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

May 3, 2022

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. Existing work in this space often focuses on the privacy of ballot contents, assurances to voters that their votes are tabulated, and verification that election results are correct; however, privacy of voter identity is often offloaded to trust requirements on election organizers or tallying authorities, or implies other kinds of trust related to cryptographic construction instantiation. 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, use verifiable encryption and threshold decryption to diffuse trust in tallying authorities, require no trusted setup for cryptographic primitives, and use efficient proving systems to reduce computation and communication complexity. These properties make Aura a competitive candidate for use in a variety of applications where trust minimization is desirable or necessary.

1 Introduction

Electronic voting poses unique and systemic challenges in research, development, implementation, and deployment. Unfortunately, most device-based voting machine methods currently in common use are commercial products developed without public expert input, broad auditing, documented protocols, or track records of security that hold up to close examination.

*Corresponding author: aram@firo.org

Accountable online voting, where a so-called "bulletin board" of public ballots is employed for transparency and auditability, has inherently different trust and security requirements than closed machine-based 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. Voter 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 privately where photography is prohibited can prevent this in practice, but circumstances may differ significantly in the online case where ballots are publicly visible for audit purposes and a voter may cast its ballot from the device of its choosing.

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. Since many properties and threat models of physical elections are challenging to precisely map to the digital realm, it is similarly challenging to precisely specify requirements that meet all use cases that could arise.

Informally, we require the following properties:

- **Public parameters**: We require that all cryptographic constructions be instantiated with only public verificable parameters, with no trusted parties required for setup (aside from election-specific trust requirements like voter registration).
- **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.
- **Vote privacy**: It is not possible for an observer to determine the vote 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.
- **Ballot soundness**: It is not possible for a voter to cast a ballot included in the election result if:
  - The voter is not authorized to vote in the election.
  - The voter has already cast a valid ballot.
  - The vote included in the ballot is not valid according to election rules.

We mention separately an important but subtle property:

- **Coercion resistance**: It is possible for a voter to privately cast multiple ballots that each invalidate any previous ballots.

At face value, this appears to contradict ballot soundness. However, we show later that it is possible to construct algorithms that permit voters to privately cast multiple ballots that are initially judged as valid by participants during the election, but later can be verifiably identified by talliers as re-votes, such that only the most recent ballot cast by the voter is included in the result. Further, this process does not reveal the identity of the voter, so voter anonymity is maintained. It is also possible to construct algorithms that do not allow for this type of coercion-resistant re-voting, should election rules or situations prohibit it.

This property assumes the possibility that a voter may be bribed or coerced into voting a particular choice, but may be outside adversarial influence at a later time prior to the election ending. Coercion resistance 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 (a voter can store its randomness and recreate a ballot to show the third party), 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 the most relevant comparison 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 [6] 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.

More recent investigations introduce complete voter privacy with different trust requirements, primarily using combinations of encrypted ballots and general zero-knowledge proving systems, to dissociate ballots from voter identity. For example, [7] 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 [10], 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.

1.3 Contribution

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

First, 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 by voters 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 provable re-randomization and a set membership proof, 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 even 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.

Aura uses constructions supporting efficient operations. The one-of-many 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 bit vector 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.

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

2 Cryptographic primitives

In this section, we describe the cryptographic constructions required for the Aura election protocol.

2.1 Distributed verifiable threshold ElGamal encryption

Aura requires a distributed verifiable threshold ElGamal cryptosystem. Such a system requires several important properties. Unlike in some threshold cryptosystems, key generation must be fully distributed and not require a trusted dealer. Additionally, the validity of the key generation must be publicly verifiable, such that distributed knowledge of valid key shares is asserted. Finally, it must be possible to produce proofs of valid encryption and decryption of messages with public verification.

