Treasury Voting Protocol Specification

Dmytro Kaidalov
dmytro.kaidalov@iohk.io
IOHK Research

April 8, 2020

The purpose of this document is to provide a detailed specification of the voting protocol proposed by B. Zhang, R. Oliynykov, and H. Balogun in [6] and implemented in the *treasury-crypto* library [1]. Even though the treasury paper [6] does a great job at describing a general concept and details of main components of the system, it is written mostly from the academic point of view, which in some cases complicates understanding of how it can be implemented. Moreover, some details, which are important for the implementation, were put off from the paper. This specification aims to resolve the gap between the treasury paper and real implementation.

The document provides a general overview of the treasury system and then dives deeply into all parts of the voting protocol.

Before reading this document it might be useful to read [6]. The specification is compiled from the following sources:

- Full treasury paper: https://eprint.iacr.org/2018/435.pdf [6]
- Conference version of the treasury paper: https://www.ndss-symposium.org/wp-content/uploads/2019/02/ndss2019_02A-2_Zhang_paper.pdf [7]
- Implementation of the voting protocol (*treasury-crypto* library): https://github.com/input-output-hk/treasury-crypto/ [1]
- Full-fledged prototype of the treasury on top of the Scorex framework: https://github.com/input-output-hk/TreasuryCoin/ [2]
- Internal discussions and personal notes

Contents

1 Treasury System Overview

The decentralized nature of blockchain systems complicates their maintenance, further development and governance. System improvements have to be publicly proposed, approved, and funded, keeping the corresponding level of decentralization.

To that end, it is important to provide a sustainable decentralized treasury system, which is oriented towards governing funds for recurring tasks of the blockchain development, maintenance and support. Having this component is important for maintaining a decentralized system in the long-term prospective.

The basis of the treasury system is a collaborative decision-making process which can be done through the voting. A key feature expected from the voting procedure is the absence of a centralized control over the operational process. That is, it must neither rely on trusted parties or powerfull minority, nor introduce incentives to their appearance. Ideally, all cryptocurrency stake holders are entitled to participate in the decision-making process.

The basic flow of a decision making process is depicted in Fig.1. In the first stage a proposal is submitted for consideration to the community (e.g. provide xxx coins from the treasury for a specific development team to implement feature Y). The second stage is voting where corresponding participants of the system express their opinion by posting voting ballots on a blockchain. To achieve better collaborative intelligence, it is allowed to delegate the voting power to a special actor called *expert*. In the third stage the system processes ballots, counts votes and concludes a decision. In the final stage the decision is executed (e.g. the coins are transferred from treasury to the development team).

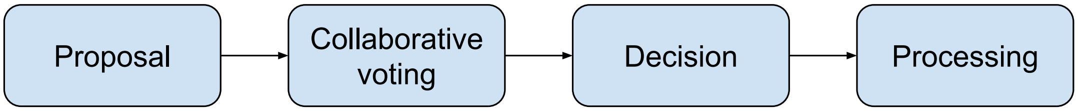

Figure 1: Basic flow of a decision-making process

This process is repeated periodically. Each such period is called a treasury epoch. In our treasury system, each epoch consists of the following stages:

1. **Pre-voting stage**.
   a) Proposals submission.
   b) Voters/Experts/Committee registration.
   c) Randomness revealing.
   d) Randomized selection of the voting comittee.
   a) Distributed voting key generation.
2. **Voting stage**.
   a) Ballots casting.
3. **Post-voting stage**.
   a) Joint decryption of tally.
   b) Randomness commitment for the next epoch.
   c) Execution.

1.1 Pre-voting Stage

Entities. All stake holders are eligible to participate in case they registered themselves. The stake holders may have one or more of the following roles.

- **Project owners** $\mathcal{O} := \{o_1, \ldots, o_k\}$, who submit proposals for funding;
- **Voting committee** $\mathcal{C} := \{c_1, \ldots, c_l\}$ - special actors that maintain a decentrilized voting procedure (e.g., generate a shared voting public key and collectively decrypt the voting result);

- **Voters** $\mathcal{V} := \{v_1, \ldots, v_n\}$ - a set of stake holders that deposited a certain amount of stake to participate in voting; the voting power is proportional to the amount of deposited stake;
- **Experts** $\mathcal{E} := \{e_1, \ldots, e_m\}$ - a special type of voters that have specialist knowledge and expertise in some field; their voting power equals to the sum of voting power of all regular voters that delegated their stake to the expert.

