Aura: private voting with reduced trust on tallying authorities

Aram Jivanyan[1,2]⋆ and Aaron Feickert[3]

[1] Firo
[2] Yerevan State University
[3] Cypher Stack

Abstract. Electronic voting has long been an area of active and challenging research. Security properties relevant to physical voting in elections with a variety of threat models and priorities are often difficult to reproduce in cryptographic systems and protocols. Even in voting systems where ballot contents are private and results are verifiable, ballot anonymity is often offloaded to requirements of trusted parties. Here we introduce Aura, an election protocol that reduces trust on tallying authorities and organizers while ensuring voter privacy. Ballots in Aura are dissociated from voter identity cryptographically and use verifiable encryption and threshold decryption to mitigate trust in tallying authorities Aura requires no trusted setups for cryptographic primitives and uses efficient proving systems to reduce computation and communication complexity. These properties make Aura a competitive candidate for use in a variety of applications where verifiable trust minimization is desirable or necessary.

1 Introduction

Electronic voting poses unique and systemic challenges in research, development, implementation, and deployment. Verifiable electronic voting, where a public election record is employed for transparency and auditability, has inherently different trust and security requirements than other legacy techniques; in this case, ballot and election properties and tabulation methods must be secured cryptographically in order to achieve the required goals of a particular application.

Requirements, risks, and threat models in elections are complex and varied. Ballot anonymity is often required and reasonably guaranteed in physical elections, where ballots contain no identifying information about the voter at the time of tallying. Avoidance of voter coercion and bribery may also be important in major elections; a voter entering a voting booth alone where photography is prohibited can prevent this in practice, but this may not be the case if the election is conducted online.

⋆ Corresponding author: `aram@firo.org`

1.1 Requirements

Properties and requirements on voting protocols have long been the subject of interesting and evolving research, but as yet there does not appear to be a universal set of guidelines by which to analyze such constructions.

Informally, we require the following properties:

- **Public parameters**: Aside from election-specific trust requirements, all cryptographic constructions must use publicly-verifiable parameters.
- **Correctness**: A voter authorized for an election can cast a ballot that is included in the election result.
- **Universal verifiability**: Any observer can verify that all valid ballots are included in the final result, and that the result correctly represents only those ballots.
- **Ballot privacy**: It is not possible for an observer to determine the choices associated with a valid ballot.
- **Voter anonymity**: It is not possible for an observer to determine the voter associated with a valid ballot, or if a particular voter voted at all.
- **Coercion resistance**: It is possible for a voter to privately cast multiple ballots that each invalidate any previous ballots.

Coercion resistance assumes the possibility that a voter may be bribed or coerced into voting a particular choice, but is outside adversarial influence at a later time prior to the election ending. It is often related to the idea of a receipt-free election, where a voter is not able to provide evidence of its vote to a third party at any time; while Aura does not have this property, we consider the listed form of coercion resistance to be useful nonetheless.

1.2 Prior work

There is a large and growing body of research over several decades relating to security models and instantiations of electronic voting protocols using a variety of cryptographic techniques, but we do not attempt to provide a comprehensive review here.

Arguably one of the most relevant comparisons to our current work is Helios, a popular deployed protocol for so-called "boardroom" elections where many risks relevant to large-scale public elections are not present. The original Helios protocol [1] relies heavily on talliers, election organizers, and a central server; ballots are publicly linked to voter identity, and talliers act as a mixnet to shuffle ballots prior to decryption. Later work proposed an informally-described protocol update to Helios [2] that replaces expensive verifiable shuffling with homomorphic ballot decryption and a set of proofs of ballot validity; however, individual ballots are still linked to voter identity. The research of [7] introduces a straightforward verifiable ElGamal threshold cryptosystem for talliers that does not require a trusted dealer, and augments Helios to include this; however, the method provided is vulnerable to key cancellation and provides no particular guarantees on key validity. Another relevant comparison is ElectionGuard, a well-specified protocol for verifiable elections [4] that includes more robust verifiable key generation and decryption. However, it does not provide any particular verifiable guarantees on ballot anonymity, relying on election administrators to assert voter eligibility and decouple voter identity from ballot data. Additionally, although it provides an option for ballot spoiling, this requires individual decryption of such ballots for verification.

More recent work supports complete voter privacy with different trust requirements, primarily using encrypted ballots and generic circuit-based proving systems, to dissociate ballots from voter identity. For example, [8] uses a zk-SNARK construction to anonymize ballots, and relies on organizer-supplied token randomizers as a form of coercion resistance; however, soundness and voter anonymity are compromised in the case of a malicious organizer producing the proving system common reference string. In Vote-SAVER [12], voter anonymity is similarly provided by a zk-SNARK construction, and coercion resistance is achieved by having untrusted third parties conduct provable re-randomization; however, this crucially relies on proving system malleability, and therefore is currently limited (to our knowledge) to proving systems where soundness depends on a trusted organizer to produce a non-malicious common reference string. More recent work like Kryvos [9] examines more complex voting methods and adds partial or full hiding of tally details, but soundness depends on trusted organizers and proofs are large.

1.3 Contribution

Aura presents a protocol combining several useful properties that improve on earlier work.

We minimize the trust on election participants, including tally authorities with the joint capability to decrypt ballot results. In Aura, all cryptographic components may be instantiated with public verifiable parameters. Keys used to authenticate ballots can be generated by voters themselves, and the key used for decrypting election results is constructed by tally authorities in a distributed and verifiable manner that does not require a trusted dealer.

