boldsymbol{g}_j^{\boldsymbol{a}_i}, \quad U''_\ell = \prod_{j-i=\ell} \boldsymbol{h}_j^{\boldsymbol{b}_i}, \quad v_\ell = \sum_{j-i=\ell} \langle \boldsymbol{a}_i, \boldsymbol{b}_j \rangle$$

Then computes: $U_\ell = U'_\ell \cdot U''_{-\ell} \cdot \boldsymbol{u}^{v_\ell}$.
(d) $\mathcal{P} \to \mathcal{V}$: Sends $\{U_\ell\}_{\ell \neq 0}$ (note: $U_0 = P$).
(e) $\mathcal{V} \to \mathcal{P}$: Sample challenge $x \xleftarrow{\$} \mathbb{Z}_p^*$.
(f) Both parties fold generators and commitment:

$$\boldsymbol{g}' = \prod_{i=1}^{k} \boldsymbol{g}_i^{x^{k-i}},\ \boldsymbol{h}' = \prod_{i=1}^{k} \boldsymbol{h}_i^{x^{i-1}},\ P' = P^{x^{k-1}} \cdot \prod_{\ell \neq 0} U_\ell^{x^{k-\ell-1}}$$

(g) $\mathcal{P}$ folds witnesses: $\boldsymbol{a}' = \sum_{i=1}^{k} x^{i-1} \boldsymbol{a}_i, \quad \boldsymbol{b}' = \sum_{i=1}^{k} x^{k-i} \boldsymbol{b}_i$.
(h) Recurse on $\mathsf{IPA}_{k,m}(\boldsymbol{g}', \boldsymbol{h}', P', c'; \boldsymbol{a}', \boldsymbol{b}')$ where $c' = \langle \boldsymbol{a}', \boldsymbol{b}' \rangle$.

Figure 1: Protocol 2.5 ($\mathsf{IPA}_{k,m}$): Inner Product Argument with Iterative Padding.

a single folding sequence rather than μ independent ones. The formal specification and security proof are provided in Appendix C.

2.6 Extended Chaum-Pedersen Argument

We present the *Extended Chaum-Pedersen Argument* (eCP), enabling a prover $\mathcal{P}$ to convince a verifier $\mathcal{V}$ of knowledge of a witness $\boldsymbol{a} \in \mathbb{Z}_p^n$ satisfying a linear-map preimage relationship. This argument will serve as the core of MAYA's Ciphertext Consistency Phase (Section 3.3). Formally, the eCP protocol proves the relation

$$\mathcal{R}_{\mathsf{eCP}} = \{(\boldsymbol{G} \in \mathbb{G}^{\gamma \times n}, \boldsymbol{P} \in \mathbb{G}^{\gamma}; \boldsymbol{a} \in \mathbb{Z}_p^n) : \boldsymbol{P} = \boldsymbol{G}^{\boldsymbol{a}}\}. \tag{2}$$

The protocol follows the same recursive k-ary split-and-fold paradigm as the inner-product argument $\mathsf{IPA}_{k,m}$ (Section 2.5). Similarly, we apply iterative padding: at each recursion step, if the current witness size is not divisible by k, we extend $\boldsymbol{a}$ with zeros and $\boldsymbol{G}$ with identity elements to the nearest multiple of k. We write $\mathcal{P} \leftrightarrow \mathcal{V} : \mathsf{eCP}_{k,m}(\boldsymbol{G}, \boldsymbol{P}; \boldsymbol{a})$.

Communication Complexity. Let $d = \lceil \log_k(n/m) \rceil$ denote the recursion depth and γ denote the number of simultaneous relations (i.e., the number of rows in $\boldsymbol{G}$). The protocol transmits $\gamma \cdot (2k-2) \cdot d$ group elements (cross-term commitments $\{\boldsymbol{U}_\ell\}_{\ell \neq 0}$ per round, with γ elements per commitment vector) and at most m field elements (final witness vector in the base case). The total communication is $O(\gamma \cdot k \cdot \log_k(n/m))$ group elements plus $O(m)$ field elements. Note that while logarithmic in the witness size n, the communication remains linear in the number of simultaneous relations γ.

Padding Overhead. The cumulative padding is bounded by $(k-1) \cdot d = O(\log(n/m))$. For the generator matrix $\boldsymbol{G}$, each padding step appends at most $k-1$ identity columns, incurring negligible computational cost due to sparse representation.

Theorem 2. *Protocol 2.6 for the relation* (2) *has perfect completeness and computational witness-extended emulation, assuming the hardness of the Discrete Logarithm Relation (DLR) assumption.*

The proof for Theorem 2 is given in Appendix D.

Batch Verification and Soundness. To improve verification efficiency, we would like to aggregate the γ simultaneous relations $\{P_i = \boldsymbol{g}_i^{\boldsymbol{a}}\}_{i=1}^{\gamma}$ (where $\boldsymbol{g}_i$ denotes the i-th row of $\boldsymbol{G}$) into a single check via random linear combination. However, as established by Hoffmann et al. [22], batching linear equations over potentially adversarial generators may compromise soundness if a malicious prover can adapt the witness to the batching randomness.

To mitigate this, we rely on the *commitment extending technique* [22]: we explicitly require that at least one row of $\boldsymbol{G}$, without loss of generality, the first row $\boldsymbol{g}_1$, consists of independently sampled generators serving as a globally binding commitment key. By ensuring the prover commits to $\boldsymbol{a}$ via $P_1 = \boldsymbol{g}_1^{\boldsymbol{a}}$ *prior* to the generation of the batching challenge, we anchor the witness. As a result, the validity of the batched combination implies the validity of all γ original equations with overwhelming probability, relying on the hardness of the DLR assumption in $\boldsymbol{g}_1$ and the Schwartz-Zippel lemma.

This optimization reduces the verification workload from γ distinct multi-scalar multiplications to a single combined check, and crucially reduces the proof size from $O(\gamma \cdot k \cdot \log_k(n/m))$ to $O(k \cdot \log_k(n/m))$ group elements, independent of the number of relations γ. The formal specification of the batched eCP protocol, including proofs of completeness and witness-extended emulation, is provided in Appendix E.

Public Input: $G \in \mathbb{G}^{\gamma \times n}$, $\boldsymbol{P} \in \mathbb{G}^{\gamma}$
Prover's Witness: $\boldsymbol{a} \in \mathbb{Z}_p^n$ such that $\boldsymbol{P} = \boldsymbol{G}^{\boldsymbol{a}}$
Parameters: Folding factor $k \geq 2$, base case size $m \geq 1$

1. Base Case ($n \leq m$):
$\mathcal{P} \to \mathcal{V}$: Sends $\boldsymbol{a} \in \mathbb{Z}_p^n$.
$\mathcal{V}$ checks if $\boldsymbol{P} = \boldsymbol{G}^{\boldsymbol{a}}$, accept; otherwise, reject.

2. Recursion ($n > m$):
(a) **Iterative Padding:** If $n \not\equiv 0 \pmod{k}$:

- Compute $r = n \bmod k$ and extend:

$$\boldsymbol{a} \leftarrow \boldsymbol{a} \| \boldsymbol{0}^{k-r}, \quad \boldsymbol{G} \leftarrow \boldsymbol{G} \| \boldsymbol{G}_{\text{pad}}$$

  where $\boldsymbol{G}_{\text{pad}} \in \mathbb{G}^{\gamma \times (k-r)}$ consists of deterministically derived generators.

- Update $n \leftarrow n + (k - r)$.

(b) Let $n' = n/k$. Partition $\boldsymbol{a}$ into k contiguous chunks of size n':

$$\boldsymbol{a} = (\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k) \text{ where } \boldsymbol{a}_i = (a_{(i-1)n'+1}, \ldots, a_{in'}) \in \mathbb{Z}_p^{n'}$$

Similarly, partition $\boldsymbol{G}$ column-wise into k blocks $\boldsymbol{G}_1, \ldots, \boldsymbol{G}_k \in \mathbb{G}^{\gamma \times n'}$.
(c) $\mathcal{P}$ computes cross-term commitments for $\ell \in \{-(k-1), \ldots, k-1\} \setminus \{0\}$:

$$\boldsymbol{U}_\ell = \prod_{j-i=\ell} \boldsymbol{G}_j^{\boldsymbol{a}_i} \in \mathbb{G}^{\gamma}$$

where indices i, j range over $\{1, \ldots, k\}$.
(d) $\mathcal{P} \to \mathcal{V}$: Sends $\{\boldsymbol{U}_\ell\}_{\ell \neq 0}$ (note: $\boldsymbol{U}_0 = \boldsymbol{P}$).
(e) $\mathcal{V} \to \mathcal{P}$: Sample challenge $x \xleftarrow{\$} \mathbb{Z}_p^*$.
(f) Both parties fold generators and statement:

$$\boldsymbol{G}' = \prod_{i=1}^{k} \boldsymbol{G}_i^{x^{k-i}}, \quad \boldsymbol{P}' = \boldsymbol{P}^{x^{k-1}} \cdot \prod_{\ell \neq 0} \boldsymbol{U}_\ell^{x^{k-\ell-1}}$$

(g) $\mathcal{P}$ folds witness: $\boldsymbol{a}' = \sum_{i=1}^{k} x^{i-1} \boldsymbol{a}_i \in \mathbb{Z}_p^{n'}$.
(h) Recurse on $\mathsf{eCP}_{k,m}(\boldsymbol{G}', \boldsymbol{P}'; \boldsymbol{a}')$.

Figure 2: Protocol 2.6 ($\mathsf{eCP}_{k,m}$): Extended Chaum-Pedersen Argument with Iterative Padding.

3 The MAYA Shuffle Argument

We construct a public-coin argument system for the relation $\mathcal{R}_{\text{shuffle}}$, and obtain a non-interactive zero-knowledge (NIZK) argument in the random oracle model via the Fiat–Shamir transformation.

We operate within the cryptographic setting defined in Section 2. Let $(\mathbb{G}, p, g)$ be the cyclic group parameters. The public statement consists of ElGamal public key $\mathsf{pk} = \{g, f\}$, an input ciphertext vector $\boldsymbol{C} = (\boldsymbol{C}_1, \ldots, \boldsymbol{C}_n)$ and an output ciphertext vector $\boldsymbol{C}' = (\boldsymbol{C}'_1, \ldots, \boldsymbol{C}'_n)$, where each $\boldsymbol{C}_i \in \mathbb{G}^2$.

The witness consists of a permutation $\pi \in S_n$ and a vector of rerandomization scalars $\boldsymbol{r} = (r_1, \ldots, r_n) \in \mathbb{Z}_p^n$. The validity of the shuffle is defined by the requirement that for all $i \in [n]$:

$$\boldsymbol{C}'_i = \boldsymbol{C}_{\pi(i)} \cdot (g, f)^{r_i}, \tag{3}$$

where $(g, f)^r := (g^r, f^r)$ and π maps the index of an input ciphertext to its corresponding output index (i.e., $\boldsymbol{C}'$ is a rerandomized permutation of $\boldsymbol{C}$). Formally, the shuffle relation is:

$$\mathcal{R}^*_{\text{shuffle}} = \left\{ \left(\mathsf{pk}, \boldsymbol{C}, \boldsymbol{C}'; \pi, \boldsymbol{r}\right) \ : \ (3) \text{ holds} \right\}.$$

3.1 Protocol Construction

We employ the verification aggregation technique of Wikström [29] reducing the verification of the complex shuffle relation to a single aggregated equation checked against a committed permutation.

Remark 1 (Generality of Ciphertext Structure)**.** *While we present the construction using ElGamal ciphertexts $\boldsymbol{C}_i = (C_{i,1}, C_{i,2}) \in \mathbb{G}^2$ for concreteness, the protocol applies to any additively homomorphic and re-randomizable encryption scheme whose ciphertexts are tuples of group elements $\boldsymbol{C}_i \in \mathbb{G}^m$ for fixed $m \geq 2$, for instance, Multi-ElGamal (Section 3.4), the commitment consistent encryption used in PPAT-based constructions [13], and CPA ciphertexts extracted from threshold re-encryption schemes [15, 16]. The aggregated verification equation* (4) *below generalizes naturally: instead of two component equations, we obtain m equations of the form $g_k^{-r'} \cdot \prod_i C'^{e'_i}_{i,k} = \prod_i C^{e_i}_{i,k}$ for $k \in [m]$, all sharing the same witness $\boldsymbol{e}'$. The eCP argument (Section 2.6) handles this by operating on an $(m+1)$-row matrix of bases. Our benchmarks use standard ElGamal.*

The protocol proceeds as follows:

1. The verifier samples a random challenge vector $\boldsymbol{e} = (e_1, \ldots, e_n) \xleftarrow{\$} \mathbb{Z}_p^n$.
2. The prover computes a permuted challenge vector $\boldsymbol{e}' \in \mathbb{Z}_p^n$ such that $\boldsymbol{e}' = \pi(\boldsymbol{e})$, where $e'_i = e_{\pi(i)}$. The prover also computes an aggregated randomness scalar $r' = \langle \boldsymbol{e}', \boldsymbol{r} \rangle \in \mathbb{Z}_p$.
3. The prover commits to $\boldsymbol{e}'$ using the Pedersen vector commitment scheme defined in Section 2.4. Given the expanded commitment key $\mathsf{ck} = (g, h, \boldsymbol{g}, \boldsymbol{h})$ with $\boldsymbol{g} = \{g_1, \ldots, g_n\} \in \mathbb{G}^n$ and $\boldsymbol{h} = \{h_1, \ldots, h_n\} \in \mathbb{G}^n$, the commitment is defined by $V = h^s \cdot \boldsymbol{g}^{\boldsymbol{e}'} \in \mathbb{G}$, where $s \xleftarrow{\$} \mathbb{Z}_p$ is a blinding factor.
4. The core of the argument is to prove that the permuted exponents $\boldsymbol{e}'$ (committed in V) satisfy the aggregated ciphertext relation:
$$(g, f)^{-r'} \cdot \boldsymbol{C}'^{\boldsymbol{e}'} = \boldsymbol{C}^{\boldsymbol{e}}. \tag{4}$$

This reduces the protocol to a proof of knowledge for the derived relation $\mathcal{R}_{\text{shuffle}}$ defined as:

$$\mathcal{R}_{\text{shuffle}} = \left\{ \begin{array}{l} (\mathsf{pk}, \mathsf{ck}, \boldsymbol{C}, \boldsymbol{C}', V, \boldsymbol{e}; \boldsymbol{e}', r', s) : \\ V = h^s \cdot \boldsymbol{g}^{\boldsymbol{e}'} \ \wedge \ \boldsymbol{e}' \text{ is a permutation of } \boldsymbol{e} \\ \wedge \ (4) \text{ holds} \end{array} \right\} \tag{5}$$

Let the publicly computable right-hand side of Eq. 4 be denoted by $\boldsymbol{C} = (C_1, C_2) = \boldsymbol{C}^{\boldsymbol{e}}$. We define the component vectors of the shuffled ciphertexts as $\boldsymbol{c}'_1 = (C'_{11}, \ldots, C'_{n1})$ and $\boldsymbol{c}'_2 = (C'_{12}, \ldots, C'_{n2})$. The aggregate verification equations are:

$$\begin{aligned} C'_1 &= g^{-r'} \cdot \boldsymbol{c}_1'^{\boldsymbol{e}'} = g^{-r'} \cdot \prod_{i=1}^{n} {C'_{i1}}^{e'_i} = C_1, \\ C'_2 &= f^{-r'} \cdot \boldsymbol{c}_2'^{\boldsymbol{e}'} = f^{-r'} \cdot \prod_{i=1}^{n} {C'_{i2}}^{e'_i} = C_2. \end{aligned} \tag{6}$$

This formulation allows us to decompose the proof into two simultaneous claims regarding a common witness $\boldsymbol{e}'$, which naturally map to two operational phases:

1. *Phase 1: Permutation Claim.* We prove that $\boldsymbol{e}'$, committed in V, is a valid permutation of $\boldsymbol{e}$. This depends only on the witness $\boldsymbol{e}'$ and is independent of the ciphertexts.
2. *Phase 2: Consistency Claim.* We prove that this same $\boldsymbol{e}'$ satisfies the aggregated ElGamal relation defined in Eq. 6.

To achieve this, we design a single NIZK proof that binds these two phases.

3.2 Phase 1: Permutation Argument

To prove the permutation claim, the prover commits to the permuted vector $\boldsymbol{e}' = (e_{\pi(1)}, \ldots, e_{\pi(n)})$. The challenge vector $\boldsymbol{e}$ must be derived after the shuffled ciphertexts $\boldsymbol{C}'$ are published to ensure soundness; otherwise, the prover could choose the permutation adaptively. In the interactive setting, the verifier samples $\boldsymbol{e} \xleftarrow{\$} \mathbb{Z}_p^n$ after receiving $\boldsymbol{C}'$. In the non-interactive setting, $\boldsymbol{e}$ is derived via the Fiat-Shamir transformation, i.e., a seed z_e is computed by hashing the public statement, and $\boldsymbol{e} = \mathsf{PRG}(z_e)$ is expanded using a pseudo-random generator. The prover computes $\boldsymbol{e}'$ and commits to it using a single Pedersen vector commitment: $V = h^s \cdot \boldsymbol{g}^{\boldsymbol{e}'} \in \mathbb{G}$ where $\boldsymbol{g}, h \in \mathsf{ck}$, and $s \xleftarrow{\$} \mathbb{Z}_p$ is a secret blinding factor. The commitment V is sent to $\mathcal{V}$.

To prove that $\boldsymbol{e}'$ is a permutation of $\boldsymbol{e}$, we employ the classic algebraic technique based on polynomial identity [25, 21, 19]. Two vectors represent the same multiset (i.e., one is a permutation of the other) if and only if their characteristic polynomials are identical. The verifier issues a random challenge $u \in \mathbb{Z}_p$, and by the Schwartz-Zippel lemma, this polynomial identity can be verified with overwhelming probability by checking a single evaluation point:

$$\prod_{i=1}^{n} (e_i - u) = \prod_{i=1}^{n} (e'_i - u). \tag{7}$$

The verifier computes $y_{\text{LHS}} = \prod_{i=1}^{n} (e_i - u)$. The prover must show that $\prod_{i=1}^{n} (e'_i - u) = y_{\text{LHS}}$ using $\boldsymbol{e}'$ as the witness. This check is reduced to an arithmetic circuit (AC) satisfiability problem. The circuit takes the witness components $(e'_1, \ldots, e'_n)$ as inputs and contains $n - 1$ multiplication gates to compute their product, $y_{\text{RHS}} = \prod_{i=1}^{n} (e'_i - u)$.

This structure is directly compatible with the Bulletproofs framework [11], which maps an arithmetic circuit into two sets of constraints over wire values:

- *Hadamard Product Relation*: $\boldsymbol{a}_L \circ \boldsymbol{a}_R = \boldsymbol{a}_O$, enforcing correctness of multiplication gates.
- *Linear Constraints*: A system of Q linear equations enforcing correct wiring between gates and binding inputs/outputs.

Witness Encoding. We encode the permutation check (7) as an arithmetic circuit with $n-1$ multiplication gates computing the cumulative product. The circuit is structured as a chain: gate $j \in [n-1]$ computes $a_{O,j} = a_{L,j} \cdot a_{R,j}$, where

- *Left inputs:* $a_{L,1} = e'_1 - u$, and $a_{L,j} = a_{O,j-1}$ for $j \geq 2$ (chaining previous output to current input).
- *Right inputs:* $a_{R,j} = e'_{j+1} - u$ for all $j \in [n-1]$.

The final output is $a_{O,n-1} = \prod_{i=1}^{n}(e'_i - u)$, which must equal the public value $y_{\mathsf{LHS}} = \prod_{i=1}^{n}(e_i - u)$.

This circuit structure induces $Q = 2n-1$ linear constraints encoded by the weight matrices $\boldsymbol{W}_L, \boldsymbol{W}_R, \boldsymbol{W}_O \in \mathbb{Z}_p^{Q\times(n-1)}$, $\boldsymbol{W}_V \in \mathbb{Z}_p^{Q\times n}$, and $\boldsymbol{c} \in \mathbb{Z}_p^{Q}$:

- *Left input wiring* ($n-1$ constraints): Bind $a_{L,1}$ to the witness ($a_{L,1} = e'_1 - u$) and chain subsequent left inputs ($a_{L,j} = a_{O,j-1}$ for $j \geq 2$).
- *Right input wiring* ($n-1$ constraints): Bind each right input to the witness ($a_{R,j} = e'_{j+1} - u$).
- *Output constraint* (1 constraint): Bind the final output to the public target ($a_{O,n-1} = y_{\mathsf{LHS}}$).

The explicit construction of $\boldsymbol{W}_L, \boldsymbol{W}_R, \boldsymbol{W}_O, \boldsymbol{W}_V$ and $\boldsymbol{c}$ is given in Appendix F. Together with the Hadamard relation $\boldsymbol{a}_L \circ \boldsymbol{a}_R = \boldsymbol{a}_O$ enforced by the arithmetic circuit protocol, these constraints fully specify the permutation check. The relation $\mathcal{R}_{\mathsf{AC.1}}$ captures this structure:

$$\mathcal{R}_{\mathsf{AC.1}} = \left\{ \begin{array}{l} (\mathsf{ck},\ V \in \mathbb{G},\ \boldsymbol{W}_L, \boldsymbol{W}_R, \boldsymbol{W}_O, \boldsymbol{W}_V, \boldsymbol{c}\ ; \\ \quad \boldsymbol{a}_L, \boldsymbol{a}_R, \boldsymbol{a}_O \in \mathbb{Z}_p^{n-1},\ \boldsymbol{e'} \in \mathbb{Z}_p^{n},\ s \in \mathbb{Z}_p) : \\ \quad V = \boldsymbol{g}^{\boldsymbol{e'}} h^{s} \ \wedge\ \boldsymbol{a}_L \circ \boldsymbol{a}_R = \boldsymbol{a}_O \\ \quad \wedge\ \boldsymbol{W}_L \boldsymbol{a}_L + \boldsymbol{W}_R \boldsymbol{a}_R + \boldsymbol{W}_O \boldsymbol{a}_O = \boldsymbol{W}_V \boldsymbol{e'} + \boldsymbol{c} \end{array} \right\} \tag{8}$$

The first constraint binds $\boldsymbol{e'}$ to the commitment V. The second enforces multiplication gate correctness. The third enforces the circuit's wiring and output constraints.