Note that experts and voting committee members are also required to deposit some fixed amount of stake to register themselves. But this stake does not provide them voting power, but rather serves as a deterrence against malicious behaviour. In case they do not follow the protocol, the deposit is confiscated.

The deposited stake of all entities is locked for a certain amount of time (e.g., several epochs) to incentivize prudent behaviour.

Proposal submission. In order to submit a proposal for funding, a project owner submits a special proposal transaction[1] to the blockchain:

$$Proposal_{TX} \overset{\text{def}}{=} (projectID,\ recipientAddr,\ amount),$$

where:

$projectID$ – a unique identifier of the project (e.g., its name);
$recipientAddr$ – address of the recipient, where requested funds should be sent in case of approval;
$amount$ – requested amount of funds.

Note that to prevent denial-of-service attacks it is required for the submitter to burn some amount of coins.

Voters/Experts registration. In order to become a voter or expert, a stakeholder must submit the following registration transaction[2]:

$$Reg_{TX} \overset{\text{def}}{=} (role,\ Option[committeePubKey],\ pubKey,\ depositAmount,\ paybackAddress,\ sig),$$

where:

$role$ – a role for which a stakeholder is registered (voter or expert);
$committeePubKey$ – an optional field; in case a voter/expert also wants to participate in the voting committee, he provides an additional public key that is used for committee-specific operations;
$depositAmount$ – an amount of stake that a voter wants to deposit to acquire the right to participate in the voting process; the voting power is proportional to the amount of deposited stake. In case a voter also wants to be a committee member, he deposits an additional fixed amount of stake, which does not increase his voting power. In case registering an expert, deposited stake is a fixed amount only depending on if the expert also wants to participate in the committee. Experts do not have their own voting power;
$paybackAddress$ – an address where rewards should be sent;
$pubKey$ – a personal public key that will be used for issuing ballots;
sig – a signature on the whole registration transaction issued with $pubKey$.

Randomness revealing. A random value is required for the committee selection procedure. This random value is generated collectively by the voting committee of the previous treasury epoch. At this stage, they just reveal previously committed randomness. It is crucial that the randomness is revealed after the registration phase, so that committee candidates cannot influence the selection procedure by adjusting their registration data.

[1] See the implementation of a proposal transaction here:
https://github.com/input-output-hk/TreasuryCoin/.../examples/hybrid/transaction/ProposalTransaction.scala

[2] See the implementation of a registration transaction here:
https://github.com/input-output-hk/TreasuryCoin/.../examples/hybrid/transaction/RegisterTransaction.scala

Randomized selection of the voting comittee. To facilitate efficiency of the voting protocol, a voting committee is restricted to have fixed size. Since there might be more users wanting to participate in the committee, a special random selection procedure (Fig. 2) is used to determine who will be in the committee for a particular treasury epoch[3].

Committee Selection Procedure

1. For each committee member C_i calculate the value

$$t_i = H(committeePubKey_i \mid randomness),$$

   where *randomness* is some random value derived after the registration procedure has been finished (e.g., it can be a randomness derived from a blockchain or collectively generated by committee members of the previous epoch).

2. Sort all registered committee members by their t_i values.

3. Chose top l committee members, where l is a system parameter, who constitute the voting committee for the current treasury epoch.

Figure 2: Committee Selection Procedure

Distributed key generation. During the DKG phase, the elected voting committee jointly generates a shared public voting key which will be used by voters and experts to encrypt their ballots. Then, after the voting stage is finished, the committee collectively decrypt the tally and generate randomness for the next treasury epoch.

1.2 Voting Stage

After the preparation stage there are a set of proposals $\mathcal{P} := \{P_1, \ldots, P_k\}$ and three sets of voting participants:

1. **Voters** $\mathcal{V} := \{v_1, \ldots, v_n\}$. Each voter is associated with his registered $pubKey_{v_i}$ and voting power vp_{v_i}.

2. **Experts** $\mathcal{E} := \{e_1, \ldots, e_m\}$. Each expert is associated with his registered $pubKey_{e_i}$ and his number i.