There are several algorithms used in such a construction:

- KeyGen: This algorithm is run by each keyholder to generate a key share and a proof of validity.
- VerifyKeyGen: This algorithm is run by any verifier to assert the validity of key shares and use them to assemble the corresponding group key.
- Encrypt: This algorithm is run by any entity, and encrypts a scalar-valued message for a given public ElGamal key. It also produces a proof that the encryption is valid for the public key.
- VerifyEncrypt: This algorithm is run by any verifier, and asserts that a given encryption is valid.
- PartialDecrypt: This algorithm is run by a keyholder. It produces a partial decryption of an ElGamal ciphertext message, and a proof of validity.
- VerifyDecrypt: This algorithm is run by any verifier. It uses partial decryptions to produce a plaintext message, and asserts that each partial decryption is valid.

We note that while these algorithms need not be specific to threshold operations, our construction is, and requires a given threshold of keyholders to produce a successful decryption.

The construction we describe here is based on that of [6], 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 [4]; while its overlying protocol does not fall victim to this problem by the nature of its construction, the general design is vulnerable. Fortunately, the nature of Aura ballots is such that small scalar-valued messages are required, so recovery of such messages after decryption is trivial using brute-force methods not subject to denial-of-service attacks. We therefore 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 such a cryptosystem, where $\mathbb{G}$ is a prime-order group where the discrete logarithm

problem is hard, $\mathbb{F}$ is its scalar field, $G, \{H_i\}_{i=0}^{k-1} \in \mathbb{G}$ are generators with no efficiently-computable discrete logarithm relationship, $k > 0$ 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 $1 \leq t \leq \nu$). 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 $\mathsf{RepVerify}(\Pi_\beta^{\text{rep}}, G, C_\beta) = 1$, 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.
  5. On receipt of such a value $y_{\beta,\alpha}$ from another player β, checks that

  $$\sum_{j=0}^{t-1} C_{\beta,j} = y_{\beta,\alpha} G$$

  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 $0 \leq i < k$, 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 $0 \leq i < k$, 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 $1 \leq j \leq t$, checks that $\mathsf{EqVerify}(\Pi_j^{\text{dec}}, D, G, R_j, Y_j) = 1$, and aborts otherwise.
  2. For each $1 \leq j \leq t$, computes the corresponding Lagrange coefficient:

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

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 Bit vector commitment proving system

We require a proving system that, given a group element, proves in zero knowledge that it is a Pedersen vector commitment to a "bit sequence" of field elements in the set $\{0, 1\}$ whose sum is a given value. In the context of the Aura protocol, this proving system efficiently shows that a set of ballot ciphertexts encrypt valid choices according to election rules, described later.

Let $pp_{\text{bit}} = \left(\mathbb{G}, \mathbb{F}, w, k, \{G_i\}_{i=0}^{k-1}, H\right)$ be the public parameters for such a proving system. Here $\mathbb{G}$ is a prime-order group where the discrete logarithm problem is hard, $\mathbb{F}$ is its scalar field, w and k are positive integers, and the elements $\{G_i\}_{i=0}^{k-1}, H \in \mathbb{G}$ are generators with no efficiently-computable discrete logarithm relationship. The proving system itself is a sigma protocol for the relation

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

that is complete, special honest-verifier zero knowledge, and special sound.

Any public-coin instantiation of an interactive protocol for this relation can be made non-interactive by applying the (strong) Fiat-Shamir transform. For the non-interactive protocol, define the following prover and verifier algorithms for $\mathcal{R}_{\text{bit}}$, assuming fixed parameters pp_{bit} have already been selected:

- $\mathsf{BitProve}\left(B; \{b_i\}_{i=0}^{k-1}, r\right) \mapsto \Pi_{\text{bit}}$ accepts as input statement and witness elements, and outputs a proof.
- $\mathsf{BitVerify}\left(\Pi_{\text{bit}}, B\right) \mapsto \{0, 1\}$ accepts as input a proof and statement elements, and outputs a bit to indicate whether or not the proof is valid.

We describe here a simple generalization of an existing proving system by Bootle *et al.* that originally was used to show that the bit sequence elements sum to the fixed value 1, and is an instantiation of the required proving system [5]. For completeness, we describe the full interactive protocol here.