The Challenge. Proving $\mathcal{R}_{\mathsf{AC.1}}$ directly with the standard Bulletproofs protocol presents a challenge regarding the linear constraints. In standard Bulletproofs, linear constraints are verified by taking a random linear combination of the Q constraints using a challenge vector $\boldsymbol{z} \in \mathbb{Z}_p^Q$ supplied by the verifier. The verifier then computes a multiscalar multiplication. More precisely, for a linear term involving $\boldsymbol{W}_V \cdot \boldsymbol{e'}$, the verifier would need to compute:

$$\prod_i V_i^{z_i \cdot W_{V_i}},$$

where each $V_i = g^{e'_i} h^{s_i}$ is a Pedersen commitment to the individual witness element e'_i. This verification step is possible only when the verifier holds n individual commitments $\{V_i\}_{i\in[n]}$. However, in our approach, to reduce communication cost, we provide only a single aggregated vector commitment $V = \boldsymbol{g}^{\boldsymbol{e'}} h^{s}$. The verifier cannot independently extract the individual witness coordinates to compute the required linear term $\langle \boldsymbol{z}, \boldsymbol{W}_V \cdot \boldsymbol{e'} \rangle$.

We resolve this issue by separating circuit satisfiability from witness binding, and bridging the two using an auxiliary commitment. The details of the resulting protocol are available in Appendix F.

3.3 Phase 2: Ciphertext Consistency

We now prove that the exponent vector $\boldsymbol{e}'$ committed in V is the *same witness* used in the aggregated ciphertext equations. This consistency condition is captured by the relation $\mathcal{R}_{\text{AC.3}}$.

$$\mathcal{R}_{\text{AC.3}} = \left\{ \begin{array}{l} (\mathsf{pk}, \mathsf{ck}, V, C'_1, C'_2 \in \mathbb{G}, \boldsymbol{g} \in \mathbb{G}^n, \boldsymbol{C}' \in (\mathbb{G}^2)^n; \\ \quad \boldsymbol{e}' \in \mathbb{Z}_p^n, s, r' \in \mathbb{Z}_p) : V = h^s \cdot \boldsymbol{g}^{\boldsymbol{e}'} \\ \wedge\ C'_1 = g^{-r'} \cdot \boldsymbol{c}_1'^{\,\boldsymbol{e}'} \ \wedge\ C'_2 = f^{-r'} \cdot \boldsymbol{c}_2'^{\,\boldsymbol{e}'} \end{array} \right\} \tag{9}$$

A direct Σ-protocol for $\mathcal{R}_{\text{AC.3}}$ would require transmitting a linear-size response vector. We instead compress the proof to $O(\log n)$ group elements using the extended Chaum–Pedersen argument (Section 2.6). Our ciphertext consistency argument is detailed in Figure 3.

$\mathcal{P}$ computes: (10)

$y \xleftarrow{\$} \mathbb{Z}_p$ (11)

$S'_1 = g^y \cdot \boldsymbol{c}_1'^{\,\boldsymbol{s}'_L}$ // Uses $\boldsymbol{s}'_L$ from Eq. 48 (12)

$S'_2 = f^y \cdot \boldsymbol{c}_2'^{\,\boldsymbol{s}'_L}$ (13)

$\mathcal{P} \to \mathcal{V} : S'_1, S'_2$ (14)

$\mathcal{V} \to \mathcal{P} : x'$ // Same challenge x' from Eq. 103 (15)

$\mathcal{P}$ computes: (16)

$\rho_c = yx' - r' \pmod p$ (17)

$\mathcal{P} \to \mathcal{V} : \rho_c$ (18)

$\mathcal{V}$ computes and checks: (19)

$P'_1 = C_1 \cdot S_1'^{x'} \cdot g^{-\rho_c} \in \mathbb{G}$ (20)

$P'_2 = C_2 \cdot S_2'^{x'} \cdot f^{-\rho_c} \in \mathbb{G}$ (21)

Define $\boldsymbol{G} = (\boldsymbol{g}, \boldsymbol{c}'_1, \boldsymbol{c}'_2)$ and $\boldsymbol{P} = (P' h^{-\mu_c}, P'_1, P'_2)$ (22)

$\mathcal{P} \leftrightarrow \mathcal{V} : \mathsf{eCP}(\boldsymbol{G}, \boldsymbol{P}; \boldsymbol{l}_c)$ (23)

if all checks succeed: $\mathcal{V}$ accepts, else: rejects (24)

Figure 3: Protocol 3.3: Ciphertext Consistency (Phase 2).

Cross-phase binding. The two phases are cryptographically bound through three shared quantities. First, both phases operate on the *same* Pedersen vector commitment $V = h^s \cdot \boldsymbol{g}^{\boldsymbol{e}'}$, which anchors the witness $\boldsymbol{e}'$ across the permutation and the ciphertext-consistency claims. Second, the binding sub-protocol in Phase 1 (Figure 6, Appendix F) constructs a degree-1 witness polynomial $\boldsymbol{l}_c(X) = \boldsymbol{e}' + \boldsymbol{s}'_L \cdot X$, where $\boldsymbol{s}'_L$ is a fresh random masking vector; Phase 2 reuses this *same* $\boldsymbol{s}'_L$ to form its commitments (Appendix 3.3), so that both protocols share a common evaluated witness $\boldsymbol{l}_c = \boldsymbol{e}' + \boldsymbol{s}'_L \cdot x'$. Third, the challenge x' is sampled *once*, after both phases have published their commitments, preventing a cheating prover from tailoring separate witnesses for the two phases independently. By this construction, the prover proves knowledge of a single vector $\boldsymbol{l}_c$ that simultaneously opens P' (derived from the vector commitment V via Phase 1) and P'_1, P'_2 (derived from the ciphertext equations via Phase 2). Using a common witness $\boldsymbol{l}_c = \boldsymbol{e}' + \boldsymbol{s}'_L\, x'$ cryptographically binds the two phases: any inconsistency would require breaking the binding property of V under the DLR assumption.

Complexity. The outer protocol of Π_{base} (Figures 3 and 6) transmits $O(1)$ group elements and scalars, independent of n. The recursive sub-protocols contribute $O(k \cdot \log_k n)$ group elements and $O(m)$ field elements (Sections 2.5 and 2.6), yielding a total proof size of $O(k \cdot \log_k n)$ group elements. The prover performs $O(n)$ group exponentiations (Appendix I.1). The verifier performs a single batched multi-scalar multiplication of width $4n$, yielding $O(n)$ group exponentiations (Section 4.3.2).

Theorem 3. *The shuffle argument* Π_{base} *has perfect completeness, perfect special honest-verifier zero-knowledge, and computational witness-extended emulation, assuming the hardness of the DLR assumption and the security of the underlying* IPA *and* eCP *protocols.*

Neither the IPA (Theorem 1) nor the eCP (Theorem 2) is zero-knowledge: both reveal the reduced witness in the base case of the folding recursion. The shuffle argument Π_{base} is nonetheless perfectly SHVZK because the outer protocol layer blinds the witness *before* either sub-protocol sees it. In Phase 1, the prover embeds the secret circuit-wire vectors $(\boldsymbol{a}_L, \boldsymbol{a}_R, \boldsymbol{a}_O)$ as coefficients of polynomials $\boldsymbol{l}(X), \boldsymbol{r}(X)$ and attaches fresh random blinding vectors $\boldsymbol{s}_L, \boldsymbol{s}_R$ at a distinct power of X (Figure 6, Appendix F). In Phase 2, the witness polynomial $\boldsymbol{l}_c(X)$ combines the witness with a uniform masking vector $\boldsymbol{s}'_L$ (Figure 3, Appendix 3.3). In both cases, evaluating at the verifier's challenge yields vectors $(\boldsymbol{l}, \boldsymbol{r}, \boldsymbol{l}_c)$ that are uniformly distributed and independent of the witness, thereby acting as a one-time pad. These blinded vectors are then handed to the IPA and eCP, which may reveal them in the base case, but they carry no information about the actual witness $(\boldsymbol{e}', \boldsymbol{a}_L, \boldsymbol{a}_R, \boldsymbol{a}_O)$. A simulator exploits this structure by sampling $\boldsymbol{l}, \boldsymbol{r}, \boldsymbol{l}_c$ uniformly (matching the real distribution exactly), solving backwards for the commitments that are uniquely determined by the verification equations, and then invoking the sub-protocols on these simulated inputs using their honest-prover algorithms. The full proof for Theorem 3 is presented in Appendix G.

Verification structure of Π_{base}**.** The base protocol requires three sub-protocol executions. Phase 1 (Appendix F) produces two IPA instances: one proving circuit satisfiability over the blinded witness $(\boldsymbol{l}, \boldsymbol{r})$, and one proving that the bridging commitment T_2 is consistent with the vector commitment V over the binding witness $(\boldsymbol{l}_c, \boldsymbol{r}_c)$. Phase 2 produces a 3-row eCP instance (Eq. 23) proving ciphertext consistency using the same witness $\boldsymbol{l}_c$. The verifier accepts Π_{base} if and only if all three sub-protocol checks, together with the scalar consistency checks from each phase.

3.4 Protocol Optimization

We apply internal batching to reduce the three sub-protocol executions of Π_{base} to two, yielding the optimized protocol Π_{opt} shown in Figure 4. We also discuss generalizations to wider ciphertext formats and to batched verification of multiple MAYA argument instances.

To maximize performance, we apply internal batching techniques: Aggregated IPA (Appendix C) and Aggregated eCP (Appendix E) to the shuffle argument. These optimizations effectively reduce the protocol from three independent executions (two Phase-1 IPAs and one 3-row Phase-2 eCP) to two aggregated forms: one IPA (Eq. 35) and one 2-row eCP (Eq. 36). This restructuring achieves substantial computational and communication improvements, which we analyze below. The final optimized shuffle proof protocol Π_{opt} preserves all security properties of the base protocol Π_{base}. The proof is provided in Appendix H.

Computational Gain: Aggregated IPA**.** Rather than executing two independent recursive arguments, we merge them into a single "Aggregated IPA" by treating the respective witness vectors from the first argument $(\boldsymbol{l}, \boldsymbol{r})$ and the second argument $(\boldsymbol{l}_c, \boldsymbol{r}_c)$ as coefficients of a degree-1 polynomial (see Appendix C). Thus, $\mathcal{P}$ and $\mathcal{V}$ execute Eq. 35:

$$\mathcal{P} \leftrightarrow \mathcal{V} : \mathsf{IPA}(\boldsymbol{g}, \boldsymbol{h}', P_{\mathrm{agg}}, \hat{t}_{\mathrm{agg}}; \boldsymbol{l}_{\mathrm{agg}}, \boldsymbol{r}_{\mathrm{agg}}),$$

$$\mathcal{P} \to \mathcal{V} : A_I, A_O, S \tag{25}$$
$$\mathcal{V} \to \mathcal{P} : y, z \tag{26}$$
$$\mathcal{P} \to \mathcal{V} : \{T_i^*\}_{i\in\{1,3,4,5,6\}}, T_2 \tag{27}$$
$$\mathcal{V} \to \mathcal{P} : x \tag{28}$$
$$\mathcal{P} \to \mathcal{V} : \tau_x, \mu, \hat{t} \tag{29}$$
$$\mathcal{P} \to \mathcal{V} : S', T_1', S_1', S_2' \tag{30}$$
$$\mathcal{V} \to \mathcal{P} : x' \tag{31}$$
$$\mathcal{P} \to \mathcal{V} : \tau_c, \mu_c, \hat{t}_c, \rho_c \tag{32}$$
$$\mathcal{P} \to \mathcal{V} : t_{\text{cross}} = \langle \boldsymbol{l}, \boldsymbol{r}_c \rangle + \langle \boldsymbol{l}_c, \boldsymbol{r} \rangle \qquad \text{// cross-terms} \tag{33}$$
$$\mathcal{V} \to \mathcal{P} : x_{\text{agg}} \xleftarrow{\$} \mathbb{Z}_p^* \tag{34}$$
$$\mathcal{P} \leftrightarrow \mathcal{V} : \mathsf{IPA}(\boldsymbol{g}, \boldsymbol{h}', P_{\text{agg}}, \hat{t}_{\text{agg}}; \boldsymbol{l}_{\text{agg}}, \boldsymbol{r}_{\text{agg}}) \tag{35}$$
$$\mathcal{P} \leftrightarrow \mathcal{V} : \mathsf{eCP}(\boldsymbol{B}_{\text{agg}}, P'_{\text{agg}}; \boldsymbol{l}_c) \tag{36}$$

Figure 4: Protocol Π_{opt}: Final shuffle proof protocol.

where the aggregated witness is $\boldsymbol{l}_{\text{agg}} = \boldsymbol{l} + x_{\text{agg}} \cdot \boldsymbol{l}_c$ and $\boldsymbol{r}_{\text{agg}} = \boldsymbol{r} + x_{\text{agg}} \cdot \boldsymbol{r}_c$. The verifier computes the aggregated commitment $P_{\text{agg}} = P h^{-\mu} \cdot (P' h^{-\mu_c} \boldsymbol{h}'^{\boldsymbol{r}_c})^{x_{\text{agg}}}$ and the target inner product $\hat{t}_{\text{agg}} = \hat{t} + x_{\text{agg}} t_{\text{cross}} + x_{\text{agg}}^2 \hat{t}_c$.

The primary cost of a zero-knowledge proof lies in group operations, specifically Multi-Scalar Multiplications (MSMs). By batching, we perform the scalar algebra to mix the witnesses (which is computationally negligible) and then execute the expensive MSMs only once for the combined witness instead of twice. This effectively *eliminates one full IPA execution* from $\mathcal{P}$'s workload.

Communication Efficiency: Batched eCP We further optimize the eCP argument(Eq. 23), which verifies the consistency of the witness $\boldsymbol{l}_c$ across three commitments: the binding commitment ($\boldsymbol{g}^{\boldsymbol{l}_c}$) and the two ciphertext components ($\boldsymbol{c}_1'^{\boldsymbol{l}_c}, \boldsymbol{c}_2'^{\boldsymbol{l}_c}$). By aggregating these three rows into two, we achieve both computational and communication gains.

- *Computation:* Unlike IPA batching which mixes scalars, eCP batching requires mixing group elements (see Appendix E). Constructing the aggregated base $\boldsymbol{c}_1' \cdot (\boldsymbol{c}_2')^{x_{\text{agg}}}$ requires n exponentiations as a one-time setup cost. Thus, $\mathcal{P}$ and $\mathcal{V}$ execute Eq. 36:

$$\mathcal{P} \leftrightarrow \mathcal{V} : \mathsf{eCP}(\boldsymbol{B}_{agg}, P'_{\text{agg}}; \boldsymbol{l}_c).$$

  where $\boldsymbol{B}_{agg} = (\boldsymbol{g}, \boldsymbol{c}_1' \cdot (\boldsymbol{c}_2')^{x_{\text{agg}}})$ and $P'_{\text{agg}} = (P', P_1' P_2'^{x_{\text{agg}}})$. While reducing from 3 rows to 2 theoretically yields a $1.5\times$ speedup on the recursive portion, the setup cost partially offsets this gain.

- *Bandwidth:* The primary advantage is proof size reduction. By aggregating from 3 rows to 2, we decrease the number of group elements transmitted per recursion round by 33%. Combined with the IPA aggregation, this optimization is critical for overall communication efficiency.

Acceptance and non-interactivity. The verifier accepts the shuffle proof if and only if the scalar consistency checks from both phases all hold, and the two aggregated sub-protocols (35 and 36) both verify. The entire protocol is made non-interactive using the strong Fiat–Shamir heuristic [9]: The challenge vector $\boldsymbol{e} \in \mathbb{Z}_p^n$ is derived as $\boldsymbol{e} = \mathsf{PRG}(\mathsf{Hash}(I))$ where $I = (\mathsf{pk}, \mathsf{ck}, \boldsymbol{C}, \boldsymbol{C}')$ is the initial transcript. Subsequent

challenges $(u, y, z, x, x', x_{\text{agg}})$ are derived sequentially by hashing the transcript including all prior prover messages.

Sequential Batch Verification. Although the presentation above considers a single prover (i.e., a single mix node), the protocol naturally extends to a sequence of $m \geq 1$ mix nodes. Owing to the linear structure of the verification equations, an auditor can aggregate the verification of all m shuffle proofs into a single batched check. While this aggregated check masks the specific identity of a dishonest node in the event of failure, it enables a highly efficient "optimistic" verification path where the entire chain is verified at the cost of a single shuffle. This trade-off is operationally sound: in the rare event of an invalid batch, the auditor simply reverts to checking individual proofs to identify the cheater. Since invalid proofs prevent final decryption and thus halt the protocol before any plaintexts are revealed, no voter privacy is compromised. Cheating remains publicly detectable, making the optimistic (batched) path both secure and operationally dominant.

Generalization to Multi-ElGamal Encryption. Our shuffle argument naturally extends to support Multi-ElGamal encryption, where a single randomness value r is used to encrypt a vector of messages $\mathbf{M} \in \mathbb{G}^m$ under a set of public keys $\mathbf{h} = (h_1, \ldots, h_m)$. In this setting, each ciphertext consists of $m+1$ group elements $C_i = (C_{i,0}, \ldots, C_{i,m}) \in \mathbb{G}^{m+1}$, defined as $C_{i,0} = g^{r_i}$ and $C_{i,k} = M_{i,k} \cdot h_k^{r_i}$ for $k \in [1, m]$.

To adapt the protocol Π_{opt} for this "wide" ciphertext structure, while Permutation phase remains unchanged, the Ciphertext Consistency phase is generalized as follows:

1. Expanded Commitments: In step (30), the prover must bind the witness $\boldsymbol{s}'_L$ to every component of the re-randomized ciphertexts. Instead of transmitting only two commitments S'_1, S'_2, the prover computes and transmits $m+1$ commitments $\{S'_k\}_{k=0}^m$, where $S'_0 = g^y \cdot (\boldsymbol{c}'_0)^{s'_L}$ and $S'_k = h^y \cdot (\boldsymbol{c}'_k)^{s'_L}$ corresponds to the vector of the k-th ciphertext components $\boldsymbol{c}'_k \in \mathbb{G}^n$.

2. Generalized Batching: The Aggregated eCP argument in step (36) is modified to verify the consistency of all $m+1$ columns simultaneously. Leveraging the random linear combination technique used in Section 3.4, the verifier and prover replace the binary aggregation with a polynomial aggregation of degree m. Using the challenge $x_{\text{agg}} \in \mathbb{Z}_p^*$, the aggregated base $\boldsymbol{B}_{\text{agg}}$ and target P'_{agg} are constructed as:
$$\boldsymbol{B}_{\text{agg}} = \left(\boldsymbol{g}, \prod_{k=0}^{m} (\boldsymbol{c}'_k)^{x_{\text{agg}}^k}\right), \quad P'_{\text{agg}} = \left(P', \prod_{k=0}^{m} (P'_k)^{x_{\text{agg}}^k}\right)$$
where P'_k is the target value derived from the k-th commitment S'_k.

Crucially, while the prover's initial message in step (30) grows linearly with m (adding $m-1$ group elements), the size of the recursive arguments (IPA and eCP) remains constant. Thus, the protocol maintains its logarithmic communication complexity $O(\log n)$ even for complex Multi-ElGamal ciphertexts.

4 Experimental Evaluation

We now evaluate the performance of the MAYA protocol Π_{opt}. As described in Section 3, the prover $\mathcal{P}$ must generate logarithmic-size arguments, which we instantiate from the underlying primitives ($\mathsf{IPA}_{k,m}$ and $\mathsf{eCP}_{k,m}$) using the generalized k-ary folding mechanisms described in Sections 2.5 and 2.6.

As discussed in our related works (Sec.1.1), we benchmark this instantiation against the shuffle process of industry-standard Verificatum mixnet [27].

In what follows, we first describe the implementation environment in Section 4.1. Section 4.2 analyzes the impact of the folding parameter k and identifies the optimal-performing configuration. Section 4.3

then compares our protocol against Verificatum. Section 4.4 studies the effect of padding strategies and the trade-off between runtime and proof size. Section 4.5 synthesizes these findings into a performance snapshot at national-election scale ($n = 10^6$).

4.1 Setup and Methodology

We implemented MAYA in Rust, with the cryptographic parameters chosen for a 128-bit security level ($\lambda \approx 128$). The primary instantiation, MAYA-Ristretto, utilizes the `curve25519-dalek` library [2] for elliptic curve operations over Ristretto255. The Verificatum baseline (v3.1.0) [33] operates over the NIST P-256 curve so, in order to ease comparison, we also provide a secondary instantiation, MAYA-P256, using the `p256` crate [4]. The artifact is available on GitHub [5].

The implementation effort required to instantiate MAYA-Ristretto, approximately $2,400$ lines of Rust extending/modifying the Bulletproofs library, with a self-contained $\sim 1,080$-line verifier, is documented in Appendix K.

All benchmarks were conducted on a laptop equipped with an Intel Core i5-1245U processor (featuring 2 Performance-cores and 8 Efficient-cores for a total of 12 threads, with a max turbo frequency of 4.4 GHz), 16 GB of RAM, and running Ubuntu 22.04 LTS. We report results for single-threaded and multi-threaded executions.

Comparison Constraints. Our implementation was compiled with `rustc` 1.88.0. Verificatum was executed using OpenJDK 17 with the JVM heap capped at 8 GB (`-Xmx8G`); all other JVM parameters were left at OpenJDK defaults (which is consistent with the Verificatum user manual [32]). For single-threaded experiments, both systems were pinned to a single thread via `taskset`. Verificatum's internal parallelism was set to 1 through its documented properties (namely, `java.util.concurrent.ForkJoinPool.common.parallelism`, `com.verificatum.arithm.globalParallelism`, and `com.verificatum.eio.globalParallelism`). For multi-threaded experiments, these restrictions were removed, allowing both systems to utilize all 12 available logical threads.

As no production-grade Rust implementation of Verificatum exists, this cross-language comparison is unavoidable. Both systems are evaluated under identical hardware and concurrency constraints; we focus on relative scaling behavior rather than absolute constant factors.

Measurement protocol. For our implementation, timings were collected using the Criterion.rs benchmarking framework, with around 100 iterations per configuration following a 20-second warm-up period. Reported values correspond to mean execution times with automatic outlier removal. For Verificatum, each benchmark was executed for 100 iterations following a 50-iteration warm-up period, with a 10-second pause between iterations to prevent thermal throttling. The mean runtime was recorded.