3. **Committee members** $\mathcal{C} := \{c_1, \ldots, c_l\}$. Each committee member is associated with two registered keys $pubKey_{c_i}$ and $committeePubKey_{c_i}$. The latter is used to encrypt communication with other committee members.

During the voting stage, voters and experts issue voting ballots where they put their choices regarding proposals. For each proposal, a voter may chose among three options: Yes, No, Abstain, or he can delegate his voting power to some expert, in which case the chose of the expert will be counted with the corresponding voting power of the voter. Note that each proposal is treated separately, so that a voter can delegate his voting power to different experts for different proposals.

1.3 Post-voting Stage

Joint decryption of tally. After the voting stage, all ballots are collected and the voting committee jointly decrypt the tally for each proposal without revealing personal choices of voters and experts. Winning proposals are selected according to the procedure on Fig. 3.

[3] See implementation here:
https://github.com/input-output-hk/TreasuryCoin/../examples/hybrid/state/TreasuryState.scala:selectApprovedCommittee()

Proposals selection procedure

1. Filter out all proposals for which the difference between ”Yes” and ”No” votes is less than 10% of the total voting power.
2. Sort all remaining proposals according to the amount of ”Yes” votes (taking into account the voting power of different voters).
3. Top ranked proposals are funded one-by-one until the treasury budget for the epoch is exhausted.

Figure 3: Proposals selection procedure

Randomness commitment for the next epoch. At this stage, each committee member commits a random value that will be used to construct randomness for the next treasury epoch.

Execution. During the execution stage treasury funds are distributed to winning proposals. Certain proportion (e.g. 20%) of the treasury fund is used to reward the voting committee members, voters and experts.

1.4 Incentives

There are three main factors to incentivize treasury participants:

1. Rewards.
2. Deposit lock.
3. Deposit confiscation.

Rewards. The cornestone of the incentive scheme is the rewards paid for participating in the treasury protocol. All entities are eligible for rewards according to their roles. The rewards fund is a fixed portion (e.g., 20%) of the overall treasury fund, which is divided between committe members and voters. Voting committee members receive a fixed amount of reward, while voters receive rewards proportional to their voting power. Experts receive rewards as a percentage (e.g. 5%) of rewards for voters, who delegated to them[4].

The rewards are paid only if an entity follows the protocol correctly. For voters and experts it means that they issue correctly formed voting ballots in time; for committee members it means that they follow the protocol and issue all required transactions in time.

Deposit lock. All entities are required to deposit certain amount of stake depending on their roles. This deposit is locked for a prolonged period of time (e.g., 3 treasury epochs). The basic idea behind it is to establish an incentive for voters to make sensible decisions that will benefit long-term prospects of the system. Given that their stake is locked they are incentivized to make decisions that will increase value of their stake.

Deposit confiscation. Deposit confiscation is applied to experts and committee members in case they do not behave as expected. Given that experts are responsible to make decisions for voters delegating to them, their refusal to participate in a voting (i.e., not issuing a ballot) is considered as malicious behaviour that is punished by deposit confiscation.

Committee members are responsible for managing the voting procedure. Even though the protocol is robust against failure of up to 50% of members, a committee member is punished by deposit confiscation in case of not following the protocol.

[4] See implementation of payments distribution here:
https://github.com/input-output-hk/TreasuryCoin/../examples/hybrid/state/TreasuryState.scala:getPayments()

2 Voting Protocol

The section describes in details Voting and Tally stages. We assume that after the Preparation stage the following four sets are defined:

- **Proposals** $\mathcal{P} := \{p_1, \ldots, p_k\}$.
- **Voting committee** $\mathcal{C} := \{c_1, \ldots, c_l\}$.
- **Voters** $\mathcal{V} := \{v_1, \ldots, v_n\}$.
- **Experts** $\mathcal{E} := \{e_1, \ldots, e_m\}$.