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 $x \in \mathbb{F}$ 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 $\{f_i\}_{i=1}^{k-1}, z_A, z_C$ 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:

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

2.3 Commitment set proving system

We require a proving system that, given a set of group elements, proves in zero knowledge that one of them is a Pedersen commitment to zero. More specifically, we also include an "offset" group element that is subtracted from each element of the set first as a re-randomization of a nonzero input commitment, which is helpful for computational efficiency in practice. In the context of the Aura protocol, this proving system asserts that a ballot was produced by an eligible voter without revealing the voter's identity.[1]

[1] We stress that other forms of external information, like timing or network data, may leak information about voter identity; here we assert voter anonymity in a cryptographic context.

Let $pp_{\text{set}} = (\mathbb{G}, \mathbb{F}, G, H, n, m)$ be the public parameters for such a proving system. Here $\mathbb{G}$ is a prime-order group where the discrete logarithm problem is hard, $\mathbb{F}$ is its scalar field, $G, H \in \mathbb{G}$ are generators with no efficiently-computable discrete logarithm relationship, and $n, m > 1$ are integers. For notation convenience, let $N = n^m$. The proving system itself is a sigma protocol for the relation

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

that is complete, special honest-verifier zero knowledge, and special sound.

Any public-coin instantiation of an interactive protocol for this relation can be made non-interactive by applying the (strong) Fiat-Shamir transform. For the non-interactive protocol, define the following prover and verifier algorithms for $\mathcal{R}_{\text{set}}$, assuming fixed parameters pp_{set} have already been selected:

- $\mathsf{SetProve}\left(\{C_i\}_{i=0}^{N-1}, C'; l, r\right) \mapsto \Pi_{\text{set}}$ accepts as input statement and witness elements, and outputs a proof.
- $\mathsf{SetVerify}\left(\Pi_{\text{set}}, \{C_i\}_{i=0}^{N-1}, C'\right) \mapsto \{0, 1\}$ accepts as input a proof and statement elements, and outputs a bit to indicate whether or not the proof is valid.

The one-of-many proving system in [5], with a simple modification as done in [8], may be used for this purpose.

2.4 Other proving systems

We require several other simple proving systems relating to assertions of representation and discrete logarithm equality that are used by other cryptographic primitives in Aura. Each such proving system has a standard Schnorr-type non-interactive instantiation 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.

2.4.1 Representation proving system

This proving system asserts knowledge of a group element representation. The public parameters are $pp_{\text{rep}} = (\mathbb{G}, \mathbb{F})$, where $\mathbb{G}$ is a prime-order group where the discrete logarithm problem is hard, and $\mathbb{F}$ is its scalar field. 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:

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

2.4.2 Encryption validity proving system

This proving system asserts a valid ElGamal encryption using a specific representation assertion. The public parameters are $pp_{\text{val}} = (\mathbb{G}, \mathbb{F})$, where $\mathbb{G}$ is a prime-order group where the discrete logarithm problem is hard, and $\mathbb{F}$ is its scalar field. 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:

- $\mathsf{EncValProve}(G, Y, H, D, E; r, m) \mapsto \Pi_{\text{enc}}$
- $\mathsf{EncValVerify}(\Pi_{\text{enc}}, G, Y, H, D, E) \mapsto \{0, 1\}$

2.4.3 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 $pp_{\text{ser}} = (\mathbb{G}, \mathbb{F})$, where $\mathbb{G}$ is a prime-order group where the discrete logarithm problem is hard, and $\mathbb{F}$ is its scalar field. 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:

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

2.4.4 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 $pp_{\text{eq}} = (\mathbb{G}, \mathbb{F})$, where $\mathbb{G}$ is a prime-order group where the discrete logarithm problem is hard, and $\mathbb{F}$ is its scalar field. 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:

- $\mathsf{EqProve}(G, H, Y, Y'; y) \mapsto \Pi_{\text{eq}}$
- $\mathsf{EqVerify}(\Pi_{\text{eq}}, G, H, Y, Y') \mapsto \{0, 1\}$

2.5 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 up protocol parameters, elections, voters, and talliers. This role may be separated based on specific application needs and trust requirements.
- **Voters** cast ballots in elections.
- **Talliers** collaboratively compute and publish results at the end of elections.
- **Verifiers** assert that the setup, ballots, and tallier results are complete, accurate, and valid. Any entity or participant can act as a verifier.