Ballots are dissociated from voter identity using voter-produced proofs, and ballot validity is asserted by a combination of verifiable ElGamal encryption and a bit vector proving system. Even in the case of collusion between talliers (and organizers) to decrypt individual ballots, voter anonymity is perfectly retained; and while multiple vote attempts by a voter can be reliably detected, this process occurs after the close of the election, and allows for safer mitigation of voter coercion by permitting such a voter to invalidate a coerced ballot anonymously and without revealing its contents.

Aura uses constructions supporting efficient operations. The proving system used to assert voter anonymity supports batch verification that greatly reduces the marginal complexity of verification, and scales extremely well in proof size even with a large number of voters. Further, a single commitment proving system is used to assert that a set of vote ciphertexts are valid, both with valid ElGamal vote messages and the overall number of choices selected by a voter; this proving system also supports batch verification and scales more efficiently than previous work, while remaining flexible for single- and multi-choice election rules.

We show a comparison between Aura and other designs in Table 1. While no generic circuit-based design appears to be in common use, we reference Vote-SAVER [12] as it is well specified as an example of such a construction.

Table 1. Comparison of properties of Aura to other systems; here, a trust-free setup means participant collusion during parameter generation cannot forge ballots or break voter privacy

Protocol	Ballot privacy	Voter privacy	Trust-free setup
ElectionGuard [4]	✓	✗	✓
Helios [2]	✓	✗	✓
Vote-SAVER [12]	✓	✓	✗
Aura [this work]	✓	✓	✓

While we use well-studied techniques and provably-secure cryptographic components to build Aura, we stress that the overall protocol analysis is informal, and we defer a formal security model and protocol-level proofs to future work.

2 Cryptographic primitives

In this section, we describe the cryptographic constructions required for the Aura election protocol. Throughout these descriptions, let $\mathbb{G}$ be a prime-order group where the discrete logarithm and decisional Diffie-Hellman problems are hard, and let $\mathbb{F}$ be its scalar field.

2.1 Distributed verifiable threshold ElGamal encryption

Aura requires a distributed verifiable threshold ElGamal cryptosystem, where a cohort of designated parties is required to decrypt messages. In such a construction, we require that key generation be fully distributed with no trusted parties. Further, the validity of key generation, encryption, and decryption must be verifiable.

The construction we describe here is based on that of [7], which describes a distributed threshold design intended for use in Helios. However, that construction is vulnerable to key cancellation attacks, does not assert proper joint key representation, and uses verification keys that (if maliciously crafted) do not allow for publicly-verifiable decryption. Further, the design is generic to support arbitrary group elements as messages, which is not secure in general [5]; while its overlying protocol does not fall victim to this problem by the nature of its construction, the general design is vulnerable. We modify the design to address these shortcomings, specify abort points in the protocol, and indicate simplifications where possible.

Let $pp_{\text{enc}} = (\mathbb{G}, \mathbb{F}, G, \{H_i\}_{i=0}^{k-1}, k, t, \nu)$ be the public parameters for the construction, where $G, \{H_i\}_{i=0}^{k-1} \in \mathbb{G}$ are independent generators, $k > 0$ is the number of valid message generators, t is the threshold of keyholders required for decryption, and ν is the total number of keyholders (so $1 \leq t \leq \nu$). The algorithms we define here rely on several auxiliary proving systems; these are introduced and defined shortly, but we reference them now. We assume that pp_{enc} is available to all algorithms, which we describe now:

- $\mathsf{KeyGen}(\alpha) \mapsto (Y_\alpha, \Pi_\alpha^{\text{key}})$: The function takes as input a player index $1 \leq \alpha \leq \nu$. It does the following:
  1. Chooses a set $\{a_{\alpha,j}\}_{j=0}^{t-1} \subset \mathbb{F}$ of scalars uniformly at random, and defines the polynomial
  $$f_\alpha(x) = \sum_{j=0}^{t-1} a_{\alpha,j} x^j$$
  and vector $C_\alpha = \{C_{\alpha,j}\}_{j=0}^{t-1} = \{a_{\alpha,j} G\}_{j=0}^{t-1}$ using these values.
  2. Produces a proof of representation $\Pi_\alpha^{\text{rep}} = \mathsf{RepProve}(G, C_{\alpha,0}; a_{\alpha,0})$, and sends the tuple $(C_\alpha, \Pi_\alpha^{\text{rep}})$ to all other players.
  3. On receipt of such a tuple $(C_\beta, \Pi_\beta^{\text{rep}})$ from another player β, verifies that $\mathsf{RepVerify}(\Pi_\beta^{\text{rep}}, G, C_{\beta,0}) = 1$, and aborts otherwise.
  4. For each $1 \leq \beta \leq \nu$, computes a value $y_{\alpha,\beta} = f_\alpha(\beta)$ and sends it to player β (using a private and secure side channel).
  5. On receipt of such a value $y_{\beta,\alpha}$ from another player β, checks that
  $$\sum_{j=0}^{t-1} \alpha^j C_{\beta,j} = y_{\beta,\alpha} G$$
  and aborts otherwise.
  6. Computes its private key share
  $$y_\alpha = \sum_{\beta=1}^{\nu} y_{\beta,\alpha}$$
  and public key share $Y_\alpha = y_\alpha G$ and public group key
  $$Y = \sum_{\beta=1}^{\nu} C_{\beta,0}.$$
  7. Produces a proof of representation $\Pi_\alpha^{\text{key}} = \mathsf{RepProve}(G, Y_\alpha; y_\alpha)$.

  The function outputs $(Y_\alpha, \Pi_\alpha^{\text{key}})$.
- $\mathsf{VerifyKeyGen}(\{Y_\alpha, \Pi_\alpha^{\text{key}}\}_{\alpha=1}^{\nu}) \mapsto Y$: The function takes as input a set of key shares and proofs from a set of ν players. It does the following:
  1. For each $1 \leq \alpha \leq \nu$, checks that $\mathsf{RepVerify}(\Pi_\alpha^{\text{key}}, G, Y_\alpha) = 1$, and aborts otherwise.
  2. Sets $Y = \sum_{\alpha=1}^{\nu} Y_\alpha$.