4.2 Optimizing the Folding Factor

We benchmark prover performance across folding factors $k \in \{2, 3, 4, 5, 6, 8, 16\}$ for input sizes from $n = 10^3$ to $n = 10^6$. Unless otherwise specified, we set the base-case parameter at $m = k - 1$ and terminate recursion in our logarithmic arguments when $n \leq m$, transmitting final witness vectors of size at most m (as discussed in Section 2.5.2).

Figure 5 reports proof creation time normalized to the $k = 2$ baseline. The $k = 4$ configuration consistently achieves the lowest or near-lowest normalized runtime across all input sizes, reaching $\approx 0.82\times$–$0.88\times$ relative to $k = 2$ (a 12–18% improvement). The $k = 3$ configuration performs comparably, achieving $\approx 0.83\times$–$0.97\times$ depending on the input size, and is essentially tied with $k = 4$ at $n = 10^4$. For $k \geq 8$, arith-

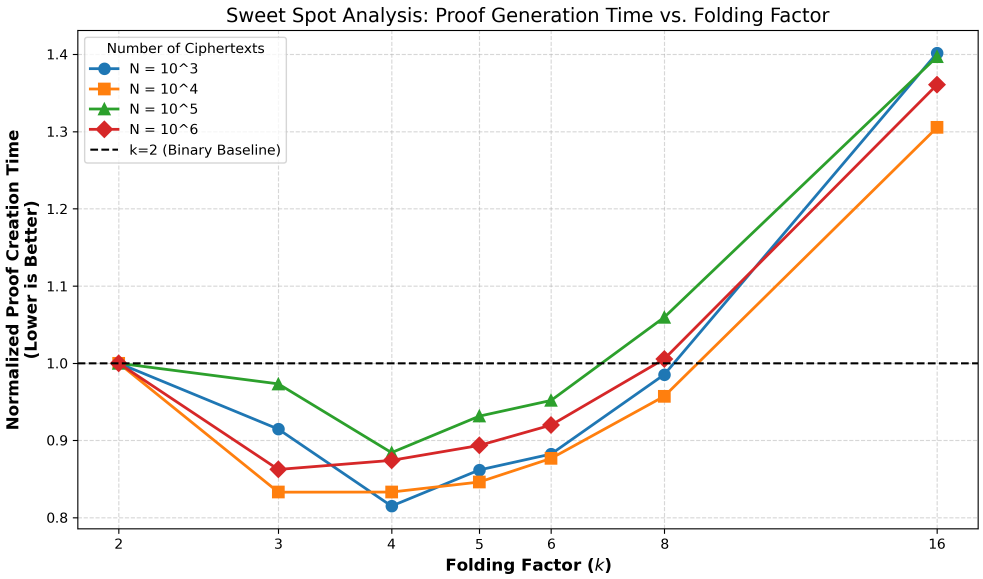


Figure 5: Sweet Spot Analysis. Proof creation time normalized to the generic k-ary baseline with $k = 2$ (dashed line at $y = 1.0$).

metic overhead dominates: $k = 8$ returns to near-baseline performance, and $k = 16$ incurs a consistent 1.3–$1.4\times$ slowdown across all input sizes.

These results reflect a trade-off between arithmetic cost, which scales as $O(nk)$ per the multi-scalar multiplications (MSMs) complexity, and structural overhead, which scales with recursion depth $d = \log_k(n/m)$. At $k \in \{3, 4\}$, the reduction in recursion depth compensates for the increased per-round work, whereas at $k \geq 8$ the per-round cost dominates. A detailed cost model analysis appears in Appendix L.

Based on these results, we adopt $k = 4$ as our primary configuration, as it delivers the most stable improvement across all input sizes. Although $k = 3$ is competitive, and occasionally marginally faster at large n, $k = 4$ offers a more consistent speedup. In subsequent comparisons, we report results for both $k = 4$ (4-ary folding) and $k = 2$ (binary folding) so that the former can be evaluated alongside the standard $k = 2$ baseline commonly used in prior work.

4.3 Performance Comparison with Verificatum

We benchmark MAYA against the shuffle process of Verificatum [34], focusing on three deployment-relevant metrics: prover throughput (operational cost), verifier efficiency (auditability), and proof size (storage and communication overhead). All benchmarks utilize the iterative padding and sparse representation strategies detailed in Section 4.4 to accommodate input sizes that are not powers of k.

4.3.1 Prover Throughput.

Prover timings include all steps the prover performs: generating the common reference string (CRS), computing fixed-base exponentiation tables for pk (i.e., g, f) for faster scalar multiplication [14], re-encrypting the ciphertexts (i.e., computing $\boldsymbol{C}'$), deriving the challenge $\boldsymbol{e}$ by hashing all public statements including the input ciphertexts $\boldsymbol{C}$ and output ciphertexts $\boldsymbol{C}'$, and finally generating the proof of shuffle. This fixed-base precomputation technique is also adopted in Verificatum implementation.

Table 1: MAYA-Ristretto prover performance (in milliseconds) for a shuffle of n ciphertexts in both single-threaded (ST) and multi-threaded (MT) settings.

n	Mode	Binary folding	4-ary folding	Verificatum
1,000	ST	259.78	248.63	2,911.23
	MT	163.43	149.13	2,573.10
10,000	ST	2,589.40	2,361.71	6,149.56
	MT	1,303.99	1,210.48	4,082.20
100,000	ST	27,060.31	24,451.83	31,640.20
	MT	13,059.43	12,148.51	16,371.85
1,000,000	ST	276,890.28	248,366.91	292,394.82
	MT	129,150.77	120,287.61	183,809.97

Table 1 reports total proving times of MAYA-Ristretto for varying shuffle sizes in single-threaded and multi-threaded settings, respectively. The timings for MAYA-P256 are given in Appendix N.

The reported timings exclude disk I/O (reading the public key pk and input ciphertexts $\boldsymbol{C}$; writing the proof, commitment V, and shuffled ciphertext list $\boldsymbol{C}'$). When I/O is included, the end-to-end prover time for $n = 10^6$ with 4-ary folding is 254.91 s (single-threaded) and 122.42 s (multi-threaded). For binary folding, the corresponding figures are 283.53 s and 131.05 s, respectively. In practice, these I/Os may or may not be relevant, depending on whether inputs and outputs are written locally or directly exchanged with other parties using whatever network is available.

MAYA's provers exhibit linear runtime in the number of ciphertexts. The 4-ary folding prover consistently outperforms binary folding by 4–11% in the single-threaded setting and 7–10% in the multi-threaded setting. Compared to Verificatum, MAYA-Ristretto with 4-ary folding is faster across all scales in the single-threaded setting. The large advantage at small n is partly attributable to Verificatum's JVM startup overhead, which is amortized at larger scales where algorithmic costs dominate.

Multi-threading yields a $\approx 2\times$ speedup for MAYA-Ristretto across all input sizes, consistent with the parallelizable structure of the underlying MSM computations. Verificatum achieves a $1.6\times$ multi-threaded speedup at $n = 10^6$, reaching 183.8 s compared to 120.3 s for MAYA-Ristretto with 4-ary folding.

These results demonstrate that recursive folding does not impose a significant constant-factor overhead on the prover, even while reducing proof size from linear to logarithmic. Overall, MAYA-Ristretto achieves prover efficiency comparable to the state-of-the-art linear-size Verificatum construction.

4.3.2 Verifier Efficiency.

We benchmark the cost of verifying a MAYA shuffle argument. Verification consists of all steps the verifier performs: generating the CRS, deriving the challenge $\boldsymbol{e}$ by hashing all public statements, and checking the verification equations as a single batched MSM that aggregates all verification equations via random linear combinations.

Following the communication analysis in Sections 2.5 and 2.6, the verifier's dominant MSM width is approximately $4n$ points (representing $\boldsymbol{g}, \boldsymbol{h}, \boldsymbol{c}'_1, \boldsymbol{c}'_2$), plus $3(2k-2)d$ points for the IPA and eCP protocols, and a 21-point constant for circuit and commitment terms. For $n = 10^6$ and $k = 4$, this logarithmic contribution is $3 \cdot 6 \cdot 10 = 180$ points, which is negligible compared to the $4n$ linear term that drives overall verification time.

Table 2 reports verification times of MAYA-Ristretto across varying input sizes in single-threaded and multi-threaded settings. As before, we also offer timings for MAYA-P256 in Appendix O.

Table 2: MAYA-Ristretto verifier performance (in milliseconds) for a shuffle of n ciphertexts in both single-threaded (ST) and multi-threaded (MT) settings.

n	Mode	Binary folding	4-ary folding	Verificatum
1,000	ST	35.78	35.94	1,314.64
	MT	36.55	36.25	722.77
10,000	ST	329.90	322.51	4,987.67
	MT	172.95	184.11	2,029.15
100,000	ST	3,395.22	3,395.22	38,199.41
	MT	1,735.48	1,738.96	17,283.31
1,000,000	ST	33,467.88	32,669.19	367,345.90
	MT	17,524.76	17,126.44	168,543.29

The reported timings exclude disk I/O. When I/O is included, primarily reading the two ciphertext lists $\boldsymbol{C}$ and $\boldsymbol{C}'$, the end-to-end verifier time for $n = 10^6$ with 4-ary folding is 45.63 s (single-threaded) and 21.54 s (multi-threaded). For binary folding, the corresponding figures are 46.42 s and 21.88 s, respectively.

A further observation is that CRS regeneration accounts for roughly 40% of MAYA-Ristretto's verifier time in both single-threaded setting (13.2s out of 32.7s) and the multi-threaded setting (7.52s out of 17.1s). For MAYA-P256, the corresponding share is likewise about 40% in the single-threaded case (20.7s out of 49.4s), and exceeds 50% in the multi-threaded case (12.2s out of 21.6s). Since the CRS generation is independent of the ciphertexts, it can be performed once, possibly before the election, and reused across multiple verifications. Excluding this cost, the MAYA-Ristretto verifier's actual cryptographic work reduces to approximately 20 s (single-threaded) or 10 s (multi-threaded) at $n = 10^6$.

Multi-threading provides a $\approx 2\times$ verifier speedup at large n but yields no benefit for small n (e.g., $n = 1{,}000$). This is because parallelism primarily targets MSMs, whereas other verifier operations, such as Fiat–Shamir transcript replay, scalar reconstruction for IPA and eCP, and point decompression, are executed sequentially. Additionally, the Dalek MSM engine is internally single-threaded, meaning parallelism is limited to protocol-level task splitting via `rayon::join`. By contrast, MAYA-P256 leverages the Arkworks library, which natively supports parallel MSMs. This allows it to achieve a $\approx 2\times$ verifier speedup even at small scales and a $3.3\times$–$3.4\times$ multi-threaded speedup for the prover at large n (Appendices N and O).

Unlike on the prover side, binary and 4-ary folding achieve nearly identical verification times (within 2.4%). This is expected: since all folding configurations share the same dominant $4n$-width MSM for the batched verification check, the logarithmic-depth terms (180 points for $k = 4$ vs. 120 for $k = 2$) contribute negligibly at large n.

The I/O Bottleneck. At $n = 10^6$, the single-threaded verifier spends 45.6s end-to-end: about 13 s is spent on file I/O (reading and parsing the ciphertext lists), roughly 19 s on the aggregation MSMs and CRS generation, leaving only ~ 14 s on the batched proof verification. In other words, parsing the input files takes nearly as long as cryptographically checking the proof itself. This suggests that, at current performance levels, further optimizing the cryptographic verifier yields diminishing returns; the bottleneck is shifting toward efficient data serialization and I/O, an engineering challenge orthogonal to the proof system design.

4.3.3 Proof Size.

For the purpose of our comparison with Verificatum, we report the pure shuffle proof size, defined as the sum of the three cryptographic objects specified in Algorithm 19 of the Verificatum specification [31]: the permutation commitment μ (consisting of n group elements), the commitment τ^{pos} (containing $2n+5$ group elements), and the response σ^{pos} (containing $2n+4$ scalars). For the P-256 instantiation, these components correspond to the files `permutationCommitment01.bt`, `PoSCommitment01.bt`, and `PoSReply01.bt` produced by Verificatum.

To isolate the cost of the argument itself, we exclude auxiliary data found in Verificatum's output directory, such as input/output ciphertexts, public keys, and protocol metadata. While necessary for execution, these artifacts are not part of the cryptographic proof. Under this definition, shuffling $n = 10^6$ ciphertexts yields a proof of approximately 304.2 MB, distinct from the ≈ 805 MB total directory size reported by Verificatum's command-line interface. This methodology mirrors our own reporting, which includes only serialized proof elements.

At $n = 10^6$, Verificatum requires 304.2 MB, while MAYA-Ristretto ($k = 4$) requires 6.6 KB and 7.5 KB for MAYA-P256, a reduction factor of approximately 4.6×10^4.

4.4 Handling Arbitrary Input Sizes

Standard k-ary folding requires $n = k^d$ for some integer d. Padding to the next power of k can multiply the prover work by a factor of k in the worst case (e.g., $n = 2^{20} + 1$ requires padding to 2^{21}). We address this via two complementary techniques: *sparse padding* to reduce the cost of dummy elements, and *iterative padding* to minimize how many dummy elements are needed in the first place.

Sparse padding. Dummy ciphertexts are represented as identity elements $(1_{\mathbb{G}}, 1_{\mathbb{G}})$, which can be skipped in MSM computations via $1_{\mathbb{G}}^x = 1_{\mathbb{G}}$. The permutation for padded slots is publicly fixed to the identity. To ensure soundness, we enforce $e'_i = 0$ for each padded index $i > n$. Consequently, the input vectors of the IPA and eCP have length n_{padded}. A cost decomposition (Appendix P) shows that ciphertext-dependent operations, which benefit from sparsity, account for roughly one third of prover cost. The remaining two thirds are structure-dependent and scale with the padded size n_{padded}, making it advantageous to minimize n_{padded} directly.

Iterative padding. This motivates our second technique. Rather than padding to k^d at initialization (Global Pre-Padding, Appendix A), we pad incrementally at each recursion step to the nearest multiple of k, bounding the total overhead to at most $(k-1) \cdot d$ elements regardless of how n aligns with powers of k (Sections 2.5, 2.6).

Results. We benchmark both strategies for $n = 2^{20} + 1$ across $k \in \{2, 3, 4, 5\}$ and varying recursion depths d (Table 12, Appendix Q). We restrict to small k values as explained in Section 4.2. Timings report only the proof generation step, since this is where padding has the greatest impact. Global pre-padding exhibits cliff behavior: at maximal depth, it increases prover time by 34–65% depending on k. Iterative padding eliminates this cliff by adding at most $(k-1) \cdot d \leq 30$ dummy elements.

To quantify the total benefit, naïve global padding at $k = 2$, $d = 20$ requires 397 s and $1{,}048{,}575$ dummy elements. In contrast, iterative padding at $k = 4$, $d = 10$ achieves 215 s with 30 dummy elements, representing 46% speedup with approximately 35,000× fewer dummies. Compared to the ideal power-of-k baseline ($n = 2^{20}$), which averages ≈215 s, the iterative approach incurs essentially no overhead, confirming that it effectively neutralizes misalignment penalties.

Table 3: MAYA's performance at scale for $n = 10^6$ ciphertexts in the multi-threaded setting.

	Time (s)			Improvement	
Scheme	$\mathcal{P}$	$\mathcal{V}$	Size	$\mathcal{V}$	Size
Verificatum	183.8	168.5	304 MB	Baseline	
MAYA-Ristretto	120.3	17.1	6.6 KB	9.8×	47,000×
MAYA-P256	148.0	21.6	7.5 KB	7.8×	41,000×

Crucially, with iterative padding, prover time is nearly independent of recursion depth: for $k = 4$, increasing from $d = 3$ (proof size 1.5 MB) to $d = 10$ (proof size 6.7 KB) changes runtime by less than 6%. Early stopping at $d = 3$ would inflate the proof by 230× while saving only 1.4% of runtime. This implies that early stopping (large m, shallow d) does *not* yield a meaningful speed advantage.

We conclude that iterative padding with maximal recursion depth ($d = \lceil \log_k n \rceil$) is the recommended strategy: it maintains the expected prover time regardless of input alignment and enables aggressive proof minimization (6.7 KB for MAYA-Ristretto, 7.5 KB for MAYA-P256) without computational penalty.

4.5 Summing-up our findings

Based on the preceding analysis, we recommend small folding factor k (e.g., $k \leq 8$) with iterative padding and maximal recursion depth ($d = \lceil \log_k n \rceil$) for production deployments. For arbitrary input sizes, whether processing an entire election monolithically or handling the final batch of a partitioned set, iterative padding with sparse dummy ciphertexts eliminates worst-case performance cliffs.

Tables 3 provides a comprehensive snapshot of the performance of MAYA for $n = 10^6$ ciphertexts (relevant for national elections) with $k = 4$ using the iterative padding and sparse representation strategies (Section 4.4).

MAYA-Ristretto proves in 120.3 s and verifies in 17.1 s (9.8× faster than Verificatum), while reducing proof size from 304 MB to 6.6 KB. MAYA-P256 benefits even more from parallelism than either MAYA-Ristretto or Verificatum, due to Arkworks' native parallel MSM implementation, which helps to reduce prover time to 148 s, competitive with Verificatum's 183.8 s. The MAYA-P256 verifier completes in 21.6 s (7.8× faster than Verificatum), confirming that the verification speedup is robust across both curve implementations.

Proof size is independent of the threading model and depends on the folding factor and recursion depth. The modest difference between MAYA-Ristretto (6.6 KB) and MAYA-P256 (7.5 KB) arises from the compressed point encoding: Ristretto255 points are 32 bytes, whereas compressed P-256 points (Short Weierstrass) are 33 bytes.

Further engineering considerations, including precomputation opportunities and their trade-offs, are detailed in Appendix R.

5 Conclusion

MAYA offers a new efficient logarithmic-size shuffle argument with a transparent setup, together with two implementations: MAYA-Ristretto (Curve25519), our primary implementation, and MAYA-P256 (P-256). Both offer substantial benefits compared to the state-of-the-art libraries deployed in practice: argument size and verification time are considerably reduced, while the argument computation time remains similar to the best current solutions.

MAYA's argument computation requirements are quite practical: on a single core of a standard laptop, MAYA-Ristretto computes a shuffle argument for a million ECC ciphertexts in approximately 4 minutes. With multi-threading (12 threads), this falls to about 2 minutes.

Implementing a verifier can also be a fairly straightforward task: by relying on the `curve25519-dalek` library supporting efficient elliptic curve and multi-scalar multiplication operations, our MAYA verifier requires roughly 1,080 lines of Rust code (including I/O) and can verify a ~7 KB shuffle argument for a million ciphertexts in 17 s on a standard laptop. Short arguments and fast verification enable any citizen to conveniently download and verify the argument.

We evaluated our MAYA implementation on a broad range of parameters, including padding strategies for input sizes where the ciphertext list is not a power of the folding factor, and showed that iterative padding with sparse dummy ciphertexts eliminates the performance cliffs of standard k-ary folding, yielding predictable prover time for arbitrary input sizes with essentially no overhead at large scale.

Overall, MAYA reduces the shuffle proof from the dominant component of election audit data to a negligible fraction, making universal verifiability considerably more practical and cheaper at national scale.

References

[1] Oleg Andreev. Non-power-of-two inner product proof. GitHub Issue #198, dalek-cryptography/bulletproofs, 2019.

[2] Oleg Andreev, Henry de Valence, and Cathie Yun. dalek-cryptography bulletproofs. `https://github.com/dalek-cryptography/bulletproofs`, 2018.

[3] Diego F. Aranha and Conrado P. L. Gouvêa. RELIC is an efficient LIbrary for cryptography. `https://github.com/relic-toolkit/relic`, 2009.

[4] Arkworks Contributors. ark-secp256r1: Arkworks ecosystem library for the P-256 (secp256r1) curve. `https://crates.io/crates/ark-secp256r1`, 2023.

[5] Artifacts. `https://github.com/uclcrypto/maya_shuffle_proof`, 2026.

[6] Thomas Attema. Compressed σ-protocol theory. 2023.

[7] Stephanie Bayer and Jens Groth. Efficient zero-knowledge argument for correctness of a shuffle. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 263–280. Springer, 2012.

[8] Josh Benaloh, Tal Moran, Lee Naish, Kim Ramchen, and Vanessa Teague. Shuffle-sum: coercion-resistant verifiable tallying for STV voting. *IEEE Trans. Inf. Forensics Secur.*, 4(4):685–698, 2009.

[9] David Bernhard, Olivier Pereira, and Bogdan Warinschi. How not to prove yourself: Pitfalls of the fiat-shamir heuristic and applications to helios. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 626–643. Springer, 2012.

[10] Jonathan Bootle, Andrea Cerulli, Pyrros Chaidos, Jens Groth, and Christophe Petit. Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 327–357. Springer, 2016.

[11] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE symposium on security and privacy (SP)*, pages 315–334. IEEE, 2018.

[12] Véronique Cortier, Alexandre Debant, Olivier Esseiva, Pierrick Gaudry, Audhild Høgåsen, and Chiara Spadafora. A practical and fully distributed e-voting protocol for the swiss context. Cryptology ePrint Archive, Paper 2025/1625, 2025.

[13] Edouard Cuvelier, Olivier Pereira, and Thomas Peters. Election verifiability or ballot privacy: Do we need to choose? In *European Symposium on Research in Computer Security*, pages 481–498. Springer, 2013.

[14] Henri Devillez, Olivier Pereira, and Thomas Peters. How to verifiably encrypt many bits for an election? In *European Symposium on Research in Computer Security*, pages 653–671. Springer, 2022.

[15] Henri Devillez, Olivier Pereira, and Thomas Peters. Traceable receipt-free encryption. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 273–303. Springer, 2022.

[16] Thi Van Thao Doan, Olivier Pereira, and Thomas Peters. Threshold receipt-free single-pass evoting. In *International Joint Conference on Electronic Voting*, pages 20–36. Springer, 2024.

[17] Taher A ElGamal. *Cryptography and logarithms over finite fields*. Stanford University, 1984.

[18] Prastudy Fauzi, Helger Lipmaa, Janno Siim, and Michal Zajac. An efficient pairing-based shuffle argument. In *Advances in Cryptology - ASIACRYPT 2017*, Lecture Notes in Computer Science, pages 97–127. Springer, 2017.

[19] Jens Groth. Linear algebra with sub-linear zero-knowledge arguments. In *Annual International Cryptology Conference*, pages 192–208. Springer, 2009.

[20] Jens Groth. A verifiable secret shuffle of homomorphic encryptions. *J. Cryptol.*, 23(4):546–579, 2010.