We also assume a public bulletin board $\mathcal{B}$ is used to store all public data; this includes election parameters, keys, ballots, tally data, and other information. The instantiation of $\mathcal{B}$ is especially suited for a blockchain-type construction for which modification or erasure of posted data is infeasible.

An election consists of several steps, represented by algorithms that we describe in detail later.

- SetupElection: This algorithm is run by organizers and sets up $\mathcal{B}$, outputs public parameters for the election, identifies authorized voters in the election, and selects talliers.
- SetupTally: This algorithm is run by talliers and sets up the threshold keys used for result decryption.
- SetupVoter: This algorithm is run by voters and sets up the ballot keys used to cast ballots.

- VerifySetup: This algorithm can be run by any network participant to check the correctness of the setup processes.
- Vote: This algorithm is run by voters; it produces a ballot and submits it to $\mathcal{B}$.
- VerifyBallot: This algorithm is run by voters or any other network participant; it checks that a ballot is valid.
- Tally: This algorithm is run by talliers after the election concludes; it produces the results of the election.
- VerifyTally: This algorithm is run by verifiers after the results are produced; it asserts that the results represent all valid ballots correctly.

In cases where the trust model for an election differs from that implied here, the setup algorithms may differ, and yield different analysis.

3.2 Algorithms

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.

3.2.1 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 $1 \leq k_{\min} \leq k_{\max} \leq k$. 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.

3.2.2 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}.

3.2.3 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.

3.2.4 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}.

3.2.5 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

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

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

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

where $\overline{C}_{l_i} = C_i$.

8. Assembles a ballot tuple:

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

9. Generates a proof of serial number validity

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

that binds B_i to its initial transcript.

10. Posts the ballot tuple B_i and binding proof $\Pi_{\text{ser},i}$ to $\mathcal{B}$ as the voter's anonymized and authenticated ballot.

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.

3.2.6 VerifyBallot

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

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

any verifier does the following:

1. Checks that $\mathsf{SerValVerify}(\Pi_{\text{ser}}, F, G, H, Y, C', D', E') \mapsto 1$ using B as a transcript binding, and aborts otherwise.
2. For each $j \in [0, k')$, checks that $\mathsf{VerifyEncrypt}(Y, j, D_j, E_j, \Pi_{\text{enc},j}) \mapsto 1$, and aborts otherwise.
3. Checks that
$$\mathsf{BitVerify}\left(\Pi_{\text{bit}}, \sum_{j=0}^{k'-1} E_j \right) \mapsto 1,$$
and aborts otherwise.
4. Assembles the set $\overline{C}$ as in Vote, checks that $\mathsf{SetVerify}(\Pi_{\text{set}}, \overline{C}, C') \mapsto 1$, and aborts otherwise.

3.2.7 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 the tuple $(R_{\text{ser},i,j}, \Pi^{\text{dec}}_{\text{ser},i,j})$ 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:

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

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

3.2.8 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
   $$\mathsf{VerifyDecrypt}(D'_i, E'_i, \{j, Y_j, R^{\text{dec}}_{\text{ser},i,j}, \Pi_{\text{ser},i,j}\}_{j=1}^{t})$$
   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

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

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

4 Remarks

We conclude with informal observations and remarks about Aura's security and efficiency.

4.1 Security

While we do not provide a formal security analysis here, it is relevant to discuss how Aura's design works toward the properties introduced earlier.

All cryptographic components in Aura are instantiated with public parameters. While the key generation process for talliers is inherently a multiparty computation, this is a verifiable process that itself does not require specific trust for its instantiation.

Correctness follows in a straightforward manner by inspection.

Universal verifiability is achieved. Any observer can check ballot validity by running VerifyBallot on all ballots appearing on the bulletin board, and run VerifyTally to check that these ballots all appear. The final tally validity is further assured in VerifyTally from the use of the verifiable threshold decryption construction.