The function outputs Y.

- $\mathsf{Encrypt}(m, i, Y) \mapsto (D, E, \Pi_{\text{enc}})$: The function takes as input a message $m \in \mathbb{F}$, a message generator index $0 \le i < k$, and a public key Y. It does the following:
  1. Chooses a nonce $r \in \mathbb{F}$ uniformly at random.
  2. Sets $D = rG$ and $E = rY + mH_i$.
  3. Produces a proof of encryption:

  $$\Pi_{\text{enc}} = \mathsf{EncValProve}(G, Y, H_i, D, E; (r, m))$$

  The function outputs (D, E, Π_{enc}).
- $\mathsf{VerifyEncrypt}(Y, i, D, E, \Pi_{\text{enc}}) \mapsto \{0, 1\}$: The function takes as input an ElGamal public key Y, message generator index $0 \le i < k$, ElGamal ciphertext (D, E), and a proof of encryption. If

  $$\mathsf{EncValVerify}(\Pi_{\text{enc}}, G, Y, H_i, D, E) = 1$$

  it outputs 1; otherwise, it outputs 0.
- $\mathsf{PartialDecrypt}(y_\alpha, D, E) \mapsto (R_\alpha, \Pi_\alpha^{\text{dec}})$: The function takes as input a private key share y_α and ElGamal ciphertext (D, E). It does the following:
  1. Computes $R_\alpha = y_\alpha D$.
  2. Produces a proof of discrete logarithm equality:

  $$\Pi_\alpha^{\text{dec}} = \mathsf{EqProve}(D, G, R_\alpha, y_\alpha G; y_\alpha)$$

  The function outputs $(R_\alpha, \Pi_\alpha^{\text{dec}})$.
- $\mathsf{VerifyDecrypt}(D, E, \{j, Y_j, R_j, \Pi_j^{\text{dec}}\}_{j=1}^{t}) \mapsto m$: The function takes as input ElGamal ciphertext (D, E), a threshold set of t player indices, corresponding public key shares, and associated partial decryption data. We note that for the sake of notation convenience, the set of players is reindexed here; in practice, any threshold of players may be used with their corresponding indices. It does the following:
  1. For each $1 \le j \le t$, checks that $\mathsf{EqVerify}(\Pi_j^{\text{dec}}, D, G, R_j, Y_j) = 1$, and aborts otherwise.
  2. For each $1 \le j \le t$, computes the corresponding Lagrange coefficient:

  $$\lambda_j = \prod_{i=1, i \ne j}^{t} \frac{i}{i - j}$$

  3. Computes the following:

  $$M = E - \sum_{j=1}^{t} \lambda_j R_j$$

  4. Uses brute force (or another appropriate computational method) to find $m \in \mathbb{F}$ such that $mH = M$.

  The function outputs m.

2.2 Proving systems

We require several proving systems for use in Aura. Each can be instantiated non-interactively, either by instantiations cited, or using standard Schnorr-type representation proof techniques. Each is provable to be complete, special sound, and special honest-verifier zero knowledge. For each proving system, we list the public parameters, relevant relation, and prover and verifier functions; we omit the specific instantiations.

Bit vector commitment proving system This proving system asserts that a given group element is a Pedersen vector commitment to elements in the set $\{0,1\}$ whose sum is a specified value. The public parameters are $pp_{\text{bit}} = \left(\mathbb{G}, \mathbb{F}, w, k, \{G_i\}_{i=0}^{k-1}, H\right)$, where $w, k > 0$ and $\{G_i\}_{i=0}^{k-1}, H \in \mathbb{G}$ are independent generators. The relation is the following:

$$\mathcal{R}_{\text{bit}} = \left\{ pp_{\text{bit}}, B \in \mathbb{G}; \{b_i\}_{i=0}^{k-1}, r \in \mathbb{F} : B = rH + \sum_{i=0}^{k-1} b_i G_i, \right.$$
$$\left. b_i \in \{0,1\} \forall i \in [0,k), \sum_{i=0}^{k-1} b_i = w \right\}$$

The relevant algorithms are the following:

– $\mathsf{BitProve}\left(B; \{b_i\}_{i=0}^{k-1}, r\right) \mapsto \mathit{\Pi}_{\text{bit}}$
– $\mathsf{BitVerify}\left(\mathit{\Pi}_{\text{bit}}, B\right) \mapsto \{0,1\}$

A simple generalization of an existing proving system by Bootle *et al.* may be used as an instantiation of the required proving system [6]. For completeness, we include a full description of the generalization in Appendix A.

Commitment set proving system This proving system asserts that some group element in a given set is, when offset by another group element, a Pedersen commitment to zero. The public parameteres are $pp_{\text{set}} = (\mathbb{G}, \mathbb{F}, G, H, n, m)$, where $n, m > 1$ and G, H are independent generators. Let $N = n^m$. The relation is the following:

$$\mathcal{R}_{\text{set}} = \left\{ pp_{\text{set}}, \{C_i\}_{i=0}^{N-1}, C' \in \mathbb{G}; l \in [0,N), r \in \mathbb{F} : C_l - C' = rH \right\}$$

The relevant algorithms are the following:

– $\mathsf{SetProve}\left(\{C_i\}_{i=0}^{N-1}, C'; l, r\right) \mapsto \mathit{\Pi}_{\text{set}}$
– $\mathsf{SetVerify}\left(\mathit{\Pi}_{\text{set}}, \{C_i\}_{i=0}^{N-1}, C'\right) \mapsto \{0,1\}$

The proving system in [6], with a simple modification as done in [10], may be used for this purpose.