[21] Jens Groth and Yuval Ishai. Sub-linear zero-knowledge argument for correctness of a shuffle. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 379–396. Springer, 2008.

[22] Max Hoffmann, Michael Klooß, and Andy Rupp. Efficient zero-knowledge arguments in the discrete log setting, revisited. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2093–2110, 2019.

[23] Max Hoffmann and Michael Klooß. QESA_ZK: Implementation of efficient zero-knowledge arguments. `https://github.com/emsec/QESA_ZK`, 2019.

[24] Yehuda Lindell. Parallel coin-tossing and constant-round secure two-party computation. *Journal of Cryptology*, 16(3):143–184, 2003.

[25] C Andrew Neff. A verifiable secret shuffle and its application to e-voting. In *Proceedings of the 8th ACM conference on Computer and Communications Security*, pages 116–125, 2001.

[26] Swiss Post. New swiss post e-voting system: cryptographic protocol disclosed, 2021.

[27] Björn Terelius and Douglas Wikström. Proofs of restricted shuffles. In *International Conference on Cryptology in Africa*, pages 100–113. Springer, 2010.

[28] The Ethereum Foundation Cryptography Research Team. Curdleproofs: A shuffle argument protocol. `https://github.com/asn-d6/curdleproofs/`, 2022.

[29] Douglas Wikström. A commitment-consistent proof of a shuffle. In *Australasian Conference on Information Security and Privacy*, pages 407–421. Springer, 2009.

[30] Douglas Wikström. Complexity analysis of the verificatum mix-net (vmn version 3.1.0). Technical report, Verificatum AB, 2022. Technical report.

[31] Douglas Wikström. How to implement a stand-alone verifier for the verificatum mix-net. Technical report, Verificatum, 2022.

[32] Douglas Wikström. Verificatum mix-net user manual (version 3.1.0). `https://www.verificatum.org/files/vmnum-3.1.0.pdf`, 2022. Accessed: 2026-04-25.

[33] Douglas Wikström. Verificatum mixnet. `https://www.verificatum.org`, 2022.

[34] Douglas Wikström and Verificatum Contributors. Verificatum Mix-Net (VMN). `https://github.com/verificatum/verificatum-vmn`, 2022.

A Global Pre-Padding

Algorithm 1 $\mathsf{IPA}^{*}_{k,m}$: Inner Product Argument with Global Pre-Padding

Input: $(\boldsymbol{g}, \boldsymbol{h} \in \mathbb{G}^n, u, P \in \mathbb{G}; \boldsymbol{a}, \boldsymbol{b} \in \mathbb{Z}_p^n)$

Parameters: Folding factor $k \geq 2$, recursion depth $d \geq 1$, base case size $m \geq 1$ such that $k^d \cdot m \geq n$.

Initialization Phase:

(i) Compute target padded size: $n' = k^d \cdot m$.

(ii) Extend all vectors to size n':

$$\boldsymbol{g} \leftarrow \boldsymbol{g} \| \boldsymbol{g}_{\text{pad}}, \quad \boldsymbol{h} \leftarrow \boldsymbol{h} \| \boldsymbol{h}_{\text{pad}}, \quad \boldsymbol{a} \leftarrow \boldsymbol{a} \| \boldsymbol{0}^{n'-n}, \quad \boldsymbol{b} \leftarrow \boldsymbol{b} \| \boldsymbol{0}^{n'-n}$$

where $\boldsymbol{g}_{\text{pad}}, \boldsymbol{h}_{\text{pad}} \in \mathbb{G}^{n'-n}$ are derived deterministically
from public parameters (e.g., via hash-to-curve from the transcript).

(iii) Set $n \leftarrow n'$ and initialize round counter $r \leftarrow 0$.

1. Base Case $(r = d)$:

$\mathcal{P} \rightarrow \mathcal{V}$: Sends $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{Z}_p^m$.

$\mathcal{V}$ checks if $P = \boldsymbol{g}^{\boldsymbol{a}} \boldsymbol{h}^{\boldsymbol{b}} u^{\langle \boldsymbol{a}, \boldsymbol{b} \rangle}$.

If equal, accept; otherwise, reject. **Stop.**

2. Recursion $(r < d)$:

(a) Let $n' = n/k$. Partition vectors into k contiguous chunks of size n':
$\boldsymbol{g} = (\boldsymbol{g}_1, \ldots, \boldsymbol{g}_k)$ where $\boldsymbol{g}_i = (g_{(i-1)n'+1}, \ldots, g_{in'})$,
and similarly for $\boldsymbol{h}, \boldsymbol{a}, \boldsymbol{b}$.

(b) $\mathcal{P}$ computes cross terms for $\ell \in \{-(k-1), \ldots, k-1\} \setminus \{0\}$:

$$U'_\ell = \prod_{j-i=\ell} \boldsymbol{g}_j^{\boldsymbol{a}_i}, \quad U''_\ell = \prod_{j-i=\ell} \boldsymbol{h}_j^{\boldsymbol{b}_i}, \quad v_\ell = \sum_{j-i=\ell} \langle \boldsymbol{a}_i, \boldsymbol{b}_j \rangle$$

Then computes: $U_\ell = U'_\ell \cdot U''_{-\ell} \cdot u^{v_\ell}$.

(c) $\mathcal{P} \rightarrow \mathcal{V}$: Sends $\{U_\ell\}_{\ell \neq 0}$ (note: $U_0 = P$).

(d) $\mathcal{V} \rightarrow \mathcal{P}$: Sample challenge $x \xleftarrow{\$} \mathbb{Z}_p^*$.

(e) Both parties fold generators and commitment:

$$\boldsymbol{g}' = \prod_{i=1}^{k} \boldsymbol{g}_i^{x^{k-i}}, \quad \boldsymbol{h}' = \prod_{i=1}^{k} \boldsymbol{h}_i^{x^{i-1}}, \quad P' = P^{x^{k-1}} \cdot \prod_{\ell \neq 0} U_\ell^{x^{k-\ell-1}}$$

(f) $\mathcal{P}$ folds witnesses:

$$\boldsymbol{a}' = \sum_{i=1}^{k} x^{i-1} \boldsymbol{a}_i, \quad \boldsymbol{b}' = \sum_{i=1}^{k} x^{k-i} \boldsymbol{b}_i$$

(g) Update: $\boldsymbol{g} \leftarrow \boldsymbol{g}', \boldsymbol{h} \leftarrow \boldsymbol{h}', P \leftarrow P', \boldsymbol{a} \leftarrow \boldsymbol{a}', \boldsymbol{b} \leftarrow \boldsymbol{b}', n \leftarrow n', r \leftarrow r + 1$.

(h) Go to **Step 1** if $r = d$; otherwise, go to **Step 2**.

B Proof of Theorem 1

Proof. **Perfect Completeness.** Perfect completeness follows by inspection. An honest prover with a valid witness performs the algebraic operations described; since the padding and folding operations preserve the inner product relation $c' = \langle \boldsymbol{a}', \boldsymbol{b}' \rangle$ and the commitment structure $P' = (\boldsymbol{g}')^{\boldsymbol{a}'} (\boldsymbol{h}')^{\boldsymbol{b}'} u^{c'}$ at every step, the base case verification succeeds with probability 1.

Computational Witness-Extended Emulation. We construct an expected polynomial-time knowledge extractor $\mathcal{E}$ by induction on the recursion structure. We rely on the $(2k-1)$-special soundness of the split-and-fold protocol, as established by Attema et al. [6].

Let d denote the total number of recursive rounds. For round $t \in \{0, \ldots, d-1\}$, let n_t denote the dimension after padding (so n_0 is the padded initial dimension, and $n_t = n_{t-1}/k$ after folding).

Base Case ($n \leq m$). The prover sends $(\boldsymbol{a}, \boldsymbol{b})$ directly. The extractor $\mathcal{E}$ simply outputs these vectors. If the verifier accepts, the relation $P = \boldsymbol{g}^{\boldsymbol{a}} \boldsymbol{h}^{\boldsymbol{b}} u^{\langle \boldsymbol{a}, \boldsymbol{b} \rangle}$ holds by definition.

Recursive Extraction ($n > m$). Assume we have an extractor $\mathcal{E}_{n'}$ for dimension $n' = \lceil n/k \rceil$ that uses $T(n')$ transcripts. We construct $\mathcal{E}_n$ using $(2k-1) \cdot T(n')$ transcripts as follows. The extractor $\mathcal{P}^*$ to receive $\{U_\ell\}_{\ell \neq 0}$, then it rewinds $\mathcal{P}^*$ to obtain $(2k-1)$ accepting transcripts with distinct challenges $x_1, \ldots, x_{2k-1} \in \mathbb{Z}_p^*$.

For each x_j, the recursive protocol produces an instance at dimension n'. By the inductive hypothesis, $\mathcal{E}_{n'}$ extracts witnesses $(\mathbf{a}'_j, \boldsymbol{b}'_j)$ such that:

$$P'_j = (\boldsymbol{g}'_j)^{\boldsymbol{a}'_j} (\boldsymbol{h}'_j)^{\boldsymbol{b}'_j} u^{\langle \boldsymbol{a}'_j, \boldsymbol{b}'_j \rangle} \tag{37}$$

where $P'_j, \boldsymbol{g}'_j, \boldsymbol{h}'_j$ are the folded values using challenge x_j.

For each $j \in [2k-1]$, equation (37) can be rewritten as:

$$\prod_{\ell=-(k-1)}^{k-1} U_\ell^{x_j^{k-1-\ell}} = \prod_{i,s=1}^{k} \boldsymbol{g}_s^{x_j^{k-s+i-1} \boldsymbol{a}'_j[i]} \cdot \boldsymbol{h}_i^{x_j^{i-1+k-s} \boldsymbol{b}'_j[s]} \cdot \boldsymbol{u}^{x_j^{k-1+i-s} \langle \boldsymbol{a}'_j[i], \boldsymbol{b}'_j[s] \rangle}$$

where $\boldsymbol{a}'_j[i]$ denotes the i-th n'-sized block of the reconstructed witness. Taking logarithms (in the exponent), for each generator $g_{s,\nu}$ (the ν-th component of $\boldsymbol{g}_s$):

$$\sum_\ell (k-1-\ell) \cdot [\log U_\ell]_{g_{s,\nu}} \cdot x_j^{k-1-\ell} = \sum_i (k-s+i-1) \cdot a'_{j,i,\nu} \cdot x_j^{k-s+i-1}$$

This is a polynomial identity in x_j of degree $2(k-1)$.

Define vectors $\boldsymbol{c}^{(g)}, \boldsymbol{c}^{(h)}, \boldsymbol{c}^{(u)} \in \mathbb{Z}_p^{2k-1}$ as the coefficients of powers $x^0, x^1, \ldots, x^{2k-2}$ on the left-hand side (from U_ℓ openings). Define $\boldsymbol{d}_j^{(g)}, \boldsymbol{d}_j^{(h)}, \boldsymbol{d}_j^{(u)}$ as the corresponding coefficients from the extracted witnesses $(\boldsymbol{a}'_j, \boldsymbol{b}'_j)$. For each $j \in [2k-1]$, we have the polynomial evaluation:

$$\sum_{t=0}^{2k-2} c_t^{(g)} \cdot x_j^t = \sum_{t=0}^{2k-2} d_{j,t}^{(g)} \cdot x_j^t$$

This gives us a system of $(2k-1)$ equations in $(2k-1)$ unknown coefficients. The Vandermonde matrix $V \in \mathbb{Z}_p^{(2k-1)\times(2k-1)}$ with $V_{jt} = x_j^t$ is invertible when all x_j are distinct. If $\boldsymbol{c}^{(g)} \neq \boldsymbol{d}_j^{(g)}$ for some j after solving, we obtain a non-trivial discrete logarithm relation.

From the coefficient structure, we can extract $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k$ and $\boldsymbol{b}_1, \ldots, \boldsymbol{b}_k$ such that:

$$\boldsymbol{a}'_j = \sum_{i=1}^{k} x_j^{i-1} \boldsymbol{a}_i, \quad \boldsymbol{b}'_j = \sum_{i=1}^{k} x_j^{k-i} \boldsymbol{b}_i$$

This is a Vandermonde system with k unknowns and $(2k-1)$ equations. We select any k challenges to form an invertible $k \times k$ Vandermonde submatrix and solve.

Now, we show $P = \boldsymbol{g}^{\boldsymbol{a}} \boldsymbol{h}^{\boldsymbol{b}} u^{\langle \boldsymbol{a}, \boldsymbol{b} \rangle}$ where $\boldsymbol{a} = (\boldsymbol{a}_1 \| \cdots \| \boldsymbol{a}_k)$, $\boldsymbol{b} = (\boldsymbol{b}_1 \| \cdots \| \boldsymbol{b}_k)$. Indeed, the coefficient of x^{k-1} in the polynomial identity (corresponding to $\ell = 0$) gives:

$$U_0 = \prod_{i=1}^{k} \boldsymbol{g}_i^{\boldsymbol{a}_i} \boldsymbol{h}_i^{\boldsymbol{b}_i} u^{\langle \boldsymbol{a}_i, \boldsymbol{b}_i \rangle}$$

Since $U_0 = P$ by construction, we have: $P = \boldsymbol{g}^{\boldsymbol{a}} \boldsymbol{h}^{\boldsymbol{b}} u^{\sum_{i=1}^{k} \langle \boldsymbol{a}_i, \boldsymbol{b}_i \rangle}$.

For the inner product, the verifier checks $c'_j = \langle \boldsymbol{a}'_j, \boldsymbol{b}'_j \rangle$. The extractor must ensure $c = \langle \boldsymbol{a}, \boldsymbol{b} \rangle$. Consider the polynomial $Q(x) = \langle \boldsymbol{a}'(x), \boldsymbol{b}'(x) \rangle$. The coefficient of $x^{k-1-\ell}$ in $Q(x)$ is exactly $v_\ell = \sum_{j-i=\ell} \langle \boldsymbol{a}_i, \boldsymbol{b}_j \rangle$. From the polynomial identity established above, the term $u^{Q(x_j)}$ must match the u-component of the right-hand side commitments for all j. This implies the coefficients of the polynomial in the exponent match. Specifically, summing v_ℓ over all ℓ gives $\sum_{i,j} \langle \boldsymbol{a}_i, \boldsymbol{b}_j \rangle = \langle \boldsymbol{a}, \boldsymbol{b} \rangle$. Thus, the witness $(\boldsymbol{a}, \boldsymbol{b})$ satisfies the relation for the commitment P and scalar c.

Runtime and Error. Let $T(n)$ denote the number of transcripts needed for extraction at dimension n. In the base case, $T(n) = 1$. In the recursive case, $T(n) = (2k-1) \cdot T(n/k)$. The recurrence is $T(n) = (2k-1)^d \approx (n/m)^{\log_k(2k-1)}$ with $d = \lceil \log_k(n/m) \rceil$. This is polynomial in n, so the extractor runs in expected polynomial time. The probability of extraction failure (i.e., all $(2k-1)$ rewinds produce the same challenge) is at most:

$$\kappa \leq \frac{(2k-2)}{|\mathbb{Z}_p|} \cdot d$$

which is negligible for $|\mathbb{Z}_p| \approx 2^{256}$. □

C Aggregated Inner Product Argument

Protocol C (Aggregated Inner Product Argument). Consider a prover $\mathcal{P}$ and a verifier $\mathcal{V}$ with common input statements $\{(P_i, c_i)\}_{i=1}^m$ and generator parameters $\boldsymbol{g}, \boldsymbol{h} \in \mathbb{G}^n$, the prover claims knowledge of witnesses $\{(\boldsymbol{a}_i, \boldsymbol{b}_i)\}_{i=1}^m$ such that:

$$P_i = \boldsymbol{g}^{\boldsymbol{a}_i} \boldsymbol{h}^{\boldsymbol{b}_i} \quad \text{and} \quad c_i = \langle \boldsymbol{a}_i, \boldsymbol{b}_i \rangle \quad \forall i \in [m]. \tag{38}$$

The protocol proceeds as follows:

1. The prover sends all the cross-terms of the polynomial $c(x)$, i.e., $\langle \boldsymbol{a}_i, \boldsymbol{b}_j \rangle$ for $i, j \in [m]$, $i \neq j$, to the verifier.

2. The verifier samples a challenge $x \leftarrow \mathbb{Z}_p$ and send it to the prover.

3. The prover defines the aggregated witnesses as polynomials in x:

$$\boldsymbol{a}(x) = \sum_{i=1}^{m} x^{i-1}\boldsymbol{a}_i, \quad \boldsymbol{b}(x) = \sum_{i=1}^{m} x^{i-1}\boldsymbol{b}_i$$

4. Both parties compute the aggregated statement:

$$P(x) = \prod_{i=1}^{m} P_i^{x^{i-1}}, \quad c(x) = \langle \boldsymbol{a}(x), \boldsymbol{b}(x) \rangle.$$

5. Both parties execute the sub-routine $\mathsf{IPA}(\boldsymbol{g}, \boldsymbol{h}, P(x), c(x); \boldsymbol{a}(x), \boldsymbol{b}(x))$.

Theorem 4. *The Aggregated* IPA *presented in Protocol C has perfect completeness and computational witness-extended-emulation (WEE) assuming the hardness of the Discrete Logarithm problem and the security properties of the underlying* IPA.

C.1 Completeness

Proof. Suppose an honest prover knows correct witnesses $\{(\boldsymbol{a}_i, \boldsymbol{b}_i)\}_{i=1}^{m}$. First, we verify the consistency of the aggregated commitment $P(x)$:

$$\begin{aligned} \boldsymbol{g}^{\boldsymbol{a}(x)}\boldsymbol{h}^{\boldsymbol{b}(x)} &= \boldsymbol{g}^{\sum_{i=1}^{m} x^{i-1}\boldsymbol{a}_i}\boldsymbol{h}^{\sum_{i=1}^{m} x^{i-1}\boldsymbol{b}_i} \\ &= \prod_{i=1}^{m} \left(\boldsymbol{g}^{\boldsymbol{a}_i}\boldsymbol{h}^{\boldsymbol{b}_i}\right)^{x^{i-1}} \\ &= \prod_{i=1}^{m} P_i^{x^{i-1}} = P(x) \end{aligned}$$

Second, we verify the scalar aggregation. The prover calculates $c(x)$ based on the honest witnesses: $c(x) = \langle \sum_{i=1}^{m} x^{i-1}\boldsymbol{a}_i, \sum_{j=1}^{m} x^{j-1}\boldsymbol{b}_j \rangle$, which can be publicly computed from the cross-terms of the polynomial $c(x)$.

Since the prover honestly executes the underlying IPA for inputs $\boldsymbol{a}(x), \boldsymbol{b}(x)$, and the underlying IPA satisfies completeness, the verifier will accept with probability 1. □

C.2 Computational Witness Extended Emulation

Proof. We construct a knowledge extractor $\mathcal{E}$ that interacts with a convincing prover $\mathcal{P}^*$. $\mathcal{E}$ utilizes the extractor $\mathcal{X}^{\mathsf{IPA}}$ of the underlying IPA protocol as a subroutine.

Since the aggregated witnesses $\boldsymbol{a}(x)$ and $\boldsymbol{b}(x)$ are constructed as polynomials of degree $m-1$ in the challenge x, extraction is achieved by rewinding $\mathcal{P}^*$ to obtain transcripts for m distinct challenges $\{x_k\}_{k=1}^{m} \subset \mathbb{Z}_p$. For each x_k, $\mathcal{E}$ executes $\mathcal{P}^*$ up to the IPA phase and invokes $\mathcal{X}^{\mathsf{IPA}}$. By the WEE of the underlying IPA, this yields vectors $\boldsymbol{A}_k, \boldsymbol{B}_k \in \mathbb{Z}_p^n$ satisfying:

$$\boldsymbol{g}^{\boldsymbol{A}_k}\boldsymbol{h}^{\boldsymbol{B}_k} = P(x_k) = \prod_{i=1}^{m} P_i^{x_k^{i-1}}. \tag{39}$$

To recover the individual witnesses, we observe that the extracted vectors $\boldsymbol{A}_k$ correspond to evaluations of the witness polynomial at x_k. We can express the relationship between the extracted values and the target witnesses $\boldsymbol{a}_i$ via the Vandermonde matrix $V \in \mathbb{Z}_p^{m \times m}$, where $V_{kj} = x_k^{j-1}$. Since the challenges are distinct, $\det(V) \neq 0$. $\mathcal{E}$ computes the witnesses by inverting the linear system:

$$(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_m)^\top = V^{-1}(\boldsymbol{A}_1, \ldots, \boldsymbol{A}_m)^\top. \tag{40}$$

The same interpolation strategy is applied to recover $\boldsymbol{b}_i$ from $\boldsymbol{B}_k$.

It remains to show that the extracted witnesses satisfy the statements P_i and inner product relations c_i for $i \in [m]$. For the statements, substituting the interpolated values into the extraction equation yields:

$$\prod_{i=1}^{m} \left(\boldsymbol{g}^{\boldsymbol{a}_i} \boldsymbol{h}^{\boldsymbol{b}_i}\right)^{x_k^{i-1}} = \prod_{i=1}^{m} P_i^{x_k^{i-1}}. \tag{41}$$

Since this equality holds for m distinct points and $m < p$, the polynomials in the exponents are identical. Under the Discrete Logarithm assumption, this implies term-wise equality $\boldsymbol{g}^{\boldsymbol{a}_i} \boldsymbol{h}^{\boldsymbol{b}_i} = P_i$ for all $i \in [m]$.

Finally, we verify the inner product consistency. The prover asserts a scalar value $c(x)$ satisfying $c(x) = \langle \boldsymbol{a}(x), \boldsymbol{b}(x) \rangle$. Note that the RHS is a polynomial of degree $2m - 2$. By extending the rewinding strategy to $2m - 1$ points (or invoking the Schwartz-Zippel lemma), we establish that this polynomial equality holds identically. The coefficient of $x^{2(i-1)}$ in the expansion corresponds strictly to the term $\langle \boldsymbol{a}_i, \boldsymbol{b}_i \rangle$. Therefore, the extracted witnesses satisfy $c_i = \langle \boldsymbol{a}_i, \boldsymbol{b}_i \rangle$, completing the proof. □

D Proof of Theorem 2

Proof. **Perfect Completeness.** We verify that an honest prover with a valid witness $\mathbf{a}$ always convinces the verifier.