2.1 Preliminaries

2.1.1 Basic Math

The implementation[5] of our scheme is based on elliptic curve groups for efficiency. Let $\sigma := (p, a, b, g, q, \zeta)$ be the elliptic curve domain parameters over $\mathbb{F}_p$, consisting of a prime p specifying the finite field $\mathbb{F}_p$, two elements $a, b \in \mathbb{F}_p$ specifying an elliptic curve $E(\mathbb{F}_p)$ defined by $E : y^2 \equiv x^3 + ax + b \pmod{p}$, a base point $g = (x_g, y_g)$ on $E(\mathbb{F}_p)$, a prime q which is the order of g, and an integer ζ which is the cofactor $\zeta = \#E(\mathbb{F}_p)/q$. We denote the cyclic group generated by g by $\mathbb{G}$, and it is assumed that the DDH assumption holds over $\mathbb{G}$, that is for all p.p.t. adversary $\mathcal{A}$:

$$\mathsf{Adv}^{\mathsf{DDH}}_{\mathbb{G}}(\mathcal{A}) = \left| \Pr\left[\begin{array}{l} x, y \leftarrow \mathbb{Z}_q; b \leftarrow \{0,1\}; h_0 = g^{xy}; \\ h_1 \leftarrow \mathbb{G} : \mathcal{A}(g, g^x, g^y, h_b) = b \end{array} \right] - \frac{1}{2} \right| \leq \epsilon(\lambda) \ ,$$

where $\epsilon(\cdot)$ is a negligible function.

2.1.2 ElGamal Encryption

We employ lifted ElGamal encryption scheme as the candidate of the additively homomorphic public key cryptosystem in our protocol construction. It consists of the following p.p.t. algorithms:

- $\mathsf{Gen}_{\mathsf{gp}}(1^\lambda)$: take input as security parameter λ, and output $\sigma := (p, a, b, g, q, \zeta)$.
- $\mathsf{Gen}(\sigma)$: pick $\mathsf{sk} \leftarrow \mathbb{Z}_q^*$ and set $\mathsf{pk} := h = g^{sk}$, and output $(\mathsf{pk}, \mathsf{sk})$.
- $\mathsf{Enc}'_{\mathsf{pk}}(n; r)$: output $e := (e_1, e_2) = (g^r, n \cdot h^r)$ is a regular ElGamal encryption, where $n \in E(\mathbb{F}_p)$ is an EC point.
- $\mathsf{Dec}'_{\mathsf{sk}}(e)$: output $n = e_2 \cdot e_1^{-\mathsf{sk}}$ is a regular ElGamal decryption.
- $\mathsf{Enc}_{\mathsf{pk}}(m; r)$: output $e := (e_1, e_2) = (g^r, g^m h^r)$ is a lifted ElGamal encryption, where $m \in \mathbb{Z}_q^*$ is an integer.
- $\mathsf{Dec}_{\mathsf{sk}}(e)$: output $m = \mathsf{Dlog}(\mathsf{Dec}'_{sk}(e))$, where $\mathsf{Dlog}(x)$ is a discrete logarithm of x. (Note that since $\mathsf{Dlog}(\cdot)$ is not efficient, the message space should be a small set, say $m \in \{0,1\}^\xi$, for $\xi \leq 30$.)

It is well known that lifted ElGamal encryption scheme is IND-CPA secure under the DDH assumption. It has additively homomorphic property:

$$\mathsf{Enc}_{\mathsf{pk}}(m_1; r_1) \cdot \mathsf{Enc}_{\mathsf{pk}}(m_2; r_2) = \mathsf{Enc}_{\mathsf{pk}}(m_1 + m_2; r_1 + r_2) \ .$$

Remark: The key generation and decryption algorithm of the lifted ElGamal encryption can be efficiently distributed.

[5] Implementation of all basic crypto can be found here:
https://github.com/input-output-hk/treasury-crypto/../core/crypto

2.1.3 Hybrid Encryption

We employ a hybrid symmetric/assymetric encryption scheme for long messages that can not be encrypted directly with ElGamal scheme. The basic idea is that a message is encrypted with a block cipher with a randomly generated symmetric key, whereas the key is encrypted with ElGamal and sent along the encrypted message. The hybrid encryption consists of the following algorithms:

- $\mathsf{Gen}(\sigma)$: pick $\mathsf{sk} \leftarrow \mathbb{Z}_q^*$ and set $\mathsf{pk} := h = g^{sk}$, and output $(\mathsf{pk}, \mathsf{sk})$.
- $\mathsf{Gen}^S(\sigma)$: pick $s \leftarrow \mathbb{Z}_q^*$ and set $\mathsf{ck} := g^s$, and output ck.
- $\mathsf{SEnc}_{\mathsf{ck}}(m)$: output $e := AESEnc_{H(ck)}(m)$, where H is a cryptographic hash function and $AESEnc$ is an encryption method of AES block cipher.
- $\mathsf{SDec}_{\mathsf{ck}}(e)$: output $m := AESDec_{H(ck)}(e)$, where $AESDec$ is a decryption method of AES block cipher.
- $\mathsf{HEnc}_{\mathsf{pk}}(m, r, ck)$: output $e := (e_1, e_2) = (Enc'_{\mathsf{pk}}(ck, r), SEnc_{\mathsf{ck}}(m))$ is a hybrid encryption.
- $\mathsf{HDec}_{\mathsf{sk}}(e)$: output $m := SDec_{\mathsf{ck}}(e_2)$, where $ck = Dec'_{\mathsf{sk}}(e_1)$.

2.1.4 Pedersen Commitment

In the unit vector zero-knowledge proof, we use Pedersen commitment as a building block. It is perfectly hiding and computationally binding under the discrete logarithm assumption. More specifically, it consists of the following 4 PPT algorithms. Note that those algorithms (implicitly) take as input the same group parameters, $\mathsf{param} \leftarrow \mathsf{Gen}_{\mathsf{gp}}(1^\lambda)$.

- $\mathsf{KeyGen}^{\mathsf{C}}(\mathsf{param})$: pick $s \leftarrow \mathbb{Z}_q^*$ and set $\mathsf{ck} := h = g^s$, and output ck.
- $\mathsf{Com}_{\mathsf{ck}}(m; r)$: output $c := g^m h^r$ and $d := (m, r)$.
- $\mathsf{Open}(c, d)$: output $d := (m, r)$.
- $\mathsf{Verify}_{\mathsf{ck}}(c, d)$: return valid if and only if $c = g^m h^r$.

Pedersen commitment is also additively homomorphic, i.e.

$$\mathsf{Com}_{\mathsf{ck}}(m_1; r_1) \cdot \mathsf{Com}_{\mathsf{ck}}(m_2; r_2) = \mathsf{Com}_{\mathsf{ck}}(m_1 + m_2; r_1 + r_2) \ .$$

2.2 Distributed Key Generation

Distributed key generation (DKG) is a fundamental building block of the voting process in our proposed treasury system. Ideally, the protocol termination should be guaranteed when up to $t = \lceil \frac{n}{2} \rceil - 1$ out of n committee members are corrupted. A naive way of achieving threshold distributed key generation is as follows. Each of the voting committee members C_i first generates a public/private key pair $(\mathsf{pk}_i, \mathsf{sk}_i) \leftarrow \mathsf{Gen}(\mathsf{param})$. Each C_i then posts pk_i to the blockchain and use $(t+1, n)$-threshold *verifiable secret sharing* (VSS) to share sk_i to all the other committee members. The combined voting public key can then be defined as $\mathsf{pk} := \prod_{i=1}^{n} \mathsf{pk}_i$.

However, this approach is problematic in the sense that the adversary can influence the distribution of the final voting public key by letting the corrupted committee members abort selectively. Alternatively, we will adopt the distributed key generation protocol proposed by Gennaro *et al.* [4]. In a nutshell, the protocol lets the committee members C_i first post a "commitment" of pk_i. After sharing the corresponding sk_i via $(t + 1, n)$-threshold VSS, the committee members C_i then reveal pk_i. We will use the blockchain to realize the broadcast channel and peer-to-peer channels. We give a full description of our distributed key generation protocol. It is adapted from the DKG proposed by Gennaro *et al.*, which allows us to accommodate up to $t < n/2$ malicious players in the protocol. That is, guaranteeing that with $\lfloor \frac{n}{2} \rfloor + 1$ honest players, all the players should be able to agree on a uniformly random public key pk such that no malicious players can influence the distribution of the generated public key. The corresponding secret key is shared among all the players.