Representation proving system This proving system asserts knowledge of a group element representation. The public parameters are $pp_{\text{rep}} = (\mathbb{G}, \mathbb{F})$. The relation is the following:

$$\mathcal{R}_{\text{rep}} = \{pp_{\text{rep}}, \{G_i\}_{i=0}^{n-1}, Y; \{y_i\}_{i=0}^{n-1} : Y = \sum_{i=0}^{n-1} y_i G_i\}$$

The relevant algorithms are the following:

– $\mathsf{RepProve}(\{G_i\}_{i=0}^{n-1}, Y; \{y_i\}_{i=0}^{n-1}) \mapsto \mathit{\Pi}_{\text{rep}}$
– $\mathsf{RepVerify}(\mathit{\Pi}_{\text{rep}}, \{G_i\}_{i=0}^{n-1}, Y) \mapsto \{0, 1\}$

Encryption validity proving system This proving system asserts a valid ElGamal encryption using a specific representation assertion. The public parameters are $pp_{\text{val}} = (\mathbb{G}, \mathbb{F})$. The relation is the following:

$$\mathcal{R}_{\text{val}} = \{pp_{\text{enc}}, G, Y, H, D, E; (r, m) : D = rG, E = mY + rH\}$$

The relevant algorithms are the following:

– $\mathsf{EncValProve}(G, Y, H, D, E; r, m) \mapsto \mathit{\Pi}_{\text{enc}}$
– $\mathsf{EncValVerify}(\mathit{\Pi}_{\text{enc}}, G, Y, H, D, E) \mapsto \{0, 1\}$

Serial validity proving system This proving system asserts a valid ElGamal encryption using a specific representation assertion matches a particular partial commitment opening. The public parameters are $pp_{\text{ser}} = (\mathbb{G}, \mathbb{F})$. The relation is the following:

$$\mathcal{R}_{\text{ser}} = \{pp_{\text{ser}}, F, G, H, Y, C, D, E; (s, r, r') : \\ C = sG + rH, D = r'G, E = sF + r'Y\}$$

The relevant algorithms are the following:

– $\mathsf{SerValProve}(F, G, H, Y, C, D, E; s, r, r') \mapsto \mathit{\Pi}_{\text{ser}}$
– $\mathsf{SerValVerify}(\mathit{\Pi}_{\text{ser}}, F, G, H, Y, C, D, E) \mapsto \{0, 1\}$

Discrete logarithm equality proving system This proving system asserts two group elements share the same discrete logarithm with respect to specified generators. The public parameters are $pp_{\text{eq}} = (\mathbb{G}, \mathbb{F})$. The relation is the following:

$$\mathcal{R}_{\text{eq}} = \{pp_{\text{eq}}, G, H, Y, Y'; y : Y = yG, Y' = yH\}$$

The relevant algorithms are the following:

– $\mathsf{EqProve}(G, H, Y, Y'; y) \mapsto \mathit{\Pi}_{\text{eq}}$
– $\mathsf{EqVerify}(\mathit{\Pi}_{\text{eq}}, G, H, Y, Y') \mapsto \{0, 1\}$

2.3 Unforgeable signature scheme

In Aura, different types of participants submit messages to a public bulletin board. For some of the messages, observers must verify their authenticity in order to assert they are created by the claimed entity. For other messages, this property is not required (or even desired). We therefore assume the existence of an unforgeable signature scheme on arbitrary messages that can be bound to contexts to mitigate replay attacks. Constructions like context-prefixed Schnorr digital signatures may be used for this purpose. In the protocol, we describe which entities are assumed to possess signing and verification keys for this signature scheme.

3 Protocol

3.1 Overview

There are several types of entities in Aura that interact during the election process. Organizers set protocol parameters for elections, voters, and talliers. Voters cast ballots in elections. Talliers collaboratively compute and publish results at the end of elections. Verifiers assert that elections and results are valid.

We assume a public bulletin board $\mathcal{B}$ is used to store election data. The instantiation of $\mathcal{B}$ is especially suited for a blockchain-type construction for which modification or erasure of posted data is computationally infeasible.

3.2 Algorithms

An election consists of several steps, represented by algorithms that we describe in detail here. We assume that the organizer, the talliers, and all voters possess signing keys (with corresponding verification keys) for the unforgeable signature scheme, which can be used to sign and verify arbitrary messages to authenticate them. The distribution of such keys is outside the scope of this protocol.

SetupElection The organizer does the following:

1. Chooses a unique election identifier $\mathbb{I} \in \{0,1\}^*$, and prepares parameter m_{elec} as a human-readable description of the election, which may include auxiliary information for voters as necessary by election rules.
2. Selects parameter $k > 0$ as the number of candidates or choices in the election and, for each $i \in [0, k)$, produces a pair (i, m_i), where m_i is a human-readable description of choice i.
3. Selects parameters $k_{\min}$ and $k_{\max}$ corresponding (respectively) to the minimum and maximum number of choices a voter may make; we require that $1 \le k_{\min} \le k_{\max} \le k$. For convenience, let $k' = k + k_{\max} - k_{\min}$.
4. For each $i \in [0, k')$, samples a generator $H_i \in \mathbb{G}$ uniformly at random in a publicly-verifiable way.