Base Case ($n \leq m$). The prover sends $\mathbf{a} \in \mathbb{Z}_p^n$ directly. The verifier checks $P_i = \mathbf{g}_i^{\mathbf{a}}$ for all $i \in [\gamma]$, which holds by the premise of the relation.

Recursive Case ($n > m$). Assume completeness holds for dimension $n' = \lceil n/k \rceil$.

If $n \not\equiv 0 \pmod{k}$, let $\delta = k - (n \bmod k)$. The prover pads the witness $\tilde{\mathbf{a}} = \mathbf{a} \| \mathbf{0}^\delta$. The generators are padded asymmetrically:

- Row 1: $\tilde{\mathbf{g}}_1 = \mathbf{g}_1 \| \mathbf{g}_{1,\text{pad}}$, where $\mathbf{g}_{1,\text{pad}}$ are arbitrary group elements.
- Rows $i \geq 2$: $\tilde{\boldsymbol{g}}_i = \boldsymbol{g}_i \| \mathbf{1}^\delta$, where $\mathbf{1}$ is the identity element $1_{\mathbb{G}}$.

For Row 1, the relation holds: $\tilde{\boldsymbol{g}}_1^{\tilde{\boldsymbol{a}}} = \boldsymbol{g}_1^{\boldsymbol{a}} \cdot \boldsymbol{g}_{1,\text{pad}}^{\mathbf{0}} = \boldsymbol{g}_1^{\boldsymbol{a}} = P_1$. For Rows $i \geq 2$, the relation also holds: $\tilde{\boldsymbol{g}}_i^{\tilde{\boldsymbol{a}}} = \boldsymbol{g}_i^{\boldsymbol{a}} \cdot \mathbf{1}^{\mathbf{0}} = \boldsymbol{g}_i^{\boldsymbol{a}} = P_i$. Thus, padding with either random generators or identity elements preserves the validity of the statement given the padded witness $\tilde{\boldsymbol{a}}$.

The rest of the completeness argument follows standard generalized folding logic. The cross-term commitments U_ℓ and folded witnesses $\boldsymbol{a}'$ are constructed homomorphically. Since the algebraic structure is preserved, the folded statement $P_i' = (\boldsymbol{g}_i')^{\boldsymbol{a}'}$ holds if and only if the original statement holds. By the inductive hypothesis, the recursive call succeeds.

Computational Witness-Extended Emulation. We construct an expected polynomial-time extractor $\mathcal{E}$ by induction on the recursion depth. The extraction strategy relies on the fact that Row 1 acts as a globally binding commitment to the witness, even if other rows use non-binding bases (e.g., identity elements).

Base Case ($n \leq m$). The prover sends $\boldsymbol{a}$ directly. The extractor outputs $\boldsymbol{a}$. If the verifier accepts, the relation holds by definition.

Recursive Case ($n > m$). Assume an extractor $\mathcal{E}_{n'}$ exists for dimension $n' = \lceil n/k \rceil$. $\mathcal{E}$ runs $\mathcal{P}^*$ to receive $\{U_{i,\ell}\}_{i\in[M],\ell\neq 0}$. It rewinds $\mathcal{P}^*$ to obtain $(2k-1)$ accepting transcripts with distinct challenges $x_1, \ldots, x_{2k-1} \in \mathbb{Z}_p^*$.

For each challenge x_j, the recursive protocol produces an instance at dimension n'. By the inductive hypothesis, $\mathcal{E}_{n'}$ extracts a witness $\boldsymbol{a}'_j$ such that:

$$P'_{i,j} = (\boldsymbol{g}'_{i,j})^{\boldsymbol{a}'_j} \quad \forall i \in [\gamma]$$

For each $j \in [2k-1]$, the folding relation for Row 1 gives:

$$P'_{1,j} = P_1^{x_j^{k-1}} \cdot \prod_{\ell\neq 0} U_{1,\ell}^{x_j^{k-\ell-1}} = (\boldsymbol{g}'_{1,j})^{\boldsymbol{a}'_j}$$

Expanding both sides as polynomials in x_j of degree $2(k-1)$, we can match coefficients. The extracted witnesses satisfy the folding relation: $\boldsymbol{a}'_j = \sum_{s=1}^{k} x_j^{s-1}\boldsymbol{a}_s$ This is a Vandermonde system. With $(2k-1) \geq k$ distinct challenges, we can select k challenges to form an invertible $k \times k$ Vandermonde matrix V with $V_{js} = x_j^{s-1}$, and solve:

$$(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k)^\top = V^{-1}(\boldsymbol{a}'_1, \ldots, \boldsymbol{a}'_k)^\top$$

From the polynomial identity (matching the coefficient of x^{k-1}, i.e., $\ell = 0$):

$$U_{1,0} = P_1 = \prod_{s=1}^{k} (\boldsymbol{g}_1)_s^{\boldsymbol{a}_s} = \boldsymbol{g}_1^{\boldsymbol{a}}$$

where $\boldsymbol{a} = (\boldsymbol{a}_1 \| \cdots \| \boldsymbol{a}_k)$.

If the polynomial coefficients don't match, we obtain a non-trivial DLR in $\boldsymbol{g}_1$. For rows $i \geq 2$, we must show $P_i = \boldsymbol{g}_i^{\boldsymbol{a}}$ using the *same* witness $\boldsymbol{a}$ extracted from Row 1. Apply the same polynomial analysis to row i:

$$P'_{i,j} = P_i^{x_j^{k-1}} \cdot \prod_{\ell\neq 0} U_{i,\ell}^{x_j^{k-\ell-1}}$$

Since the extracted $\boldsymbol{a}'_j$ satisfies $P'_{i,j} = (\boldsymbol{g}'_{i,j})^{\boldsymbol{a}'_j}$ (from the accepting transcript), and the folding is deterministic given $\boldsymbol{a}'_j$, the polynomial identity implies:

$$P_i = \prod_{s=1}^{k} (\boldsymbol{g}_i)_s^{\boldsymbol{a}_s} = \boldsymbol{g}_i^{\boldsymbol{a}}$$

If this fails for any row i, we would have two different witnesses $\boldsymbol{a}$ (from Row 1) and $\boldsymbol{a}^*$ (hypothetically from Row i) that both produce accepting proofs. But the witness is *shared* across all rows in the protocol, so this would require $\boldsymbol{g}_1^{\boldsymbol{a}} = \boldsymbol{g}_1^{\boldsymbol{a}^*}$ with $\boldsymbol{a} \neq \boldsymbol{a}^*$, yielding a DLR in $\boldsymbol{g}_1$.

The restriction of $\boldsymbol{a}$ to the original n positions (before padding) yields a valid witness for the original relation.

□

E Aggregated Chaum-Pedersen Argument

Protocol E (Aggregated Chaum-Pedersen Argument). The following is a protocol to prove $\exists \boldsymbol{a} : \boldsymbol{P} = \boldsymbol{G}^{\boldsymbol{a}}$. Common input is $(\boldsymbol{G}, \boldsymbol{P}) \in \mathbb{G}^{m\times n} \times \mathbb{G}^m$. The prover's witness is $\boldsymbol{a} \in \mathbb{Z}_p^n$. We denote the rows of $\boldsymbol{G}$ as $\boldsymbol{g}_1, \ldots, \boldsymbol{g}_m$, where each $\boldsymbol{g}_i \in \mathbb{G}^n$. As seen in Section 2.6, a crucial requirement is that the first row, $\boldsymbol{g}_1$, consists of generators sampled uniformly at random, such that their relative discrete logarithm relations are unknown.

1. The verifier sends a random challenge vector $\boldsymbol{x} = (x_2, \ldots, x_m) \in \mathbb{Z}_p^{m-1}$ to the prover.

2. Both parties construct $\boldsymbol{B} \in \mathbb{G}^{2\times n}$ and $\boldsymbol{u} \in \mathbb{G}^2$, defined as:

$$\boldsymbol{B} = \begin{pmatrix} \boldsymbol{g}_1 \\ \prod_{i=2}^m \boldsymbol{g}_i^{x_i} \end{pmatrix}, \quad \boldsymbol{u} = \begin{pmatrix} P_1 \\ \prod_{i=2}^m P_i^{x_i} \end{pmatrix}$$

3. Both parties execute the sub-routine $\mathsf{eCP}(\boldsymbol{B}, \boldsymbol{u}; \boldsymbol{a})$ for the relation $\boldsymbol{u} = \boldsymbol{B}^{\boldsymbol{a}}$.

Theorem 5. *The Aggregated Chaum-Pedersen Argument presented in Protocol E has perfect completeness and computational witness-extended-emulation (WEE), assuming the hardness of the DLR assumption and the security properties of the underlying* eCP.

E.1 Completeness

Proof. Completeness follows immediately from the linearity of the construction. Since $\boldsymbol{g}_1^{\boldsymbol{a}} = P_1$ by hypothesis, the first row of the compressed system is satisfied. For the second row, observing that $\boldsymbol{g}_i^{\boldsymbol{a}} = P_i$ for all i:

$$\left(\prod_{i=2}^m \boldsymbol{g}_i^{x_i}\right)^{\boldsymbol{a}} = \prod_{i=2}^m (\boldsymbol{g}_i^{\boldsymbol{a}})^{x_i} = \prod_{i=2}^m P_i^{x_i}.$$

Since $\boldsymbol{B}^{\boldsymbol{a}} = \boldsymbol{u}$, and the underlying eCP-protocol is perfectly complete, the verifier accepts the proof. □

E.2 Computational Witness-Extended Emulation

Proof. We construct an expected polynomial-time extractor $\mathcal{E}$ with oracle access to a prover $\mathcal{P}^*$. The extraction relies on a tree-based rewinding strategy to unbind the batching challenges.

Let $n_c = m - 1$ be the number of required batching challenges. The Extractor $\mathcal{E}$ invokes $\mathcal{P}^*$ to obtain an initial state. $\mathcal{E}$ then forks the execution n_c times. In each fork $k \in \{1, \ldots, n_c\}$, $\mathcal{E}$ sends a uniformly random batching challenge vector $\boldsymbol{x}^{(k)} = (x_2^{(k)}, \ldots, x_m^{(k)}) \in \mathbb{Z}_p^{m-1}$.

For each batching challenge $\boldsymbol{x}^{(k)}$, the parties engage in the sub-protocol eCP for the statement $\boldsymbol{B}^{(k)\boldsymbol{a}} = \boldsymbol{u}^{(k)}$. Since eCP has computational witness-extended-emulation (see Appendix D), $\mathcal{E}$ performs an inner rewinding on the eCP protocol to extract a witness $\boldsymbol{a}^{(k)} \in \mathbb{Z}_p^n$ such that $\boldsymbol{B}^{(k)\boldsymbol{a}^{(k)}} = \boldsymbol{u}^{(k)}$.

We first argue that the extracted witnesses must be identical across all branches. Indeed, for any branch k, the construction of $\boldsymbol{B}^{(k)}$ leaves the first row static: $\boldsymbol{g}_1^{\boldsymbol{a}^{(k)}} = P_1$. Suppose, for contradiction, that $\mathcal{E}$ extracts two distinct witnesses $\boldsymbol{a}^{(j)} \neq \boldsymbol{a}^{(k)}$. This implies $\boldsymbol{g}_1^{(\boldsymbol{a}^{(j)} - \boldsymbol{a}^{(k)})} = \boldsymbol{1}_{\mathbb{G}}$. Since $\boldsymbol{g}_1$ consists of random generators, finding such a vector $\delta = \boldsymbol{a}^{(j)} - \boldsymbol{a}^{(k)} \neq \boldsymbol{0}$ constitutes finding a non-trivial discrete logarithm relation. Under the DLR assumption, this occurs with negligible probability. Thus, we conclude $\boldsymbol{a}^{(1)} = \cdots = \boldsymbol{a}^{(m-1)} = \boldsymbol{a}^*$.

It remains to show that $\boldsymbol{a}^*$ satisfies the original unbatched relations. From the second row of the unified system, we have that for all k:

$$\left(\prod_{i=2}^m \boldsymbol{g}_i^{x_i^{(k)}}\right)^{\boldsymbol{a}^*} = \prod_{i=2}^m P_i^{x_i^{(k)}} \implies \prod_{i=2}^m \left(\boldsymbol{g}_i^{\boldsymbol{a}^*} \cdot P_i^{-1}\right)^{x_i^{(k)}} = \boldsymbol{1}_{\mathbb{G}}.$$

Since $\mathbb{G}$ is a cyclic group of prime order p, let g be a fixed generator of $\mathbb{G}$. Let $\Delta_i \in \mathbb{Z}_p$ be the discrete logarithm such that $g^{\Delta_i} = \boldsymbol{g}_i^{\boldsymbol{a}^*} \cdot P_i^{-1}$. The equation becomes a linear relation in the exponents:

$$\forall k: \quad \sum_{i=2}^m x_i^{(k)} \cdot \Delta_i \equiv 0 \pmod{p}.$$

We can express this system of $m-1$ equations as a matrix-vector product $X \cdot \boldsymbol{\Delta} = \mathbf{0}$, where $X_{k,i} = x_i^{(k)}$ represents the matrix of challenges. Since the challenges are sampled uniformly from $\mathbb{Z}_p$, X is invertible with probability $1 - \frac{m-1}{p}$ by the Schwartz-Zippel lemma. Multiplying by X^{-1}, we deduce that $\Delta_j = 0$ for all $j \in \{2, \ldots, m\}$, which implies $\boldsymbol{g}_j^{\boldsymbol{a}^*} = P_j$. Combined with the binding of the first row ($\boldsymbol{g}_1^{\boldsymbol{a}^*} = P_1$), $\boldsymbol{a}^*$ is a valid witness for the full statement. □

F Permutation Argument

We detail our permutation argument here and in Figure 6. The proof proceeds in two coupled steps, linked by a commitment T_2 to the problematic linear term. To preserve the flow of the presentation, we defer the construction of the weight matrices until the end of this section.

Step 1: Circuit Satisfiability. Let $\boldsymbol{z} \xleftarrow{\$} \mathbb{Z}_p^Q$ be the verifier's random linear-combination challenge. The prover defines $t_2 = \langle \boldsymbol{z}, \boldsymbol{W}_V \cdot \boldsymbol{e'} \rangle$ and commits to it as $T_2 = g^{t_2} h^{\tau_2}$ as specified in Eqs. (66)–(69). Using the witness $\boldsymbol{e'}$ (already committed in V), the prover derives the arithmetic-circuit wire assignments $(\boldsymbol{a}_L, \boldsymbol{a}_R, \boldsymbol{a}_O)$ and proves that the arithmetic-circuit relation $\mathcal{R}_{\mathsf{AC.1}}$ is satisfied, *assuming* that T_2 correctly commits to the linear term required by the circuit's linear constraints.

To keep the communication logarithmic, the prover replaces the explicit transmission of the vectors $\boldsymbol{l}$ and $\boldsymbol{r}$ (Eq. (79)) with a logarithmic-sized inner-product argument from Section 2.5. In particular, instead of explicitly checking $\hat{t} \stackrel{?}{=} \langle \boldsymbol{l}, \boldsymbol{r} \rangle$ and $P \stackrel{?}{=} h^{\mu} \cdot \boldsymbol{g}^{\boldsymbol{l}} \cdot \boldsymbol{h'}^{\boldsymbol{r}}$, the prover and verifier execute Eq. 87

$$\mathcal{P} \leftrightarrow \mathcal{V} : \mathsf{IPA}(\boldsymbol{g}, \boldsymbol{h'}, Ph^{-\mu}, \hat{t};, \boldsymbol{l}, \boldsymbol{r}).$$

At the end of this step, the verifier is convinced that the arithmetic circuit is satisfied *provided that* T_2 correctly commits to t_2.

Step 2: Binding T_2 to the Vector Commitment V. In the second step, the prover proves that the same witness $\boldsymbol{e'}$ committed in V was used to compute the value t_2 committed in T_2. This consistency requirement is captured by the relation

$$\mathcal{R}_{\mathsf{AC.2}} = \left\{ \begin{array}{l} (\mathsf{ck}, V, T_2 \in \mathbb{G}, \boldsymbol{z} \in \mathbb{Z}_p^Q, \boldsymbol{W}_V \in \mathbb{Z}_p^{Q \times n}; \\ \quad \boldsymbol{e'} \in \mathbb{Z}_p^n, s, \tau_2 \in \mathbb{Z}_p) : \\ V = \boldsymbol{g}^{\boldsymbol{e'}} h^s \ \wedge \ T_2 = g^{t_2} h^{\tau_2} \\ \wedge \ t_2 = \langle \boldsymbol{z}, \boldsymbol{W}_V \cdot \boldsymbol{e'} \rangle \end{array} \right\} \tag{42}$$

To prove this, we use a polynomial-based inner-product argument. The prover defines a masked vector polynomial $l_c(X) = \boldsymbol{e'} + \boldsymbol{s}'_L \cdot X \in Z_p^n[X]$ (Eq. 96) where $\boldsymbol{s}'_L \xleftarrow{\$} \mathbb{Z}_p^n$, and a public vector $\boldsymbol{r}_c = \langle \boldsymbol{z}, \boldsymbol{W}_V \rangle$, such that $\hat{t}_c(X) = \langle l_c(X), \boldsymbol{r}_c \rangle$ has the constant term t_2. The verifier samples a random challenge $x' \xleftarrow{\$} \mathbb{Z}_p$ and performs the following checks:

1. Correctness of t_2. The verifier checks if t_2 is indeed the constant term of $\hat{t}_c(X)$ using $g^{\hat{t}_c(x')} h^{\tau_c} = T_2 \cdot (T'_1)^{x'}$ (Eq. (112)).

2. Consistency with V. The prover computes $P' = V \cdot S'^{x'}$ (Eq. (113)), which evaluates the commitment to $l_c(X)$ at x'. The verifier checks $\boldsymbol{g}^{l_c(x')} = P' h^{-\mu_c}$, guaranteeing that $l_c(X)$ is well formed and bound to the witness $\boldsymbol{e'}$ committed in V.

Finally, an inner-product argument (Eq. (114)) is used to prove $\hat{t}_c(x') = \langle l_c(x'), \boldsymbol{r}_c \rangle$ without revealing the $O(n)$-size vector $l_c(x')$. Together, these checks ensure that both $\mathcal{R}_{\mathsf{AC.1}}$ and $\mathcal{R}_{\mathsf{AC.2}}$ can be satisfied only if the same witness vector $\boldsymbol{e}'$ is used consistently, thereby restoring soundness despite the use of a single aggregated vector commitment.

Weight matrices. The weight matrices $\boldsymbol{W}_L, \boldsymbol{W}_R, \boldsymbol{W}_O \in \mathbb{Z}_p^{Q \times (n-1)}$, $\boldsymbol{W}_V \in \mathbb{Z}_p^{Q \times n}$, and the constant vector $\boldsymbol{c} \in \mathbb{Z}_p^Q$ encode the $Q = 2n{-}1$ linear constraints described in Section 3.2 in the standard Bulletproofs constraint-system format [11]. Each multiplication gate $j \in [n{-}1]$ contributes variables $(a_{L,j},\ a_{R,j},\ a_{O,j})$ satisfying the Hadamard relation $\boldsymbol{a}_L \circ \boldsymbol{a}_R = \boldsymbol{a}_O$. The linear constraints, expressed as $\boldsymbol{W}_L\boldsymbol{a}_L + \boldsymbol{W}_R\boldsymbol{a}_R + \boldsymbol{W}_O\boldsymbol{a}_O = \boldsymbol{W}_V\boldsymbol{e}' + \boldsymbol{c}$, enforce the following wiring:

1. *Left-input wiring* ($n{-}1$ rows): row 1 sets $a_{L,1} = e'_1 - u$ (i.e., $W_L(1,1) = 1$, $W_V(1,1) = 1$, $c_1 = -u$); rows $j \geq 2$ chain $a_{L,j} = a_{O,j-1}$ (i.e., $W_L(j,j) = 1$, $W_O(j, j{-}1) = -1$).
2. *Right-input wiring* ($n{-}1$ rows): row $n{-}1{+}j$ sets $a_{R,j} = e'_{j+1} - u$ (i.e., $W_R(n{-}1{+}j,\ j) = 1$, $W_V(n{-}1{+}j,\ j{+}1) = 1$, $c_{n-1+j} = -u$).
3. *Output constraint* (1 row): row $2n{-}1$ sets $a_{O,n-1} = y_{\mathrm{LHS}}$ (i.e., $W_O(2n{-}1,\ n{-}1) = 1$, $c_{2n-1} = -y_{\mathrm{LHS}}$).

Together with the Hadamard relation, these constraints fully encode the permutation check Eq. 7.

G Proof of Theorem 3

We prove security of the shuffle protocol Π_{base} (Protocols 3.2 and 3.3). Security of the optimized protocol Π_{opt} (Section 3.4) then follows by composition (Section H).

The argument links the circuit satisfiability of the permutation network ($\mathcal{R}_{\mathsf{AC.1}}$) to the consistency of the re-randomized ciphertexts ($\mathcal{R}_{\mathsf{AC.3}}$) via the intermediate witness $\boldsymbol{e}'$ committed in V ($\mathcal{R}_{\mathsf{AC.2}}$).

G.1 Perfect Completeness

Proof. Completeness follows from the algebraic consistency of the honest prover's operations. Assume $\mathcal{P}$ possesses valid witnesses $(\boldsymbol{e}', r', s)$ where $\boldsymbol{e}'$ is a permutation of $\boldsymbol{e}$.

Relation $\mathcal{R}_{\mathsf{AC.1}}$. The honest prover constructs T_2 as a commitment to $t_2 = \langle \boldsymbol{z}, \boldsymbol{W}_V \cdot \boldsymbol{e}' \rangle$. The circuit is padded to length n_{padded} with constraints enforcing $e'_i = 0$ for $i > n$ (see Section 4.4).

Expanding the verification equation:

$$\begin{aligned} t_2^* &= \langle \boldsymbol{a}_L, \boldsymbol{a}_R \circ \boldsymbol{y}^n \rangle - \langle \boldsymbol{a}_O, \boldsymbol{y}^n \rangle + \langle \boldsymbol{z}, \boldsymbol{W}_L\boldsymbol{a}_L + \boldsymbol{W}_R\boldsymbol{a}_R + \boldsymbol{W}_O\boldsymbol{a}_O \rangle + \delta(y,z) \\ &= \delta(y,z) + \langle \boldsymbol{z}, \boldsymbol{c} + \boldsymbol{W}_V \cdot \boldsymbol{e}' \rangle \quad \text{(by linear circuit constraints)} \\ &= t_2 + \delta(y,z) + \langle \boldsymbol{z}, \boldsymbol{c} \rangle \end{aligned}$$

Since the honest prover computes $\hat{t}$ and τ_x to satisfy the polynomial identity $t(X) = \langle \boldsymbol{l}(X), \boldsymbol{r}(X) \rangle$, the verification checks 86 and 87 hold by the homomorphic properties of commitments and perfect completeness of the IPA (Theorem 1), respectively.