Protocol description. Given (g, h) as the common reference string (CRS), let $\mathsf{C} := \{\mathsf{C}_1, \mathsf{C}_2, ..., \mathsf{C}_k\}$ be the set of election committee members, and let $(\mathsf{pk}_i, \mathsf{sk}_i), (\mathsf{pk}_i^*, \mathsf{sk}_i^*)$ be two pairs of public/secret keys[6] associated with $C_i, i \in [k]$. Note that $(\mathsf{pk}_i^*, \mathsf{sk}_i^*)$ is used to communicate with other committee members, while $(\mathsf{pk}_i, \mathsf{pk}_i)$ is used to generate a shared election key. The adversary is able to corrupt up to $t < k/2$ committee members.

In the first round, each committee member C_i sets $a_{i,0} = \mathsf{sk}_i$ and picks random $a_{i,1}, ...a_{i,t}$ and random $a'_{i,0}, ..., a'_{i,t}$ where t is the maximum number of members that can be corrupted. Each member then define two polynomials of degree t of the form:

$$f_i(x) = a_{i,0} + a_{i,1}x + ... + a_{i,t}x^t$$

$$f'_i(x) = a'_{i,0} + a'_{i,1}x + ... + a'_{i,t}x^t$$

Therefore, each committee member C_i contributes $\mathsf{sk}_i = a_{i,0} = f(0)$ to the combined election secret key sk. Furthermore, to confirm the correctness of commitments, each member C_i posts a corresponding commitment $E_{i,l} = g^{a_{i,l}} h^{a'_{i,l}}$, $l \in [0, t]$, on the blockchain. For every other member of the election committee, each C_i computes $s_{i,j} = f_i(j)$ and $s'_{i,j} = f'_i(j)$ and posts to the blockchain, the encryption of $s_{i,j}$ and $s'_{i,j}$ under the public key pk_j^* of C_j (note: $j \neq i$). Note that only C_j can decrypt these commitments. This signifies the end of the first round.

In the second round, each committee member C_j fetches all $e_{i,j}$ and $e'_{i,j}$ encrypted under their public key from the blockchain and decrypt them using their private key sk_j^* to obtain their corresponding shares $s_{i,j}$ and $s'_{i,j}$. In order to verify that the shares they have received are valid, each committee member checks if: $g^{s_{i,j}} h^{s'_{i,j}} = \prod_{l=0}^{t} (E_{i,l})^{i^l}$ for $i \in [k], i \neq j$. Where this check fails, C_j posts a complain against C_i by revealing the decrypted shares and the evidence (in a form of a non-interactive ZK proof), that shares are decrypted correctly (see Fig. 4 for more details). Members with at least one valid complain against them are disqualified from participating in the key generation process.

In the third round, following the disqualification of some members, each qualified committee member C_i, $i \in \mathcal{J}$ posts $A_{i,l} := g^{a_{i,l}}$ for $\ell \in [0, .., t]$ to the blockchain. Note that secret keys sk_i of qualified committee members can be reconstructed as follows: $\mathsf{sk}_i := \sum_{j \in [\mathcal{J}]} \gamma_i \cdot s_{i,j}$, where $\gamma_i := \prod_{\ell \in \mathcal{J} \setminus \{i\}} \frac{\ell}{\ell - i}$.

In the fourth round, each qualified committee member C_i checks if : $g^{s_{j,i}} = \prod_{\ell=0}^{t} (A_{j,\ell})^{i^\ell}$ for $j \in \mathcal{J}, j \neq i$. Where the check fails, C_i posts a complaint against C_j together with the evidence $(s_{j,i}, s'_{j,i})$ on the blockchain, such that $g^{s_{j,i}} h^{s'_{j,i}} = \prod_{\ell=0}^{t} (E_{j,\ell})^{i^\ell}$ and $g^{s_{j,i}} \neq \prod_{\ell=0}^{t} (A_{j,\ell})^{i^\ell}$. Members with at least one valid complain against them are disqualified from further participation.

In the fifth and final round, each qualified committee member C_i checks if complaints raised against other committee members in the fourth round are valid. For all members, against whom valid complaints in the forth round were raised, other qualified committee members post shares $s_{j,i}$ they received from them. Therefore, allowing everyone to reconstruct secret keys of failed committee members as: $\mathsf{sk}_j := \sum_{i \in [\mathcal{J}]} \gamma_j \cdot s_{j,i}$, where $\gamma_j := \prod_{\ell \in \mathcal{J} \setminus \{j\}} \frac{\ell}{\ell - j}$, and redefine $A_{j,0} := g^{\mathsf{sk}_j}$.
Finally, the shared election public key is calculated as $\mathsf{pk} := \prod_{j \in [\mathcal{J}]} A_{j,0}$. Note that since all needed data to calculate pk is posted on the blockchain, every node in the network can reconstruct pk locally.