5. Samples group generators $F, G, H \in \mathbb{G}$ uniformly at random in a publicly-verifiable way.
6. Prepares a list L_{voters} of the N_{voters} voter verification keys corresponding to authorized voters in the election, and lets $n, m > 1$ such that $N_{\text{voters}} = n^m$.
7. Prepares a list L_{tally} of the $N_{\text{tally}} > 0$ tallier verification keys corresponding to the authorized talliers in the election, and a threshold $1 \leq t \leq N_{\text{tally}}$ of talliers required for result decryption.
8. Prepares the public parameters for required underlying cryptographic constructions:
   - Samples a prime-order group $\mathbb{G}$ with a scalar field $\mathbb{F}$.
   - Sets $pp_{\text{enc}} = (\mathbb{G}, \mathbb{F}, \{H_i\}_{i=0}^{k'-1}, k', t, N_{\text{tally}})$ as the parameters for a distributed verifiable ElGamal encryption system.
   - Sets $pp_{\text{bit}} = (\mathbb{G}, \mathbb{F}, k_{\max}, k', \{H_i\}_{i=0}^{k'-1}, -)$ as the parameters for a bit vector commitment proving system, where we leave the final parameter undefined (to be set at a later step).
   - Sets $pp_{\text{set}} = (\mathbb{G}, \mathbb{F}, G, H, n, m)$ as the parameters for a commitment set proving system.
   - Sets $pp_{\text{rep}} = (\mathbb{G}, \mathbb{F})$ as the parameters for a representation proving system.
   - Sets $pp_{\text{val}} = (\mathbb{G}, \mathbb{F})$ as the parameters for an encryption validity proving system.
   - Sets $pp_{\text{ser}} = (\mathbb{G}, \mathbb{F})$ as the parameters for a serial validity proving system.
   - Sets $pp_{\text{eq}} = (\mathbb{G}, \mathbb{F})$ as the parameters for a discrete logarithm equality proving system.
9. Assembles the protocol public parameters

$$pp = (\mathbb{I}, m_{\text{elec}}, k, \{i, m_i\}_{i=0}^{k-1}, k_{\min}, k_{\max}, \mathbb{G}, \mathbb{F}, \{H_i\}_{i=0}^{k'-1}, F, G, H, \\ L_{\text{voters}}, N_{\text{voters}}, n, m, L_{\text{tally}}, N_{\text{tally}}, t)$$

   and posts them to $\mathcal{B}$ as an authenticated message signed with the organizer signing key.

The public parameters pp are assumed to be available to all participants and algorithms; further, all other subprotocol public parameters can be deterministically produced from pp.

SetupTally Each tallier with index $1 \leq \alpha \leq N_{\text{tally}}$ does the following:

1. Verifies the authenticated organizer message on $\mathcal{B}$ containing pp, and checks the validity of the parameters.
2. Runs $\mathsf{KeyGen}(\alpha) \mapsto (Y_\alpha, \Pi_\alpha^{\text{key}})$ interactively with the other talliers.
3. Posts the values $(\alpha, Y_\alpha, \Pi_\alpha^{\text{key}})$ to $\mathcal{B}$ as an authenticated message signed with its tallier signing key from L_{tally}.

SetupVoter Each voter with index $0 \leq i < N_{\text{voters}}$ does the following:

1. Verifies the authenticated organizer message on $\mathcal{B}$ containing pp, and checks the validity of the parameters.
2. Selects $s_i, r_i \in \mathbb{F}$ uniformly at random, and privately stores these values.
3. Computes a ballot key $C_i = s_i G + r_i H$.
4. Generates a proof of representation $\mathsf{RepProve}(\{G, H\}, C_i; \{s, r\}) \mapsto \Pi_{\text{rep},i}$.
5. Posts $(i, C_i, \Pi_{\text{rep},i})$ to $\mathcal{B}$ as an authenticated message signed with its voter signing key from L_{voters}.

We note that it is safe for a voter to reuse their ballot key across multiple elections.

VerifySetup The verifier does the following:

1. Verifies the unique authenticated organizer message on $\mathcal{B}$ containing pp, and checks the validity of the parameters.
2. For each $1 \leq \alpha \leq N_{\text{tally}}$, verifies the unique authenticated tallier message on $\mathcal{B}$ containing $(\alpha, Y_\alpha, \Pi_\alpha^{\text{key}})$ using the corresponding verification key from L_{tally}.
3. Verifies the tally keys by running $\mathsf{VerifyKeyGen}(\{Y_\alpha, \Pi_\alpha^{\text{key}}\}_{\alpha=1}^{\nu}) \mapsto Y$.
4. For each $0 \leq i < N_{\text{voters}}$, verifies the unique authenticated voter message on $\mathcal{B}$ containing $(i, C_i, \Pi_{\text{rep},i})$, and verifies the ballot key by checking that $\mathsf{RepVerify}(\Pi_{\text{rep},i}, \{G, H\}, C_i) \mapsto 1$.

At this point, all participants use Y as the undetermined parameter in pp_{bit}.

Vote Each voter with index $0 \leq i < N_{\text{voters}}$ does the following:

1. Constructs a vector $c_i = (c_{i,j})_{j=0}^{k-1}$ representing its choices among the k options, where
$$c_{i,j} = \begin{cases} 1 \text{ if the voter chooses option } j \\ 0 \text{ otherwise} \end{cases}$$
and $k_{\min} \leq \sum_{j=0}^{k-1} c_{i,j} \leq k_{\max}$.
2. For $j \in [0, k)$, encrypts each choice by setting
$$(D_{i,j}, E_{i,j}, \Pi_{\text{enc},i,j}) = \mathsf{Encrypt}(c_{i,j}, j, Y).$$
3. For $j \in [k, k')$, extends the vector c_i by setting
$$c_{i,j} = \begin{cases} 1 \text{ if } j < k + k_{\max} - \sum_{j=0}^{k-1} c_{i,j} \\ 0 \text{ otherwise} \end{cases}$$
for padding purposes, and computes encryptions
$$(D_{i,j}, E_{i,j}, \Pi_{\text{enc},i,j}) = \mathsf{Encrypt}(c_{i,j}, j, Y).$$