Relations $\mathcal{R}_{\mathsf{AC.2}}$ and $\mathcal{R}_{\mathsf{AC.3}}$. The verifier checks two conditions via sub-protocols:

1. $\mathsf{IPA}_{k,m}(\boldsymbol{g}, \boldsymbol{h}', P'h^{-\mu_c}\boldsymbol{h}'^{\boldsymbol{r}_c}, \hat{t}_c; \boldsymbol{l}_c, \boldsymbol{r}_c)$ (Eq. 114): Proves the inner product relation binding $\boldsymbol{e}'$ (in V) to t_2 (in T_2).

2. $\mathsf{eCP}_{k,m}(\boldsymbol{G}, \boldsymbol{P}'; \boldsymbol{l}_c)$ (Eq. 23) where $\boldsymbol{G} = (\boldsymbol{g}, \boldsymbol{c}'_1, \boldsymbol{c}'_2)^\top$ and $\boldsymbol{P}' = (P'h^{-\mu_c}, P'_1, P'_2)^\top$: Proves witness consistency across the commitment V and the aggregated ciphertexts.

The prover constructs arguments using $\boldsymbol{l}_c(X) = \boldsymbol{e}' + \boldsymbol{s}'_L X$ and the established aggregated relations. Since $\mathsf{IPA}_{k,m}$ and $\mathsf{eCP}_{k,m}$ satisfy completeness (Theorems 1 and 2), all verification steps succeed.

Padding Consistency. The circuit pads the witness to n_{padded}, enforcing zeros at padded positions. The sub-protocols $\mathsf{IPA}_{k,m}$ and $\mathsf{eCP}_{k,m}$ receive vectors of length n_{padded} directly, with no additional padding required at the IPA/eCP level (the circuit-level padding suffices). Any iterative padding within the recursive rounds of these sub-protocols preserves the relations as established in Appendices B and D. □

G.2 Perfect Special Honest-Verifier Zero-Knowledge (SHVZK)

Proof. We construct a simulator $\mathcal{S}$ that produces a transcript identically distributed to that of an honest prover, without access to the witness.

$\mathcal{S}$ takes as input the public statement $(\boldsymbol{g}, \boldsymbol{h}, g, h, C, C')$ and proceeds as follows:

1. Sample challenges $x, y, z, x' \xleftarrow{\$} \mathbb{Z}_p^*$.
2. Compute public parameters $\boldsymbol{h}'_i = (\boldsymbol{h}_i)^{y^{-i+1}}$, $\boldsymbol{r}_c = \langle \boldsymbol{z}, \boldsymbol{W}_V \rangle$.
3. Sample final responses uniformly:
$$\mu_c, \tau_c, \rho_c, \mu, \tau_x \xleftarrow{\$} \mathbb{Z}_p, \quad \boldsymbol{l}, \boldsymbol{r}, \boldsymbol{l}_c \xleftarrow{\$} \mathbb{Z}_p^{n_{\mathsf{padded}}}$$
4. Compute derived scalars: $\hat{t} = \langle \boldsymbol{l}, \boldsymbol{r} \rangle$, $\hat{t}_c = \langle \boldsymbol{l}_c, \boldsymbol{r}_c \rangle$.
5. Simulate commitments to satisfy verification equations:
   - Pick $A_I, A_O, T_i^* \xleftarrow{\$} \mathbb{G}$.
   - Solve for S such that the IPA verification holds:
$$S = \Big(h^{-\mu} \cdot \boldsymbol{g}^{-\boldsymbol{l}} \cdot \boldsymbol{h}'^{-\boldsymbol{r}} \cdot A_I^x \cdot A_O^{x^2} \cdot \boldsymbol{h}'^{-\boldsymbol{y}^n} \cdot W_L^x \cdot W_R^x \cdot W_O \Big)^{-x^{-3}}$$
   - Solve for S', S'_1, S'_2 using V, C_1, C_2 and responses:
$$S' = (\boldsymbol{g}^{\boldsymbol{l}_c} \cdot h^{\mu_c} \cdot V^{-1})^{x'^{-1}}$$
$$S'_1 = (\boldsymbol{c}_1'^{\boldsymbol{l}_c} \cdot g^{\rho_c} \cdot C_1^{-1})^{x'^{-1}}$$
$$S'_2 = (\boldsymbol{c}_2'^{\boldsymbol{l}_c} \cdot f^{\rho_c} \cdot C_2^{-1})^{x'^{-1}}$$
   - Solve for T_2 such that the polynomial check holds:
$$T_2 = \Big(g^{-\hat{t}} h^{-\tau_x} \cdot \big(-g^{x^2 \cdot (\delta(y,z) + \langle \boldsymbol{z}, \boldsymbol{c} \rangle)} \cdot T_1^{*x} \cdot \prod_{i=3}^{6} T_i^{*x^i} \big) \Big)^{-x^{-2}}$$
   - Solve for T'_1 such that the T_2 consistency check holds:
$$T'_1 = (g^{-\hat{t}_c} h^{-\tau_c} \cdot T_2)^{-x'^{-1}}$$
6. Invoke $\mathcal{S}^{\mathsf{IPA}_{k,m}}$ and $\mathcal{S}^{\mathsf{eCP}_{k,m}}$ to generate IPA and eCP transcripts respectively, knowing its correct witness.

Analysis. In the real protocol, responses are masked by uniformly random blinding factors ($\boldsymbol{s}_L, \boldsymbol{s}_R, \alpha, \beta$, etc.). The simulator samples these responses uniformly, perfectly matching the real distribution. Commitments are uniquely determined by responses and verification equations. Therefore, the simulated transcript is identically distributed to the real transcript. □

G.3 Computational Witness-Extended Emulation

Proof. We construct an extractor χ that utilizes the sub-protocol extractors $\chi^{\mathsf{IPA}_{k,m}}$ and $\chi^{\mathsf{eCP}_{k,m}}$, along with the Forking Lemma, to extract a valid witness. The extraction proceeds in two steps:

Step 1: Extraction of Consistency Arguments. In this step, χ extracts the witness $\boldsymbol{e}'$ and establishes its binding to the commitment V and the scalar t_2.

1. χ runs $\chi^{\mathsf{IPA}_{k,m}}$ to extract $\boldsymbol{l}_c^{(1)}, \boldsymbol{r}_c^{(1)} \in \mathbb{Z}_p^{n_{\text{padded}}}$ such that $\boldsymbol{g}^{\boldsymbol{l}_c^{(1)}} \cdot \boldsymbol{h}'^{\boldsymbol{r}_c^{(1)}} = P' \cdot h^{-\mu_c} \cdot \boldsymbol{h}'^{\boldsymbol{r}_c} = V \cdot S'^{x'} \cdot h^{-\mu_c} \cdot \boldsymbol{h}'^{\boldsymbol{r}_c}$ (Eq. 114 and Eq. 113), where $\boldsymbol{r}_c$ is public input.

2. χ runs $\chi^{\mathsf{eCP}_{k,m}}$ to extract $\boldsymbol{l}_c \in \mathbb{Z}_p^{n_{\text{padded}}}$ such that $\boldsymbol{g}^{\boldsymbol{l}_c} = V \cdot S'^{x'} \cdot h^{-\mu_c}$ (Eq. 23 and Eq. 113), $\boldsymbol{c}_1'^{\,\boldsymbol{l}_c} = C_1 \cdot S_1'^{x'} \cdot g^{-\rho_c}$ (Eq. 20), and $\boldsymbol{c}_2'^{\,\boldsymbol{l}_c} = C_2 \cdot S_2'^{x'} \cdot h^{-\rho_c}$ (Eq. 21). Steps 1 and 2 imply $\boldsymbol{g}^{\boldsymbol{l}_c^{(1)}} \cdot \boldsymbol{h}'^{\boldsymbol{r}_c^{(1)}} = \boldsymbol{g}^{\boldsymbol{l}_c} \cdot \boldsymbol{h}'^{\boldsymbol{r}_c}$. By DLR assumption, $\boldsymbol{l}_c^{(1)} = \boldsymbol{l}_c$ and $\boldsymbol{r}_c^{(1)} = \boldsymbol{r}_c$.

3. χ rewinds $\mathcal{P}^*$ to obtain two accepting transcripts with distinct challenges $x_1', x_2' \in \mathbb{Z}_p^*$. From Steps 1–2, this yields extracted vectors $\boldsymbol{l}_{c,1}, \boldsymbol{l}_{c,2} \in \mathbb{Z}_p^{n_{\text{padded}}}$. By the protocol structure, $\boldsymbol{l}_c(X) = \boldsymbol{e}' + \boldsymbol{s}_L' \cdot X$ (Eq. 96) is a degree-1 polynomial. Since $\boldsymbol{l}_{c,i} = \boldsymbol{l}_c(x_i')$, Lagrange interpolation yields:

$$\boldsymbol{s}_L' = \frac{\boldsymbol{l}_{c,1} - \boldsymbol{l}_{c,2}}{x_1' - x_2'}, \qquad \boldsymbol{e}' = \boldsymbol{l}_{c,1} - \boldsymbol{s}_L' \cdot x_1'.$$

4. Similarly, from the scalar responses $(\mu_{c,i}, \rho_{c,i})$ in both transcripts, using the polynomial structures $\mu_c(X) = s + s' \cdot X$ and $\rho_c(X) = -r' + y \cdot X$, we extract $s, s', y, r' \in \mathbb{Z}_p$:

$$s' = \frac{\mu_{c,1} - \mu_{c,2}}{x_1' - x_2'}, \quad s = \mu_{c,1} - s' \cdot x_1',$$
$$y = \frac{\rho_{c,1} - \rho_{c,2}}{x_1' - x_2'}, \quad r' = y \cdot x_1' - \rho_{c,1}.$$

5. The extractor verifies that the extracted values satisfy the commitment equations:

$$V = h^s \cdot \boldsymbol{g}^{\boldsymbol{e}'}, \qquad S' = h^{s'} \cdot \boldsymbol{g}^{\boldsymbol{s}_L'} \text{ (Eq. 93)},$$
$$S_1' = g^y \cdot \boldsymbol{c}_1'^{\,\boldsymbol{s}_L'} \text{ (Eq. 12)}, \quad S_2' = f^y \cdot \boldsymbol{c}_2'^{\,\boldsymbol{s}_L'} \text{ (Eq. 13)}.$$

   If any relation fails, comparing with the equations from Step 2 yields a non-trivial DLR in the generators, contradicting the assumption. By Appendix E, the binding property of V (with independently sampled generators $\boldsymbol{g}$) ensures that $\boldsymbol{e}'$ is the unique witness simultaneously satisfying the commitment V and the ciphertext consistency equations for C_1, C_2.

6. Using the same two transcripts, by taking the linear combinations of equation 112, χ computes $t_1', \tau_1', t_2, \tau_2 \in \mathbb{Z}_p$ such that $T_1 = g^{t_1'} h^{\tau_1'}$ (Eq. 101) and $T_2 = g^{t_2} h^{\tau_2}$ (Eq. 69). Replacing these T_1, T_2 into Eq. 112, we have $\hat{t}_c = t_2 + t_1' \cdot x'$, and $\tau_c = \tau_2 + \tau_1' \cdot x'$.

7. Step 6 implies that $\hat{t}_c = \langle \boldsymbol{l}_c, \boldsymbol{r}_c \rangle = \langle \boldsymbol{e}', \boldsymbol{r}_c \rangle + \langle \boldsymbol{s}_L', \boldsymbol{r}_c \rangle \cdot x' = t_2 + t_1' \cdot x'$. This holds for all $x' \in \{x_1', x_2'\}$. Matching coefficients for the constant term yields $t_2 = \langle \boldsymbol{e}', \boldsymbol{r}_c \rangle$. This confirms that the value t_2 (committed in T_2) is consistent with the witness $\boldsymbol{e}'$ (committed in V).

Step 2: Extraction of Circuit Witnesses. Here, χ extracts the polynomial coefficients to verify the shuffle circuit constraints.

8. χ invokes $\chi^{\mathsf{IPA}_{k,m}}$ to extract $\boldsymbol{l}, \boldsymbol{r} \in \mathbb{Z}_p^{n_{\text{padded}}}$ such that $\boldsymbol{g}^{\boldsymbol{l}} \cdot \boldsymbol{h}'^{\boldsymbol{r}} = P \cdot h^{-\mu}$ (Eq. 87), where P also satisfies Eq. 85. To isolate the polynomial coefficients, χ rewinds $\mathcal{P}^*$ to obtain 3 transcripts with distinct challenges $x \in \{x_1, x_2, x_3\}$ (fixing y, z). Let $\boldsymbol{V} \in \mathbb{Z}_p^{3\times 3}$ be the Vandermonde matrix where $V_{ij} = x_i^j$. We compute the coefficients $\boldsymbol{\nu} \in \mathbb{Z}_p^3$ to invert the system for specific powers of x. Specifically, to extract the constant term (degree 0) of the witness polynomials, we find $\boldsymbol{\nu}$ such that $\sum \nu_i x_i = 1$ and $\sum \nu_i x_i^2 = \sum \nu_i x_i^3 = 0$. Applying this linear combination to the extracted scalar μ and vectors $\boldsymbol{l}, \boldsymbol{r}$, we derive $\alpha \in \mathbb{Z}_p, \boldsymbol{a}_L, \boldsymbol{a}_R \in \mathbb{Z}_p^{n_{\text{padded}}}$ such that $A_I = h^\alpha \boldsymbol{g}^{\boldsymbol{a}_L} \boldsymbol{h}^{\boldsymbol{a}_R}$ (Eq. 46). Similarly, by targeting the coefficients for x^2 and x^3, χ extracts unique $\beta, \rho \in \mathbb{Z}_p, \boldsymbol{a}_O, \boldsymbol{s}_L, \boldsymbol{s}_R \in \mathbb{Z}_p^{n_{\text{padded}}}$ such that $A_O = h^\alpha \boldsymbol{g}^{\boldsymbol{a}_O}$ (Eq. 47) and $S = h^\alpha \boldsymbol{g}^{\boldsymbol{s}_L} \boldsymbol{h}^{\boldsymbol{s}_R}$ (Eq. 49).

9. Using equations 85 and 87, we can replace the coefficients extracted in Step 8 with the computed representations and verify the structure of the witnesses across all transcripts. We postulate that for all challenges x, y, z, the extracted witnesses satisfy:

   We then find that for all challenges x, y, x:

$$\begin{aligned}
\boldsymbol{l} &= \boldsymbol{a}_L \cdot x + \boldsymbol{a}_O \cdot x^2 + \boldsymbol{y}^{-n} \circ (z\boldsymbol{W}_R) \cdot x + \boldsymbol{s}_L \cdot x^3 \\
\boldsymbol{r} &= \boldsymbol{y}^n \circ \boldsymbol{a}_R \cdot x - \boldsymbol{y}^n + \boldsymbol{z} \cdot (\boldsymbol{W}_L \cdot x + \boldsymbol{W}_O) + \boldsymbol{y}^n \circ \boldsymbol{s}_R \cdot x^3 \\
\hat{t} &= \langle \boldsymbol{l}, \boldsymbol{r} \rangle \\
\mu &= \alpha \cdot x + \beta \cdot x^2 + \rho \cdot x^3
\end{aligned}$$

   If these equalities fail to hold for any transcript generated by $\mathcal{P}^*$, we necessarily have a non-trivial discrete log relation between the generators in ck.

10. For fixed y, z, χ rewinds $\mathcal{P}^*$ to generate 6 transcripts with different x's. By inverting the corresponding Vandermonde matrix, from equation 86 χ extracts the coefficients $\{\tau_i\}_{i\in[6]}$ and $\{t_i^*\}_{i\in[1,3,4,5,6]}$, along with the degree-2 term openings t_2, τ_2. These values satisfy $T_i^* = g^{t_i^*} h^{\tau_i}$ (Eq. 68) and $T_2 = g^{t_2} h^{\tau_2}$ (Eq. 69). This allows us to express the scalar response as $\tau_x = \sum_{i=1}^6 \tau_i x^i$ and

$$\hat{t} = \sum_{i\in[1,3,4,5,6]} t_i^* x^i + x^2(\delta(y,z) + \langle \boldsymbol{z}, \boldsymbol{c} \rangle + t_2).$$

11. We now compare the coefficient of the x^2 term in $\hat{t}$. From the inner product construction in Step 9, this coefficient is: $\langle \boldsymbol{a}_L, \boldsymbol{a}_R \circ \boldsymbol{y}^n \rangle - \langle \boldsymbol{a}_O, \boldsymbol{y}^n \rangle + \langle \boldsymbol{z}, \boldsymbol{w} \rangle + \delta(y,z)$. However, the commitment openings in Step 10 enforce that this coefficient equals $\delta(y,z) + \langle \boldsymbol{z}, \boldsymbol{c} \rangle + t_2$. Recall from Phase 1 (Step 7) that $t_2 = \langle \boldsymbol{e}', \boldsymbol{r}_c \rangle = \langle \boldsymbol{e}', \boldsymbol{W}_V^\top \boldsymbol{z} \rangle$ (Eq. 66 and Eq. 97). We isolate the terms dependent on powers of z. Using a fix y and $Q+1$ challenges z_i we have $Q+1$ equations

$$\langle \boldsymbol{a}_L, \boldsymbol{a}_R \circ \boldsymbol{y}^n \rangle - \langle \boldsymbol{a}_O, \boldsymbol{y}^n \rangle + \langle \boldsymbol{z}_i^{Q+1}, \boldsymbol{w} - \boldsymbol{W}_V \cdot \boldsymbol{e}' - \boldsymbol{c} \rangle = 0,$$

   where $\boldsymbol{z}_i^{Q+1} = (z_i, z_i^2, \ldots, z_i^Q) \in \mathbb{Z}_p^Q$.

   Since this equality holds for $Q+1$ distinct challenges z_i, we have $\sum_{j=1}^{Q+1} d_j z_i^{j-1} = 0$ with $i \in [Q+1]$, for $d_1 = \langle \boldsymbol{a}_L, \boldsymbol{a}_R \circ \boldsymbol{y}^n \rangle - \langle \boldsymbol{a}_O, \boldsymbol{y}^n \rangle$, $d_{j+1} = \boldsymbol{w} - \boldsymbol{W}_V \cdot \boldsymbol{e}' - \boldsymbol{c}$. Now, given $k \in \{1, \ldots, Q+1\}$, let $\{\nu_i\}_{i=1}^{Q+1} \subset \mathbb{Z}_p$ such that $\forall j \in \{1, \ldots, Q+1\} \setminus \{k\}$:

$$\sum_{i=1}^{Q+1} \nu_i z_i^{k-1} = 1 \quad \text{and} \quad \sum_{i=1}^{Q+1} \nu_i z_i^{j-1} = 0.$$

In other words, $\{v_i\}_{i=1}^{Q+1}$ are chosen such that for a given $k \in \{1, \ldots, Q+1\}$:

$$\sum_{i=1}^{Q+1} z_i^{j-1} v_i = \Gamma(j,k) := \begin{cases} 1, & \text{if } j = k, \\ 0, & \text{otherwise.} \end{cases}$$

Taking the linear combination of the $Q+1$ equations weighted by v_i :

$$\begin{aligned} \sum_{i=1}^{Q+1} \left(\sum_{j=1}^{Q+1} d_j z_i^{j-1} \right) v_i &= \sum_{j=1}^{Q+1} \left(\sum_{i=1}^{Q+1} z_i^{j-1} v_i \right) d_j \\ &= \sum_{j=1}^{Q+1} \Gamma(j,k) d_j = d_k = 0. \end{aligned}$$

The vanishing coefficients $d_k = 0$ for $k \in \{1, \ldots, Q+1\}$ imply two sets of constraints:

$$\boldsymbol{a}_L \circ \boldsymbol{a}_R - \boldsymbol{a}_O = \mathbf{0} \in \mathbb{Z}_p^{n_{\text{padded}}} \tag{116}$$

$$\boldsymbol{W}_L \cdot \boldsymbol{a}_L + \boldsymbol{W}_R \cdot \boldsymbol{a}_R + \boldsymbol{W}_O \cdot \boldsymbol{a}_O = \boldsymbol{W}_V \cdot \boldsymbol{e}' + \boldsymbol{c} \in \mathbb{Z}_p^Q \tag{117}$$

The circuit encodes the product $\prod_{i=1}^{n}(e'_i - u)$ via chained multiplication gates, guaranteed by Eq. 116. Equation (117) enforces that this product equals $\prod_{i=1}^{n}(e_i - u)$ (the public output)(Eq. 7). By Schwartz-Zippel over the random challenge $u \in \mathbb{Z}_p$:

$$\Pr\left[\begin{array}{l} \displaystyle\prod_{i=1}^{n}(e'_i - u) = \prod_{i=1}^{n}(e_i - u) \\ \wedge\, \boldsymbol{e}' \neq \pi(\boldsymbol{e}) \text{ for any } \pi \in S_n \end{array} \right] \leq \frac{n}{p}.$$

Since $n \ll p$, this probability is negligible. Combined with the ciphertext consistency ($\boldsymbol{C}'_i = \boldsymbol{C}_{\pi(i)} \cdot (g,h)^{r_i}$), the extracted witness $(\boldsymbol{e}', r', s)$ satisfies $\mathcal{R}_{\mathsf{shuffle}}$.

Extraction Complexity. The extractor χ invokes $\mathcal{P}^*$ on a specific tree of challenges:

- n values of y: to extract the n Hadamard constraints $a_{L,i} \cdot a_{R,i} = a_{O,i}$ from the degree-n polynomial $\sum_i (a_{L,i} a_{R,i} - a_{O,i}) y^i$;
- $(Q+1)$ values of z: to extract the Q linear constraints;
- 6 values of x: to extract the degree-6 polynomial $t(X)$;
- 2 values of x': to extract the degree-1 consistency polynomials.