Vote privacy follows from the properties of the primitives used in Vote. It is possible for a threshold cohort of talliers to decrypt individual ballots, and hence we must assume no such cohort is malicious. No observer, however, can produce any such decryption or otherwise distinguish individual ballot contents due to their underlying encryption and the properties of the related Vote proofs.

The use of a commitment set proof asserts voter anonymity, which follows even if the talliers or organizer are malicious or collude.

Ballot soundness is achieved through several checks. Since the commitment set proof is sound, no unauthorized voter knows an opening to a commitment contained in such a valid set, and hence cannot cast a valid ballot. If the voter has already cast a valid ballot, any subsequent ballot must use the same serial number since the proof of serial number validity is sound and the commitments are computationally binding.

Coercion resistance, which is related to soundness, is achieved similarly.

4.2 Efficiency

Aura can operate with good efficiency.

The most computationally-expensive construction in an Aura election is the commitment set membership proof associated to each ballot, on the assumption

that the number of voters N_{voters} will exceed the number of talliers N_{tally} and choices k in the election.

The size of this proof scales as $O(\log(N_{\text{voters}}))$ using the instantiation referenced. While verification apparently scales as $O(N_{\text{voters}})$, the use of efficient multiscalar multiplication algorithms [11] can reduce this complexity to $O(N_{\text{voters}}/\log(N_{\text{voters}}))$.

Further, the instantiation supports 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 [9, 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] Ben Adida. Helios: Web-based open-audit voting. In *Proceedings of the 17th Conference on Security Symposium*, SS'08, page 335–348, USA, 2008. USENIX Association.

[2] Ben Adida, Olivier De Marneffe, Olivier Pereira, and Jean-Jacques Quisquater. 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*, EVT/WOTE'09, page 10, USA, 2009. USENIX Association.

[3] Stephanie Bayer and Jens Groth. Efficient zero-knowledge argument for correctness of a shuffle. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology – EUROCRYPT 2012*, pages 263–280, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[4] Dan Boneh, Antoine Joux, and Phong Q. Nguyen. Why textbook ElGamal and RSA encryption are insecure. In Tatsuaki Okamoto, editor, *Advances in Cryptology — ASIACRYPT 2000*, pages 30–43, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[5] Jonathan Bootle, Andrea Cerulli, Pyrros Chaidos, Essam Ghadafi, Jens Groth, and Christophe Petit. Short accountable ring signatures based on DDH. In Günther Pernul, Peter Y A Ryan, and Edgar Weippl, editors, *Computer Security – ESORICS 2015*, pages 243–265, Cham, 2015. Springer International Publishing.

[6] Véronique Cortier, David Galindo, Stéphane Glondu, and Malika Izabachène. Distributed ElGamal à la Pedersen: Application to Helios. In

Proceedings of the 12th ACM Workshop on Workshop on Privacy in the Electronic Society, WPES '13, page 131–142, New York, NY, USA, 2013. Association for Computing Machinery.

[7] Tassos Dimitriou. Efficient, coercion-free and universally verifiable blockchain-based voting. *Computer Networks*, 174:107234, 2020.

[8] Aram Jivanyan and Aaron Feickert. Lelantus Spark: Secure and flexible private transactions. Cryptology ePrint Archive, Report 2021/1173, 2021. `https://ia.cr/2021/1173`.

[9] Toomas Krips and Helger Lipmaa. More efficient shuffle argument from unique factorization. In Kenneth G. Paterson, editor, *Topics in Cryptology – CT-RSA 2021*, pages 252–275, Cham, 2021. Springer International Publishing.

[10] Jiwon Lee, Jaekyoung Choi, Jihye Kim, and Hyunok Oh. SAVER: SNARK-friendly, additively-homomorphic, and verifiable encryption and decryption with rerandomization. Cryptology ePrint Archive, Report 2019/1270, 2019. `https://ia.cr/2019/1270`.

[11] Nicholas Pippenger. On the evaluation of powers and monomials. *SIAM Journal on Computing*, 9(2):230–250, 1980.