4. Computes a bit vector commitment proof

$$\mathit{\Pi}_{\text{bit},i} = \mathsf{BitProve}\left(\sum_{j=0}^{k'-1} E_{i,j}; \{c_{i,j}\}_{j=0}^{k'-1}, r\right),$$

where r is the sum of all nonces used in encryption proofs for $j \in [0, k'-1)$.
5. Chooses a nonce $r'_i \in \mathbb{F}$ uniformly at random, and computes the serial offset $C'_i = s_i G + r'_i H$.
6. Encrypts the ballot serial number by choosing a nonce $r''_i \in \mathbb{F}$ uniformly at random and computing $D'_i = r''_i G$ and $E'_i = s_i F + r''_i Y$.
7. Assembles $\overline{C}$ to be the set of all voter commitments $\{C_i\}$ corresponding to voter verification keys in L_{voters}, and generates a commitment set proof

$$\mathit{\Pi}_{\text{set},i} = \mathsf{SetProve}\left(\overline{C}, C'_i; l_i, r_i - r'_i\right)$$

where $\overline{C}_{l_i} = C_i$.
8. Assembles a ballot tuple:

$$B_i = \left(pp, (D_{i,j}, E_{i,j}, \mathit{\Pi}_{\text{enc},i,j})_{j=0}^{k'-1}, \mathit{\Pi}_{\text{bit},i}, C'_i, D'_i, E'_i, \mathit{\Pi}_{\text{set},i}\right)$$

9. Generates a proof of serial number validity

$$\mathit{\Pi}_{\text{ser},i} = \mathsf{SerValProve}(F, G, H, Y, C'_i, D'_i, E'_i; s_i, r'_i, r''_i)$$

that binds B_i to its initial transcript.
10. Posts the ballot tuple B_i and binding proof $\mathit{\Pi}_{\text{ser},i}$ to $\mathcal{B}$.

If the voter is coerced or bribed to submit a ballot of an adversary's choice, the voter may cast another ballot once outside of the adversary's influence by repeating these steps. As shown below, such duplicate ballots will be accepted to $\mathcal{B}$, but will be excluded from the final tally except for the last such ballot cast by the voter. This is intended to provide a weak form of coercion resistance.

VerifyBallot Given a semantically-correct ballot (without explicit voter index i) of the form

$$B = \left((D_j, E_j, \mathit{\Pi}_{\text{enc},j})_{j=0}^{k'-1}, \mathit{\Pi}_{\text{bit}}, C', D', E', \mathit{\Pi}_{\text{set}}\right),$$

any verifier does the following:

1. Checks that B does not already appear on $\mathcal{B}$, and aborts otherwise.
2. Checks that $\mathsf{SerValVerify}(\mathit{\Pi}_{\text{ser}}, F, G, H, Y, C', D', E') \mapsto 1$ using B as a transcript binding, and aborts otherwise.
3. For each $j \in [0, k')$, checks that $\mathsf{VerifyEncrypt}(Y, j, D_j, E_j, \mathit{\Pi}_{\text{enc},j}) \mapsto 1$, and aborts otherwise.

4. Checks that
$$\mathsf{BitVerify}\left(\Pi_{\text{bit}}, \sum_{j=0}^{k'-1} E_j\right) \mapsto 1,$$
and aborts otherwise.
5. Assembles the set $\overline{C}$ as in Vote, checks that $\mathsf{SetVerify}(\Pi_{\text{set}}, \overline{C}, C') \mapsto 1$, and aborts otherwise.

Rejection of duplicate ballots serves an important function. Specifically, it avoids the case where a voter submits a revised ballot in the case of coercion, and the adversary then submits the original coerced ballot to be counted instead.

While this also avoids a particular denial-of-service attack where an adversary submits "spam" copies of existing ballots to the bulletin board, it does not prevent an adversarial voter from submitting many revised ballots to the bulletin board.

It also does not address the case where ballot ordering on the bulletin board is not well defined at all times. For example, in a blockchain-type construction, it may be the case that multiple verified ballots are added to the bulletin board at the same time, such that their ordering is initially undefined. This could result in a case where a voter submits a ballot and then immediately revises it; the bulletin board ordering may not match the voter's intent. However, this is easily avoided if the time between ballot revisions exceeds the ordering time of the bulletin board.

Tally The talliers first verifiably decrypt all ballot serial numbers in order to complete the assertion of their validity and discard (for coercion-resistance purposes) recast ballots by common anonymized voters. Assume a set of t talliers indexed $1 \leq j \leq t$. Each such tallier does the following for each valid ballot i appearing on $\mathcal{B}$:

1. Runs $\mathsf{PartialDecrypt}(y_j, D'_i, E'_i) \mapsto (R_{\text{ser},i,j}, \Pi_{\text{ser},i,j})$, and posts this tuple to $\mathcal{B}$ as an authenticated message signed with its tallier signing key from L_{tally}.
2. After receiving all such partial decryptions from the threshold cohort and verifying the authenticated messages, partially (without attempting to brute-force the final decryption) runs
$$\mathsf{VerifyDecrypt}(D'_i, E'_i, \{j, Y_j, R^{\text{dec}}_{\text{ser},i,j}, \Pi_{\text{ser},i,j}\}_{j=1}^{t})$$
to obtain a serial number public key $S_i \in \mathbb{G}$.
3. Verifies the signature on the ballot i using S_i as the verification public key (against generator F).
4. If S_i appears with any other valid ballot, discard all but the most recent such ballot, according to bulletin board ordering.