The outer protocol requires $O(n \cdot Q) = O(n^2)$ transcripts. Combined with sub-protocol extraction costs of $(2k-1)^{O(\log_k n)}$ per IPA/eCP instance (Appendices B, D), the total complexity is $O(n^{2+\log_k(2k-1)})$, which is polynomial in n. By the Forking Lemma, extraction succeeds with overwhelming probability (failing only upon finding a DLR), establishing computational witness-extended emulation. □

H Security of the Optimized Protocol

The proof above establishes security for the base shuffle protocol Π_{base} (Protocols F and 3.3). We now show that the optimized protocol Π_{opt} in Section H preserves all the security properties.

Lemma 1 (Composition). *If Π_{base} satisfies perfect completeness, perfect SHVZK, and computational WEE under the DLR assumption, then so does Π_{opt}.*

Proof. The optimizations consist of two modifications:

(i) IPA Aggregation (Eq. 35): Two independent IPA instances for $(P, \hat{t}; \boldsymbol{l}, \boldsymbol{r})$ and $(P', \hat{t}_c; \boldsymbol{l}_c, \boldsymbol{r}_c)$ are combined into a single aggregated instance via random linear combination with challenge x_{agg}.

(ii) eCP Batching (Eq. 36): The 3-row eCP instance is reduced to 2 rows by batching the ciphertext components with challenge x_{agg}.

Completeness. Each optimization preserves algebraic consistency. For (i), the aggregated witnesses $\boldsymbol{l}_{\mathsf{agg}} = \boldsymbol{l} + x_{\mathsf{agg}} \cdot \boldsymbol{l}_c$ and $\boldsymbol{r}_{\mathsf{agg}} = \boldsymbol{r} + x_{\mathsf{agg}} \cdot \boldsymbol{r}_c$ satisfy the aggregated relation by linearity. For (ii), the batched base $\boldsymbol{c}'_1 \cdot (\boldsymbol{c}'_2)^{x_{\mathsf{agg}}}$ satisfies the batched equation by the same witness $\boldsymbol{l}_c$.

SHVZK. The simulator for Π_{opt} operates identically to that for Π_{base}, with the following modifications:

- Sample $x_{\mathsf{agg}} \xleftarrow{\$} \mathbb{Z}_p^*$ as an additional challenge.
- Compute $t_{\mathsf{cross}} = \langle \boldsymbol{l}, \boldsymbol{r}_c \rangle + \langle \boldsymbol{l}_c, \boldsymbol{r} \rangle$ from the simulated responses.
- Invoke $\mathcal{S}^{\mathsf{IPA}_{k,m}}$ and $\mathcal{S}^{\mathsf{eCP}_{k,m}}$ to generate IPA and eCP transcripts respectively, knowing its correct witness $\boldsymbol{l}_{\mathsf{agg}}, \boldsymbol{r}_{\mathsf{agg}}$.

The distribution remains identical since t_{cross} is determined by the uniformly sampled $\boldsymbol{l}, \boldsymbol{r}, \boldsymbol{l}_c$, and the sub-protocol simulators produce perfect simulations.

Witness-Extended Emulation. The extractor χ_{opt} proceeds as follows:

1. Invoke $\chi^{\mathsf{IPA}_{k,m}}$ to obtain $(\boldsymbol{l}_{\mathsf{agg}}, \boldsymbol{r}_{\mathsf{agg}})$.
2. Obtain two transcripts with distinct $x_{\mathsf{agg},1}, x_{\mathsf{agg},2}$. Solve the Vandermonde system to recover:

$$\boldsymbol{l}_c = \frac{\boldsymbol{l}_{\mathsf{agg},1} - \boldsymbol{l}_{\mathsf{agg},2}}{x_{\mathsf{agg},1} - x_{\mathsf{agg},2}}, \quad \boldsymbol{l} = \boldsymbol{l}_{\mathsf{agg},1} - x_{\mathsf{agg},1} \cdot \boldsymbol{l}_c$$

   Since $\boldsymbol{r}_c = \langle \boldsymbol{z}, \boldsymbol{W}_V \rangle$ is public, compute $\boldsymbol{r} = \boldsymbol{r}_{\mathsf{agg}} - x_{\mathsf{agg}} \cdot \boldsymbol{r}_c$ directly.
3. Invoke $\chi^{\mathsf{eCP}_{k,m}}$ to obtain $\boldsymbol{l}'_c$. As proven in Appendix E, the extracted $\boldsymbol{l}'_c$ satisfies all three original rows in Eq. 23, with negligible failure probability.
4. Following Steps 1–2 of χ_{base} (Appendix G.3), we have $\boldsymbol{l}'_c = \boldsymbol{l}_c$ by DLR.
5. With $(\boldsymbol{l}, \boldsymbol{r})$ and $(\boldsymbol{l}_c, \boldsymbol{r}_c)$ recovered, proceed with Steps 3–11 of χ_{base}: extract $\boldsymbol{e}', r', s$ via rewinding on x', verify $t_2 = \langle \boldsymbol{e}', \boldsymbol{r}_c \rangle$, extract circuit witnesses $(\boldsymbol{a}_L, \boldsymbol{a}_R, \boldsymbol{a}_O)$ via rewinding on x, y, z, and verify $\boldsymbol{e}'$ is a permutation of $\boldsymbol{e}$.

The additional rewinding for x_{agg} adds a factor of 2 to the extraction complexity, which remains polynomial.

□

I Comparison to Hoffmann et al.'s Shuffle Argument

Following Appendix C of [22], we analyze the computational cost of the prover and verifier in their shuffle protocol compared to ours. Let N denote the shuffle size (number of ciphertexts) and $n = 2N$ the QESA witness size. We adopt the notation of [22] throughout.

I.1 Exponentiation Counts

We first compare the total number of exponentiations, following the methodology of [22].

Hoffmann et al.'s construction.

1. **Initial commitments:** Computing $[c_\pi]$ and $[c_{\mathbf{y}}]$ requires 2 MSMs of width N.
2. **Permutation proof ($\mathsf{QESA}_{\mathsf{Copy}}$):**
   - Witness commitment $[c'_{\mathbf{w}}]$: 1 MSM of width $n = 2N$.
   - Transformed witness $[c''_{\mathbf{w}}]$: 1 MSM of width n.
   - Randomization $[\mathbf{g}']\mathbf{s}'$ and $[\mathbf{g}'']\mathbf{\Gamma}'^{\top}\mathbf{s}'$: 2 MSMs of width n.
   - $\mathsf{IPA}_{\mathsf{noZK}}$ on n-dimensional vectors: $\approx 6n = 12N$ exponentiations, since its folding phase is structurally identical to that of the IPA_k protocol (see Appendix J).
3. **Ciphertext consistency (LMPA):**
   - Randomization $[\mathbf{a}]$ (computing $[\mathbf{a}] = [\mathbf{A}]\mathbf{r}$ for a $2 \times (N{+}3)$ matrix): 2 MSMs of width N.
   - $\mathsf{LMPA}_{\mathsf{noZK}}$ on N-dimensional vectors: $\approx 6N$ exponentiations, since its folding phase is structurally identical to that of the eCP_k protocol (see Appendix J).
   - Batch aggregation: $(N{+}3)$ MSMs of width 3.

Total: $2N + 4(2N) + 12N + 2N + 6N + 3N = 33N$ exponentiations.

Our construction.

- 2 MSMs of width $2N$: $[A_I]$, $[S]$;
- 5 MSMs of width N: $[V]$, $[S']$, $[A_O]$, $[S'_1]$, $[S'_2]$;
- IPA_k on N-dimensional vectors: $\approx 6N$ exponentiations (Section J);
- eCP_k on N-dimensional vectors: $\approx 6N$ exponentiations (Section J);
- N MSMs of width 2 for $\mathbf{B}_{\mathrm{agg}}$.

Total: $4N + 5N + 6N + 6N + 2N = 23N$ exponentiations, a 30.3% reduction.

I.2 Pippenger Efficiency Analysis

Beyond raw exponentiation counts, practical performance depends heavily on multi-scalar multiplication (MSM) algorithms. The Dalek library employs Pippenger's algorithm for MSMs of width $w > 190$, achieving cost approximately $w/\log_2 w$ group operations. To ensure a fair comparison, we assume both protocols use the same library.

The prover's computational cost decomposes into two categories based on whether Pippenger's optimization applies:

Pippenger-effective operations (width > 190). These include:

- *Large initial commitments:* In our protocol, these are $[A_I]$, $[S]$ (width $2N$), and $[V]$, $[S']$, $[A_O]$, $[S'_1]$, $[S'_2]$ (width N). In Hoffmann et al.'s protocol, these are the initial commitments $[c_\pi]$, $[c_{\mathbf{y}}]$ (width N), the QESA commitments $[c'_{\mathbf{w}}]$, $[c''_{\mathbf{w}}]$, $[\mathbf{g}']\mathbf{s}'$, $[\mathbf{g}'']\mathbf{\Gamma}'^{\top}\mathbf{s}'$ (width $2N$), and the LMPA commitments (width N).
- *Cross-term MSMs in recursive protocols:* At each round j of IPA-like recursion on n-dimensional input, the prover computes 4 cross-term MSMs of width $n_j/2$. These benefit from Pippenger while $n_j/2 > 190$, i.e., for the first $\approx \log_2(N/190)$ rounds.

Table 4: IPA/eCP cross-term costs per round for $N = 2^{20}$.

Round j	n_j	Width $(n_j/2)$	$2n_j$	Pippenger cost
0	2^{20}	2^{19}	2,097,152	110,376
1	2^{19}	2^{18}	1,048,576	58,254
2	2^{18}	2^{17}	524,288	30,840
3	2^{17}	2^{16}	262,144	16,384
4	2^{16}	2^{15}	131,072	8,738
5	2^{15}	2^{14}	65,536	4,681
6	2^{14}	2^{13}	32,768	2,521
7	2^{13}	2^{12}	16,384	1,365
8	2^{12}	2^{11}	8,192	745
9	2^{11}	2^{10}	4,096	410
10	2^{10}	2^{9}	2,048	228
11	2^{9}	2^{8}	1,024	128
			Total (Pippenger-effective):	234,670

Non-Pippenger operations (width ≤ 190). These include:

- *Small-width commitments:* Our protocol requires N MSMs of width 2 for $\mathbf{B}_{\text{agg}}$, contributing $\sim 2N$ exponentiations. Hoffmann et al.'s protocol requires $(N{+}3)$ MSMs of width 3 in $\mathsf{LMPA}_{\mathsf{batch}}$, contributing $\sim 3N$ exponentiations.
- *Generator folding:* Both protocols compute folded generators of the form $\mathbf{g}' = \mathbf{g}_1^x \cdot \mathbf{g}_2$ at each recursive round. Since each position t involves a *different* base pair $(g_{1,t}, g_{2,t})$, these cannot be batched into a single MSM and thus do not benefit from Pippenger. The folding cost is $O(n)$ per protocol, with identical structure in both.
- *Late-round cross-terms:* This is where the MSM width drops below 190. For $N = 2^{20}$, this affects only rounds $j \geq 12$, contributing negligible cost.

Justification for Pippenger-focused comparison. We compare only Pippenger-effective operations because: First, folding costs are structurally identical. Both protocols use IPA-like recursion with the same folding structure, so these costs effectively cancel in a relative comparison. Second, small MSMs are negligible. A single width-2^{20} MSM costs ~50,000 normalized scalar-multiplication units (where one unit represents the complexity of a single 256-bit scalar multiplication). Because the cost drops exponentially as the recursion progresses, the cumulative work of all "late rounds", those where the vector width falls below the Pippenger effectiveness threshold of 190, amounts to fewer than 1,000 total units. This represents less than 0.1% of the total prover complexity $(1,000/931,213)$ (Table 5), meaning that any performance gains or losses in these final rounds are orders of magnitude smaller than the rounding error of the large-width MSMs.

Recursive protocol costs. For IPA-like recursion (IPA, LMPA, or eCP) on n-dimensional input with $k = 2$, each round j computes 4 cross-term MSMs of width $n_j/2 = n/2^{j+1}$, contributing $2n_j$ exponentiations. With Pippenger (when $n_j/2 > 190$), the cost per round is $4n_j/(2\log_2(n_j/2)) = 2n_j/\log_2(n_j/2)$.

Table 4 shows the costs for $N = 2^{20}$. Pippenger applies for rounds $j = 0$ to 11 (width ≥ 256).

For IPA on $2N$-dimensional input (as in Hoffmann's construction), the first round adds cost $2(2N)/\log_2(N) = 209{,}715$, yielding total Pippenger cost of $444{,}385$.

Table 5: Pippenger-effective costs for $N = 2^{20} \approx 10^6$.

Component	Ours	Hoffmann et al.
Non-recursive MSMs:		
Width-$2N$ MSMs	199,729	399,458
Width-N MSMs	262,144	262,144
Recursive cross-terms:		
IPA ($n = N$ or $2N$)	234,670	444,385
eCP/LMPA ($n = N$)	234,670	234,670
Total	931,213	1,340,657
Reduction	30.5%	

Table 6: Empirical shuffle proof timings for Hoffmann et al.'s implementation. $\mathcal{P}$ and $\mathcal{V}$ denote the prover and verifier execution times in seconds.

n	$\mathcal{P}$ [s]	$\mathcal{V}$ [s]
$1,000$	5.52	2.74
$10,000$	70.76	34.02
$100,000$	626.0	305.0

Total Pippenger-effective costs. Table 5 summarizes the Pippenger-effective costs for $N = 2^{20}$.

Our construction achieves a 30.5% reduction in Pippenger-effective group operations compared to Hoffmann et al., consistent with the 30.3% reduction in raw exponentiation counts. The improvement stems from: (1) fewer width-$2N$ MSMs (2 vs. 4 in the non-recursive phase); and (2) IPA/eCP operating on N-dimensional vectors rather than $2N$-dimensional.

I.3 Verifier's cost.

Shuffle verification in our scheme reduces to a single MSM of width $4N$ (Section 4.3.2). With Pippenger's algorithm, this corresponds to approximately $4N/\log_2(4N)$ operations.

Hoffmann et al.[22, p. 43]. report a verification cost of $10N$ exponentiations. To give a favorable interpretation, we assume that these exponentiations can be fully batched into a single MSM of width $10N$, yielding an estimated cost of $10N/\log_2(10N)$ operations.

This corresponds to a reduction by a factor of about 2.4 in this parameter regime. We emphasize that this comparison is limited to MSM complexity and assumes comparable implementation choices.

I.4 Empirical Benchmarks

To complement the analytical comparison above, we benchmark Hoffmann et al.'s implementation [23] on the same hardware as described in Section 4.1. Their implementation is written in C++ using the RELIC toolkit [3] over Curve25519. Table 6 reports the average execution times for both the prover and the verifier.

J IPA Prover Cost

We compute the prover cost for IPA-like recursive protocols: Protocol IPA_k (Section 2.5), eCP_k (Section 2.6).

Per-round cost. At round j with input size n_j and folding factor k, let $n' = n_j/k$. The prover computes:

Cross-terms. For each $\ell \in \{-(k-1), \ldots, k-1\} \setminus \{0\}$:

$$U'_\ell = \prod_{s-t=\ell} \mathbf{g}_s^{\mathbf{a}_t}, \quad U''_\ell = \prod_{s-t=\ell} \mathbf{h}_s^{\mathbf{b}_t},$$

where $s, t \in \{1, \ldots, k\}$. For fixed ℓ, there are $k - |\ell|$ pairs, each contributing an MSM of width n'. Total: $2k(k-1)n' = 2(k-1)n_j$ exponentiations.

Folding. Computing $\mathbf{g}' = \prod_{i=1}^{k} \mathbf{g}_i^{x^{k-i}}$ requires $(k-1)n'$ exponentiations per generator vector (one scalar equals 1). Total: $\frac{2(k-1)}{k} n_j$ exponentiations.

Total per round: $\frac{2(k^2-1)}{k} n_j$.

Full protocol cost. Summing over $d = \log_k n$ rounds with $\sum_{j=0}^{d-1} n_j \approx \frac{k}{k-1} n$:

$$\text{Total} = 2(k+1)n.$$

For $k = 2$: $6n$ exponentiations. For $k = 4$: $10n$ exponentiations.

Pippenger-effective portion. The cross-term MSMs (width $n_j/2k$) benefit from Pippenger when $n_j/2k > 190$. For $n = N = 2^{20}$ and $k = 2$, this holds for rounds $j = 0, \ldots, 11$.

Folding operations do not benefit from Pippenger: computing $g'_t = g_{1,t}^{x} \cdot g_{2,t}$ for each position t involves distinct base pairs, preventing batching into a single MSM.

eCP cost. The extended Chaum-Pedersen argument (Protocol 2.6) has identical structure. For γ rows, each round computes $2(k-1)\gamma$ cross-term vectors of n' elements. With $\gamma = 2$ (our batched configuration), the costs match IPA: $6n$ total exponentiations, with $\approx 234{,}670$ Pippenger-effective group operations for $n = N = 2^{20}$.

K Implementation Effort

We implement MAYA-Ristretto, built upon `curve25519-dalek`, by extending the Bulletproofs library [2] in Rust. A secondary instantiation, MAYA-P256, mirrors the same protocol logic on the Arkworks framework [4] over P-256.

Table 7 summarizes the source lines of code (SLOC) we added or modified in the primary MAYA-Ristretto implementation.

Code structure. The protocol is structured in four layers. At the lowest level, `inner_product_proof.rs` implements the k-ary folding protocols: `prove_ipa` for $\mathsf{IPA}_{k,m}$ (Protocol 2.5) and `prove_ecp` for the batched extended Chaum–Pedersen argument (Protocol 2.6). Each provides `create` (prover), `verification_scalars` (verifier), and serialization routines.

The next layer, `prover.rs` and `verifier.rs`, implements the R1CS circuit logic. The prover's `commit_vec` constructs a single aggregated vector commitment (Section 3), and `prove` implements Protocol Π_{opt} (Figure 4). The verifier's `verify` function assembles all verification equations—IPA, eCP, and circuit-consistency checks—into a single batched MSM and checks that the result is the identity element.

Table 7: Implementation effort (source lines of code, excluding comments and blank lines).

Component	Total	Verifier
Prover logic (`prover.rs`)	544	—
Verifier logic (`verifier.rs`)	401	401
Proof structure (`proof.rs`)	168	42[a]
IPA & eCP extensions (`inner_product_proof.rs`)	960	∼297[b]
Shuffle constraints (`KShuffleGadget`)	150	150
Protocol orchestration & I/O[c]	220	192
Total	**∼2,440**	**∼1,080**

[a]Deserialization logic only (`from_bytes`).
[b]Includes `verification_scalars` and `from_bytes` for `prove_ipa` and `prove_ecp`; excludes proving logic (`create`).
[c]Challenge derivation (`derive_challenge_e`), file I/O helpers, and `KShuffleGadget::{prove, verify}` orchestration.

At the top levels, the protocol orchestration layer handles the steps that precede and follow the core R1CS prover/verifier: Fiat–Shamir challenge derivation (`derive_challenge_e`), which hashes the public key, CRS identifiers, and both ciphertext lists to produce the challenge vector $\boldsymbol{e}$; the RHS aggregation (two width-n MSMs); and file I/O for reading ciphertexts and proof bundles. Both sequential and parallel (`rayon`-based) variants are provided, selected at compile time via Cargo feature flags.

Independent verification. An independent verifier requires the verification-side code from each layer. The complete procedure, as benchmarked in Table 2, proceeds as follows:

1. CRS regeneration: Regenerate the Pedersen generators and Bulletproof generator vectors. This step is independent of the ciphertexts and can be cached.
2. File read: Read the proof bundle (serialized `R1CSProof`, output commitment V, shuffled ciphertexts $\boldsymbol{C}'$) and the input ciphertexts $\boldsymbol{C}$ with the public key from disk.
3. Challenge derivation: Hash the public key, CRS identifiers, and all $4n$ ciphertext components $(\boldsymbol{c}_1, \boldsymbol{c}_2, \boldsymbol{c}'_1, \boldsymbol{c}'_2)$ into a Merlin transcript; squeeze a 32-byte seed; expand into the challenge vector $\boldsymbol{e} \in \mathbb{Z}_p^n$.
4. RHS aggregation: Compute the two width-n MSMs of Eq 6.
5. Proof verification: Initialize a fresh Merlin transcript, replay the Fiat–Shamir challenges by committing proof elements in protocol order, compute verification scalars via `verification_scalars` for both `prove_ipa` and `prove_ecp`, and execute a single batched multi-scalar multiplication combining all verification equations. Accept if and only if the result is the identity element.

The verifier performs no recursive folding and requires no witness data. Steps 1–4 use only public information; Step 5 uses only the serialized proof and public inputs.

Dependencies. The implementation depends on `curve25519-dalek` (v3.2) for elliptic curve operations and `merlin` (v2.0) for transcript management. The code is available at [5].

L Folding Factor Cost Model

We provide a cost model that explains the empirical results of Section 4.2.

The protocol executes $d = \lceil \log_k(n/m) \rceil$ recursion rounds. In each round, the prover processes vectors of size $n_j = n/k^j$ and generates $2(k-1)$ commitments. The total cost can be approximated as

$$W_{\text{total}}(k) \approx \underbrace{\sum_{j=0}^{d-1} 2(k-1) \cdot C_{\text{MSM}}\left(\frac{n}{k^j}\right)}_{\text{Arithmetic Cost}} + \underbrace{d \cdot C_{\text{overhead}}}_{\text{Structural Cost}}$$

where C_{MSM} captures the dominant multi-scalar multiplication cost and C_{overhead} models fixed per-round overheads such as memory allocation and challenge generation.

Since MSM costs are linear, the arithmetic cost term behaves as a geometric series $\sum \frac{n}{k^j} \cdot k = O(n \cdot k)$. This implies that pure arithmetic workload *increases* with k, suggesting $k = 2$ should be optimal.

However, the structural cost is proportional to the recursion depth $d = \frac{\log(n/m)}{\log k}$. As k increases from 2 to 4, the number of rounds d is halved. Our data shows that for $k = 4$, the reduction in structural overhead (saving $d/2$ rounds of expensive memory allocations) dominates the moderate increase in arithmetic density. For $k \geq 8$, the $O(nk)$ arithmetic factor grows too large, overwhelming the structural savings.

M Choice of benchmark baseline

Our evaluation uses Verificatum as the primary baseline rather than the Bayer-Groth construction [7] (as also mentioned in Section 1). The original Bayer-Groth benchmarks in [7] were conducted using ElGamal encryption over a prime-order subgroup of $\mathbb{Z}_p^*$ ($|q| = 160$ bits, $|p| = 1024$ bits), targeting an 80-bit security level. A direct comparison of these figures against a 256-bit elliptic-curve instantiation at a 128-bit security level would conflate algorithmic differences with disparities in group-operation cost. Furthermore, to the best of our knowledge, the only recent production-grade implementation of a Bayer-Groth shuffle proof is the one embedded in the Swiss Post e-voting system [26], whose integration into the broader system makes it difficult to isolate and benchmark as a standalone component. Verificatum represents a widely deployed open-source mixnet, having supported government elections in Estonia, Norway, Spain, and Israel, for instance. Its published complexity report [30] provides reproducible benchmarks on the NIST P-256 curve at $n = 10^6$. These properties make it the most appropriate baseline for a fair comparison. Accordingly, the detailed comparative analysis in Sections 4.3–4.5 focuses on $n = 10^6$ ciphertexts. We further note that the Bayer-Groth paper [7] itself reports prover performance comparable to that of Verificatum's linear-proof implementation (the Terelius-Wikström argument [27]) at the scale evaluated therein ($N = 100{,}000$).

N MAYA-P256 Prover Performance

Tables 8 and 9 report total proving times of MAYA-P256 for varying shuffle sizes in the single-threaded and multi-threaded settings, respectively.

For $n = 10^6$, the end-to-end timing of MAYA-P256 (including I/O) with 4-ary folding is 494.06 s (single-threaded) and 149.59 s (multi-threaded). For binary folding, the corresponding figures are 634.34 s and 225.86 s, respectively.

In the single-threaded setting, MAYA-P256 with 4-ary folding is approximately $1.96\times$–$2.61\times$ slower than MAYA-Ristretto across all input sizes. This gap arises from two main factors. First, MAYA-Ristretto is built on the `Curve25519-dalek` library, which benefits from curve-specific optimizations such as 51-bit limb representation on Curve25519 and a width-adaptive MSM strategy: for fewer than 190 points, Dalek bypasses the Pippenger bucket method entirely and uses variable-time Straus multiplication [2], which avoids the bucket allocation and accumulation overhead that dominates small-width MSMs. Above 190 points, Dalek switches to Pippenger with a cache-friendly window cap of $c = 8$ bits (256 buckets,

Table 8: MAYA-P256 prover performance in the single-threaded setting. Time in milliseconds for proving a shuffle of n ciphertexts.

n	Binary folding	4-ary folding
1,000	729.75	649.49
10,000	6,615.83	5,480.04
100,000	63,562.30	50,506.26
1,000,000	626,153.37	486,006.53

Table 9: MAYA-P256 prover performance in the multi-threaded setting. Time in milliseconds for proving a shuffle of n ciphertexts.

n	Binary folding	4-ary folding
1,000	309.01	243.63
10,000	2,407.37	1,679.29
100,000	22,347.30	14,983.21
1,000,000	224,198.39	147,974.10

$\sim$12 KB, fitting in L1 cache). In contrast, MAYA-P256 uses the generic Arkworks framework over P-256, which always uses a (variable-time) Pippenger-style implementation. For example, at $n = 65536$ points, Arkworks selects a window of $c = 13$ bits ($2^{13} = 8,192$ buckets, $\sim$400 KB), exceeding L1/L2 cache capacity. Second, single scalar multiplications (g^r) are $\sim$3$\times$ slower in Arkworks P-256 than in Dalek, due to the packed 64-bit limb representation required by Arkworks' generic field arithmetic.

In the multi-threaded setting, the MAYA-P256/MAYA-Ristretto ratio decreases to $1.23\times$–$1.63\times$. This is because Arkworks natively supports parallel MSM via Rayon, achieving $3.3\times$–$3.4\times$ multi- threaded speedup at large n, whereas `curve25519-dalek`'s MSM remains single-threaded even under parallel workloads (only the surrounding protocol logic is parallelized via `rayon::join`). Consequently, MAYA-P256 with multi-threaded 4-ary folding (148.0 s at $n = 10^6$) is competitive with Verificatum's multi-threaded performance (183.8 s).

The benefit of 4-ary over binary folding is more pronounced in MAYA-P256 than in MAYA-Ristretto: the 4-ary configuration is $1.21\times$–$1.29\times$ faster (single-threaded) and $1.27\times$–$1.52\times$ faster (multi-threaded), compared to $1.04\times$–$1.11\times$ and $1.07\times$–$1.10\times$ for MAYA-Ristretto, respectively. This amplification is expected: since Arkworks MSMs carry a higher per-operation cost, halving the recursion depth provides a larger absolute saving.

O MAYA-P256 Verifier Performance

Tables 10 and 11 report verification times of MAYA-P256 across varying input sizes in single-threaded and multi-threaded settings (excluding I/O), respectively.

For $n = 10^6$, the end-to-end verifier time (including I/O) for 4-ary folding is 65.17 s (single-threaded) and 25.90 s (multi-threaded). For binary folding, the corresponding figures are 64.16 s and 25.60 s, respectively.

In the single-threaded setting, MAYA-P256 verification is approximately $1.5\times$ slower than MAYA-Ristretto at $n = 10^6$ (49.4 s vs. 32.7 s for 4-ary folding). This gap is smaller than for the prover since the verifier's dominant cost is a single $4n$-width batched MSM, and the performance difference between Arkworks and Dalek narrows at large MSM widths. As discussed in Section N, at large point counts nearly

Table 10: MAYA-P256 verifier performance in the single-threaded setting (milliseconds).

n	Binary folding	4-ary folding
$1,000$	70.29	81.88
$10,000$	572.21	575.62
$100,000$	5,140.88	5,146.31
$1,000,000$	48,401.13	49,376.51

Table 11: MAYA-P256 verifier performance in the multi-threaded setting (milliseconds).

n	Binary folding	4-ary folding
$1,000$	33.47	43.21
$10,000$	264.81	276.88
$100,000$	2,208.94	2,196.55
$1,000,000$	21,262.44	21,601.33

all Pippenger buckets are occupied, eliminating Dalek's advantage from small-width optimizations (Straus algorithm, identity-skip). The remaining gap is attributable to Dalek's cache-friendly window sizing versus Arkworks' larger windows as mentioned above.

With multi-threading, MAYA-P256 achieves a $2.3\times$ verifier speedup at large n (compared to $1.9\times$ for MAYA-Ristretto), reducing the MAYA-P256/MAYA-Ristretto ratio to $1.26\times$ at $n = 10^6$. As with the prover, Arkworks benefits more from parallelism because its MSM implementation natively distributes bucket accumulation across threads, whereas Dalek's MSM remains single-threaded even in parallel mode.

P Sparse Padding Analysis

We quantify the effectiveness of sparse padding, summarized in Section 4.4, by decomposing prover workload. Let n denote the original input size and $N = n_{\text{padded}} \approx 2n$ the worst-case padded size. The structure-dependent work comprises:

- Initial commitments: $|A_I| + |A_O| + |S| \approx 5N$
- IPA_k over $(\mathbf{g}, \mathbf{h})$: $O(N)$ per round across $d = \lceil \log_k N \rceil$ rounds
- eCP_k Row 1 (generators): $O(N)$ per round across d rounds.

Thus, the dominant cost of this first category scales with N. The ciphertext-dependent work comprises:

- Initial commitments: $|V| + |S'| + |S'_1| + |S'_2| \approx 4n$
- eCP_k Row 2 (ciphertexts): $O(n)$ effective size per round.

Thus, the dominant cost of this second category scales with n.

In the worst-case padding regime $N \approx 2n$, ciphertext-dependent operations account for roughly one third of the total prover workload.

Table 12: Performance comparison of proof generation time for two studied padding strategies with $n = 2^{20} + 1 = 1,048,577$. For Global Pre-Padding, $n_{\text{padded}} = mk^d$ and "Padding" shows total dummy elements. For Iterative Padding, "Padding" shows cumulative dummy elements across all d rounds.

Configuration			Proof Size	Global Pre-Padding		Iterative Padding	
k	d	m	—	Padding	Time	Padding	Time
Early-stop regime (large m, shallow recursion)							
2	6	16,385	1.5 MB	63	262 s	6	241 s
3	4	12,946	1.2 MB	49	221 s	5	214 s
4	3	16,385	1.5 MB	63	220 s	9	212 s
5	3	8,389	0.8 MB	48	222 s	8	228 s
Balanced recursion (moderate m, medium depth)							
2	10	1,025	99 KB	1,023	266 s	10	262 s
3	6	1,439	138 KB	454	229 s	8	214 s
4	5	1,025	100 KB	1,023	219 s	15	218 s
5	4	1,678	161 KB	173	230 s	9	228 s
Deep recursion (small m, compact proofs)							
2	14	65	9.6 KB	16,383	255 s	14	252 s
3	9	54	9.3 KB	14,305	229 s	11	215 s
4	7	65	11.0 KB	16,383	219 s	21	224 s
5	6	68	12.0 KB	13,923	225 s	15	230 s
Maximal depth (minimal proofs)							
2	20	2	4.8 KB	1,048,575	**397 s**	20	241 s
3	13	1	5.8 KB	545,746	294 s	12	219 s
4	10	2	6.7 KB	1,048,575	313 s	30	**215 s**
5	9	1	7.7 KB	904,548	305 s	20	220 s

Q Padding Strategy Benchmarks

Table 12 reports proof generation times for both global pre-padding and iterative padding on a worst-case input of $n = 2^{20}+1 = 1{,}048{,}577$ ciphertexts. All experiments use sparse padding and report MAYA-Ristretto single-threaded proof generation time. Timings report only the proof generation step (as described in Section 4.3.1), where padding has the greatest impact.

Global pre-padding cliff. Global pre-padding exhibits a severe performance cliff at maximal recursion depth. In the worst case ($k = 2, d = 20$), the input is padded from $n = 1{,}048{,}577$ to $n_{\text{padded}} = 2^{21} = 2{,}097{,}152$, effectively doubling the circuit size and adding $1{,}048{,}575$ dummy elements. This increases proof generation from 241 s to 397 s—a 65% slowdown. The cliff is less extreme but still significant for $k \in \{3, 4, 5\}$, where global padding incurs 34%–46% overhead at maximal depth. For the shallow and balanced regimes, the global padding count is modest (at most $1{,}023$ for $k = 4$, $d = 5$) and the overhead is correspondingly small (0.5%–8%).

Iterative padding stability. Iterative padding eliminates the cliff entirely. The within-k range across all depths is only 5 s for $k = 3$ (214 s–219 s), 12 s for $k = 4$ (212 s–224 s), and 10 s for $k = 5$ (220 s–230 s). The largest variation occurs for $k = 2$ (21 s range), attributable to the higher total folding cost of binary recursion over 20 rounds. This stability arises because iterative padding adds at most $(k-1) \cdot d$ dummy elements—at most 30 for $k = 4$, $d = 10$—leaving the circuit size essentially unchanged from n.

Worst-case overhead. To quantify the total benefit, naïve global padding at $k = 2$, $d = 20$: 397 s with $1{,}048{,}575$ dummy elements. Iterative padding at $k = 4$, $d = 10$ achieves 215 s with 30 dummy elements—46% faster and $35{,}000\times$ fewer dummies. Compared to the exact-power-of-k baseline ($n = 2^{20}$, average proof generation ≈ 215 s), the iterative approach adds almost 0% overhead, confirming it effectively neutralizes the misalignment penalty.

R Engineering Considerations

Precomputation Opportunities. We categorize prover operations into three classes:

1. Precomputable (fixed generators, independent of witness and challenges):
   - Commitments V, A_I, A_O, S, S': $\approx 7n$ group operations
   - Round 0 of the aggregated IPA (Eq. 35): $\approx 2n$ operations
   - Round 0, Row 1 of the batched eCP (Eq. 36): $\approx n$ operations

   Subtotal: $\approx 10n$ operations
2. Challenge-dependent (generator-based, but bases vary with transcript):
   - Rounds 1 to $d-1$ of the IPA: $\approx 2n \cdot (d-1)$ operations over folded generators
   - Rounds 1 to $d-1$, Row 1 of the eCP: $\approx n \cdot (d-1)$ operations

   Subtotal: $\approx 3n \cdot (d-1)$ operations
3. Ciphertext-dependent (bases are input ciphertexts):
   - Consistency commitments S'_1, S'_2: $\approx 2n$ operations
   - All rounds, Row 2 of the eCP: $\approx n \cdot d$ operations over $\mathbf{c}'_1, \mathbf{c}'_2$

   Subtotal: $\approx 2n + n \cdot d$ operations

For $n = 10^6$ and $k = 4$ (yielding $d = 10$), the precomputable fraction is $10n/49n \approx 20\%$. Fixed-base precomputation could accelerate this portion, but requires storing a lookup table per generator point and employs variable-time table lookups that introduce timing side-channel risks. We therefore recommend the constant-time baseline for deployments in environments where the prover may be subject to adversarial timing analysis.

Side-channel considerations in MSM dispatch. The outer-protocol MSMs, i.e., commitments V, A_I, A_O, S, S', S'_1, S'_2, operate on witness-dependent scalars (the permutation and blinding factors). In MAYA-Ristretto, these use Dalek's constant-time Straus algorithm, which processes scalar digits via constant-time table lookups, preventing timing side-channels from leaking the permutation. By contrast, the IPA and eCP cross-term MSMs operate on internally generated random scalars and safely use Dalek's variable-time Pippenger. MAYA-P256 uses the generic Arkworks framework over P-256, which always uses a Pippenger-style implementation.

($\mathcal{R}_{\mathsf{AC}.1}$: Circuit Satisfiability) (43)

$\mathcal{P}$ computes: (44)

$\alpha, \beta, \rho \leftarrow \mathbb{Z}_p$ (45)

$A_I = h^{\alpha} \boldsymbol{g}^{\boldsymbol{a}_L} \boldsymbol{h}^{\boldsymbol{a}_R} \in \mathbb{G}$ (46)

$A_O = h^{\beta} \boldsymbol{g}^{\boldsymbol{a}_O} \in \mathbb{G}$ (47)

$\boldsymbol{s}_L, \boldsymbol{s}_R \leftarrow \mathbb{Z}_p^n$ (48)

$S = h^{\rho} \boldsymbol{g}^{\boldsymbol{s}_L} \boldsymbol{h}^{\boldsymbol{s}_R} \in \mathbb{G}$ (49)

$\mathcal{P} \rightarrow \mathcal{V} : A_I, A_O, S$ (50)

$\mathcal{V} : y, z \xleftarrow{\$} \mathbb{Z}_p^*$ (51)

$\mathcal{V} \rightarrow \mathcal{P} : y, z$ (52)

$\mathcal{P}$ and $\mathcal{V}$ compute: (53)

$\boldsymbol{y}^n = (1, y, y^2, \ldots, y^{n-1}) \in \mathbb{Z}_p^n$ (54)

$\boldsymbol{z} = (z, z^2, \ldots, z^Q) \in \mathbb{Z}_p^Q$ (55)

$\delta(y, z) = \langle \boldsymbol{y}^n \circ (\boldsymbol{z} \cdot \boldsymbol{W}_R), \boldsymbol{z} \cdot \boldsymbol{W}_L \rangle$ (56)

$\mathcal{P}$ computes: (57)

$l(X) = \boldsymbol{a}_L \cdot X + \boldsymbol{a}_O \cdot X^2 + \boldsymbol{y}^{-n} \circ (\boldsymbol{z}\boldsymbol{W}_R) \cdot X$ (58)

$+ \boldsymbol{s}_L \cdot X^3$ (59)

$r(X) = \boldsymbol{y}^n \circ \boldsymbol{a}_R \cdot X - \boldsymbol{y}^n + \boldsymbol{z} \cdot (\boldsymbol{W}_L \cdot X + \boldsymbol{W}_O)$ (60)

$+ \boldsymbol{y}^n \circ \boldsymbol{s}_R \cdot X^3$ (61)

$$t(X) = \langle l(X), r(X) \rangle = \sum_{i=1}^{6} t_i^* \cdot X^i \quad (62)$$

$\boldsymbol{w} = \boldsymbol{W}_L \cdot \boldsymbol{a}_L + \boldsymbol{W}_R \cdot \boldsymbol{a}_R + \boldsymbol{W}_O \cdot \boldsymbol{a}_O$ (63)

$t_2^* = \langle \boldsymbol{a}_L, \boldsymbol{a}_R \circ \boldsymbol{y}^n \rangle - \langle \boldsymbol{a}_O, \boldsymbol{y}^n \rangle + \langle \boldsymbol{z}, \boldsymbol{w} \rangle + \delta(y, z)$ (64)

$// \; t_2^* = \delta(y, z) + \langle \boldsymbol{z}, \boldsymbol{c} + \boldsymbol{W}_V \cdot \boldsymbol{e}' \rangle$ (65)

$t_2 = \langle \boldsymbol{z}, \boldsymbol{W}_V \cdot \boldsymbol{e}' \rangle \qquad // t_2^* = t_2 + \delta(y, z) + \langle \boldsymbol{z}, \boldsymbol{c} \rangle$ (66)

$\tau_i \xleftarrow{\$} \mathbb{Z}_p \forall i \in [6]$ (67)

$T_i^* = g^{t_i^*} h^{\tau_i}$ with $t_i^* \xleftarrow{\$} \mathbb{Z}_p \quad \forall i \in [1, 3, 4, 5, 6]$ (68)

$T_2 = g^{t_2} h^{\tau_2}$ (69)

$\mathcal{P} \rightarrow \mathcal{V} : \{T_i^*\}_{i \in [1,3,4,5,6]}, T_2$ (70)

$\mathcal{V} : x \xleftarrow{\$} \mathbb{Z}_p^*$ (71)

$\mathcal{V} \rightarrow \mathcal{P} : x$ (72)

$\mathcal{P}$ computes: (73)

$\boldsymbol{l} = l(x) \in \mathbb{Z}_p^n$ (74)

$\boldsymbol{r} = r(x) \in \mathbb{Z}_p^n$ (75)

$\hat{t} = \langle \boldsymbol{l}, \boldsymbol{r} \rangle \in \mathbb{Z}_p$ (76)

$$\tau_x = \sum_{i=1}^{6} \tau_i \cdot x^i \quad (77)$$

$\mu = \alpha \cdot x + \beta \cdot x^2 + \rho \cdot x^3 \in \mathbb{Z}_p$ (78)

$\mathcal{P} \rightarrow \mathcal{V} : \tau_x, \mu, \hat{t}$ (79)

$\mathcal{V}$ computes and checks: (80)

$H_i' = \boldsymbol{h}_i^{y^{-i+1}} \quad \forall i \in [n]$ (81)

$W_L = \boldsymbol{h}'^{\boldsymbol{z} \cdot \boldsymbol{W}_L}$ (82)

$W_R = \boldsymbol{g}^{\boldsymbol{y}^{-n} \circ (\boldsymbol{z} \cdot \boldsymbol{W}_R)}$ (83)

$W_O = \boldsymbol{h}'^{\boldsymbol{z} \cdot \boldsymbol{W}_O}$ (84)

$P = A_I^x \cdot A_0^{x^2} \cdot \boldsymbol{h}'^{-\boldsymbol{y}^n} \cdot W_L^x \cdot W_R^x \cdot W_O \cdot S^{x^3}$ (85)

$$g^{\hat{t}} h^{\tau_x} \stackrel{?}{=} g^{x^2 \cdot (\delta(y,z) + \langle \boldsymbol{z}, \boldsymbol{c} \rangle)} \cdot T_1^{*x} \cdot T_2^{x^2} \cdot \prod_{i=3}^{6} T_i^{*x^i} \quad (86)$$

$\mathcal{P} \leftrightarrow \mathcal{V} : \mathsf{IPA}(\boldsymbol{g}, \boldsymbol{h}', Ph^{-\mu}, \hat{t}; \boldsymbol{l}, \boldsymbol{r})$ (87)

(88)

($\mathcal{R}_{\mathsf{AC}.2}$: Binding T_2 to the Vector Commitment V) (89)

$\mathcal{P}$ computes: (90)

$s' \xleftarrow{\$} \mathbb{Z}_p$ (91)

$\boldsymbol{s}_L' \xleftarrow{\$} \mathbb{Z}_p^n$ (92)

$S' = h^{s'} \cdot \boldsymbol{g}^{\boldsymbol{s}_L'}$ (93)

$\mathcal{P} \rightarrow \mathcal{V} : S'$ (94)

$\mathcal{P}$ computes: (95)

$l_c(X) = \boldsymbol{e}' + \boldsymbol{s}_L' \cdot X \in \mathbb{Z}_p^n[X]$ (96)

$\boldsymbol{r}_c = \langle \boldsymbol{z}, \boldsymbol{W}_V \rangle \qquad //$ public (97)

$t_c(X) = \langle l_c(X), r_c(X) \rangle = t_2 + t_1' \cdot X \in \mathbb{Z}_p[X]$, (98)

$// \; t_2$ committed in T_2 (99)

$\tau_1' \xleftarrow{\$} \mathbb{Z}_p$ (100)

$T_1' = g^{t_1'} h^{\tau_1'} \in \mathbb{G}$ (101)

$\mathcal{P} \rightarrow \mathcal{V} : T_1'$ (102)

$\mathcal{V} : x' \xleftarrow{\$} \mathbb{Z}_p^*$ (103)

$\mathcal{V} \rightarrow \mathcal{P} : x'$ (104)

$\mathcal{P}$ computes: (105)

$\boldsymbol{l}_c = l_c(x') \in \mathbb{Z}_p^n$ (106)

$\hat{t}_c = \langle \boldsymbol{l}_c, \boldsymbol{r}_c \rangle \in \mathbb{Z}_p$ (107)

$\tau_c = \tau_2 + \tau_1' \cdot x' \in \mathbb{Z}_p$ (108)

$\mu_c = s + s' \cdot x' \in \mathbb{Z}_p$ (109)

$\mathcal{P} \rightarrow \mathcal{V} : \hat{t}_c, \tau_c, \mu_c$ (110)

$\mathcal{V}$ computes and checks: (111)

$g^{\hat{t}_c} h^{\tau_c} \stackrel{?}{=} T_2 \cdot T_1'^{x'}$ (112)

$P' = V \cdot S'^{x'} \in \mathbb{G}$ (113)

$\mathcal{P} \leftrightarrow \mathcal{V} : \mathsf{IPA}(\boldsymbol{g}, \boldsymbol{h}', P'h^{-\mu_c} \boldsymbol{h}'^{\boldsymbol{r}_c}, \hat{t}_c; \boldsymbol{l}_c, \boldsymbol{r}_c)$ (114)

if all checks succeed: $\mathcal{V}$ accepts, else: rejects (115)

Figure 6: Protocol F: Permutation Argument (Phase 1).