At this point, let there be N_{valid} remaining valid ballots, indexed by i. The talliers now verifiably produce the tally of all N_{valid} such ballots. Each tallier with index $1 \leq j \leq t$ does the following:

1. For each $l \in [0, k)$, computes the ballot sums for choice l by setting

$$\overline{D}_l = \sum_{i=0}^{N_{\text{valid}}-1} D_{i,l}$$

   and

$$\overline{E}_l = \sum_{i=0}^{N_{\text{valid}}-1} E_{i,l},$$

   and partially decrypting the sums:

$$\mathsf{PartialDecrypt}(y_j, \overline{D}_l, \overline{E}_l) \mapsto (R_{l,j}, \mathit{\Pi}^{\text{dec}}_{l,j})$$

2. Posts the set of tuples $\{(R_{l,j}, \mathit{\Pi}^{\text{dec}}_{l,j})\}_{l=0}^{k-1}$ to $\mathcal{B}$ as an authenticated message signed with its tallier signing key from L_{tally}.

VerifyTally Any verifier checks the authenticity of all tallier messages posted from Tally and does the following:

1. For each valid ballot i appearing on $\mathcal{B}$:
   (a) Partially (without attempting to brute-force the final decryption) runs

$$\mathsf{VerifyDecrypt}(D'_i, E'_i, \{j, Y_j, R^{\text{dec}}_{\text{ser},i,j}, \mathit{\Pi}_{\text{ser},i,j}\}_{j=1}^{t})$$

   to obtain a serial number public key $S_i \in \mathbb{G}$, and aborts if this fails.
   (b) Verifies the signature on the ballot i using S_i as the verification public key (against generator F), and aborts if this fails.
   (c) If S_i appears with any other valid ballot, discard all but the most recent such ballot, according to bulletin board ordering.
2. Assembles the set of N_{valid} remaining valid ballots, now indexed by i.
3. For each choice index $l \in [0, k)$:
   (a) Computes the ballot sums for choice l by setting

$$\overline{D}_l = \sum_{i=0}^{N_{\text{valid}}-1} D_{i,l}$$

   and

$$\overline{E}_l = \sum_{i=0}^{N_{\text{valid}}-1} E_{i,l}.$$

   (b) Finalizes the decryption

$$\mathsf{VerifyDecrypt}(\overline{D}_l, \overline{E}_l, \{j, Y_j, R^{\text{dec}}_{l,j}, \mathit{\Pi}^{\text{dec}}_{l,j}\}_{j=1}^{t}) \mapsto t_l$$

   to obtain the total votes t_l for choice l, and aborts if this fails.

4 Remarks

While we do not provide a formal security analysis here, we note that Aura meets our informal requirements. All cryptographic constructions use only public parameters, and completeness properties map to overall protocol correctness. We obtain universal verifiability since any observer can run VerifyBallot on all ballots appearing on the bulletin board, and run VerifyTally to check that these ballots all appear in the correct tally. Ballot privacy follows from the properties of the cryptographic primitives used in Vote and Tally under the assumption of no malicious threshold cohort of talliers. Voter anonymity is asserted unconditionally by the use of a commitment set proof, and coercion resistance follows from the use of encrypted ballot serial numbers that are unique and fixed for each voter identity.

4.1 Efficiency

We discuss overall scaling of Aura algorithms here, and provide a more concrete comparison to ElectionGuard in Appendix B.

The most computationally-expensive construction in an Aura election is the commitment set membership proof associated to each ballot. The size of this proof scales as $O(\log(N_{\text{voters}}))$ using the instantiation referenced. While verification at first appears to scale as $O(N_{\text{voters}})$, the use of efficient multiscalar multiplication algorithms [13] can reduce this complexity to $O(N_{\text{voters}}/\log(N_{\text{voters}}))$ with good constants.

Further, the instantiation supports efficient batch verification. When verifying proofs from multiple ballots, verifier weighting of common group elements in the required multiscalar multiplication evaluation makes the marginal verification complexity constant, amortizing the overall cost across the batch. Interestingly, this use of batch verification can make overall Aura ballot verification several times more efficient than existing efficient mixnet constructions [11,3].

Other verification steps in VerifySetup, VerifyBallot, and VerifyTally imply lower complexity, or may be similarly batched for improved efficiency.

These observations make Aura a competitive candidate for suitable applications.

References

1. Adida, B.: Helios: Web-based open-audit voting. In: Proceedings of the 17th Conference on Security Symposium. p. 335–348. SS'08, USENIX Association, USA (2008)
2. Adida, B., De Marneffe, O., Pereira, O., Quisquater, J.J.: Electing a university president using open-audit voting: Analysis of real-world use of Helios. In: Proceedings of the 2009 Conference on Electronic Voting Technology/Workshop on Trustworthy Elections. p. 10. EVT/WOTE'09, USENIX Association, USA (2009)
3. Bayer, S., Groth, J.: Efficient zero-knowledge argument for correctness of a shuffle. In: Pointcheval, D., Johansson, T. (eds.) Advances in Cryptology – EUROCRYPT 2012. pp. 263–280. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)

4. Benaloh, J., Naehrig, M.: ElectionGuard specification v1.0. `https://www.electionguard.vote/spec/`
5. Boneh, D., Joux, A., Nguyen, P.Q.: Why textbook ElGamal and RSA encryption are insecure. In: Okamoto, T. (ed.) Advances in Cryptology — ASIACRYPT 2000. pp. 30–43. Springer Berlin Heidelberg, Berlin, Heidelberg (2000)
6. Bootle, J., Cerulli, A., Chaidos, P., Ghadafi, E., Groth, J., Petit, C.: Short accountable ring signatures based on DDH. In: Pernul, G., Y A Ryan, P., Weippl, E. (eds.) Computer Security – ESORICS 2015. pp. 243–265. Springer International Publishing, Cham (2015)
7. Cortier, V., Galindo, D., Glondu, S., Izabachène, M.: Distributed ElGamal à la Pedersen: Application to Helios. In: Proceedings of the 12th ACM Workshop on Workshop on Privacy in the Electronic Society. p. 131–142. WPES '13, Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2517840.2517852
8. Dimitriou, T.: Efficient, coercion-free and universally verifiable blockchain-based voting. Computer Networks **174**, 107234 (2020). https://doi.org/10.1016/j.comnet.2020.107234
9. Huber, N., Kuesters, R., Krips, T., Liedtke, J., Mueller, J., Rausch, D., Reisert, P., Vogt, A.: Kryvos: Publicly tally-hiding verifiable e-voting. Cryptology ePrint Archive, Paper 2022/1132 (2022), `https://ia.cr/2022/1132`
10. Jivanyan, A., Feickert, A.: Lelantus Spark: Secure and flexible private transactions. Cryptology ePrint Archive, Report 2021/1173 (2021), `https://ia.cr/2021/1173`
11. Krips, T., Lipmaa, H.: More efficient shuffle argument from unique factorization. In: Paterson, K.G. (ed.) Topics in Cryptology – CT-RSA 2021. pp. 252–275. Springer International Publishing, Cham (2021)
12. Lee, J., Choi, J., Kim, J., Oh, H.: SAVER: SNARK-friendly, additively-homomorphic, and verifiable encryption and decryption with rerandomization. Cryptology ePrint Archive, Report 2019/1270 (2019), `https://ia.cr/2019/1270`
13. Pippenger, N.: On the evaluation of powers and monomials. SIAM Journal on Computing **9**(2), 230–250 (1980)

A Bit commitment proving system

We now show an efficient instantiation of a bit commitment proving system. This is a generalization of the construction used in [6], which in our notation supports only $w = 1$. The proof of security follows similarly with only minor straightforward modifications, so we omit it here.

While in Aura we describe a non-interactive construction, we show here the corresponding interactive protocol, and note that the strong Fiat-Shamir technique easily applies.

1. The prover selects $r_A, r_C, r_D, \{a_i\}_{i=1}^{k-1} \in \mathbb{F}$ uniformly at random, and sets

$$a_0 = -\sum_{i=1}^{k-1} a_i.$$

2. The prover computes the Pedersen vector commitments

$$A = r_A H + \sum_{i=0}^{k-1} a_i G_i$$
$$C = r_C H + \sum_{i=0}^{k-1} a_i(1-2b_i) G_i$$
$$D = r_D H + \sum_{i=0}^{k-1} a_i^2 G_i$$

   and sends A, C, D to the verifier.
3. The verifier selects a challenge $x \in \mathbb{F} \setminus \{0\}$ uniformly at random, and sends x to the prover.
4. For each $i \in [1, k)$, the prover sets $f_i = b_i x + a_i$. The prover also sets $z_a = rx + r_A$ and $z_C = r_C x + r_D$, and sends $\{f_i\}_{i=0}^{k-1}, z_A, z_C$ to the verifier.
5. The verifier sets

$$f_0 = wx - \sum_{i=1}^{k-1} f_i$$

   and accepts the proof if and only if the following hold:

$$A + xB = z_A H + \sum_{i=0}^{k-1} f_i G_i$$
$$xC + D = z_C H + \sum_{i=0}^{k-1} f_i(x - f_i) G_i$$

B Efficiency comparison to ElectionGuard

We compare the efficiency of some Aura components to those of ElectionGuard [4], since its design is well specified. However, ElectionGuard offloads voter privacy to organizers, which Aura specifically avoids; as a result, we cannot directly compare this.

We observe that ballot validity proofs in both protocols have two overall goals: they must show that each option is a valid encryption against the correct ElGamal key, and that only a specified number of options are chosen. In Aura, we use one proof to show that an encrypted selection is a valid encryption of some message, and another to show the validity of all such messages and the correct selection limit. In ElectionGuard, one proof shows that an encrypted selection is a valid encryption of a valid message, and another asserts the correct seection limit. This difference, which arises from proving system designs, impacts efficiency.

As before, suppose an election has k options, and that a voter must select between $k_{\min}$ and $k_{\max}$ of them. Let $k' = k + k_{\max} - k_{\min}$ for convenience. We show the total size of the ballot-related proofs in Aura and ElectionGuard in Table 2, assuming for Aura a standard Schnorr-type instantiation of the required encryption validity proving system. In both cases, we generalize and assume all proof elements can be represented using a fixed and common size.[4] Further, we account for batch verification, where it is possible to present Schnorr-type proving systems for both protocols either in a manner that supports efficient verification of multiple proofs at the same time (by including the initial prover messages in the proof), or in a manner not supporting this (by including the claimed Fiat-Shamir challenge in the proof, and requiring the verifier to reconstruct the prover messages); this affects the size of each proof and the overall computational complexity.

Table 2. Total size of each ballot's validity proofs for Aura and ElectionGuard, given in proof elements, both supporting batch verification and not

Protocol	Size (batching)	Size (no batching)
Aura	$5k' + 4$	$4k' + 4$
ElectionGuard	$8k' + 3$	$4k' + 2$

We note an interesting tradeoff, in that the total size of a ballot varies between the two protocols depending on the need for batch verification support. In the case where batch verification is desired or required, Aura ballots are significantly smaller than those of ElectionGuard; if batch verification is not used, Aura ballots are slightly larger. Since Aura is particularly intended for efficient use in decentralized settings where both the size of the public bulletin board and overall verification complexity are considered limited resources, this provides a distinct and notable advantage.

[4] While Aura is presented for general groups, [4] specifies particular group parameters.