Note that, when implemented in a decentralized blockchain setting, it is supposed that all communications among committee members are done through the blockchain. It means that on each round they will post a transaction with relevant data that will also be checked by the whole network (where possible). At the end of the final round, every node in the network will be able to reconstruct a shared election public key without any additional messages from the committee.

[6] Note that we named public keys as *pubKey* and *committeePubKey* correspondingly in the definition of a committee registration transaction [1.1]

The full DKG protocol is summarized in Figure 4.[7] The NIZK used in Round 2 is described in Fig. 5.[8]

Distributed key generation Π_{DKG}

Round 1: Each committee member C_i does the following:

- Set $a_{i,0} = sk_i$
- Pick random $a_{i,1}, a_{i,2}, \dots, a_{i,t}, b_{i,0}, b_{i,1}, \dots, b_{i,t} \leftarrow \mathbb{Z}_p$.
- Define two polynomials $f_i(x) := \sum_{\ell=0}^{t} a_{i,\ell} x^\ell$ and $f'_i(x) := \sum_{\ell=0}^{t} b_{i,\ell} x^\ell$.
- For $\ell \in [t]$, post $E_{i,\ell} := g^{a_{i,\ell}} h^{b_{i,\ell}}$ on the blockchain.
- For every other C_j, $j \in [k], j \neq i$, compute $s_{i,j} := f_i(j)$ and $s'_{i,j} := f'_i(j)$ and post $e_{i,j} \leftarrow \mathsf{HEnc}_{\mathsf{pk}_j^*}(s_{i,j}, r, ck_{i,j})$ and $e'_{i,j} \leftarrow \mathsf{HEnc}_{\mathsf{pk}_j^*}(s'_{i,j}, r', ck'_{i,j})$ on the blockchain, where $r, r' \leftarrow \mathbb{Z}_p$, $ck_{i,j}, ck'_{i,j} \leftarrow E(\mathbb{F}_p)$ are randomly generated. Only C_j can decrypt these shares.

Round 2: Each committee member C_i does the following:

- Fetch $\{(e_{j,i}, e'_{j,i})\}_{j\in[k],j\neq i}$ from the blockchain, parse as $e_{j,i} = (e_{(j,i),1}, e_{(j,i),2})$, $e'_{j,i} = (e'_{(j,i),1}, e'_{(j,i),2})$, and use pk_i^* to decrypt them, obtaining the corresponding shares $\{(s_{j,i}, s'_{j,i})\}_{j\in[k],j\neq i}$.
- For $j \in [k], j \neq i$, check if $g^{s_{j,i}} h^{s'_{j,i}} = \prod_{\ell=0}^{t} (E_{j,\ell})^{i^\ell}$. If not, post a complaint against C_j by revealing the evidence: $(s_{j,i}, s'_{j,i}, e_{j,i}, e'_{j,i}, ck_{j,i}, ck'_{j,i}, \pi, \pi')$, where π, π' are zero-knowledge proofs generated as follows (see Fig. 5 for full NIZK description):

$$\pi \leftarrow \mathsf{NIZK} \left\{ \begin{array}{l} (e_{(j,i),1}, \mathsf{ck}_{j,i}, \mathsf{pk}_i^*), (\mathsf{sk}_i^*) : \\ ck_{j,i} = \mathsf{Dec}'_{\mathsf{sk}_i^*}(e_{(j,i),1}) \wedge \mathsf{pk}_i^* = g^{\mathsf{sk}_i^*} \wedge (\mathsf{pk}_i^*, \mathsf{sk}_i^*) \in \mathcal{R}_{\text{PKE}} \end{array} \right\}$$

- (One valid complain against C_j, $j \in [k]$ will disqualify C_j.)

Round 3: Define the indices of the qualified set of committee members as $\mathcal{J}$. Each committee member C_i does the following:

- For $\ell \in [t]$, post $A_{i,\ell} := g^{a_{i,\ell}}$ to the blockchain.
- Note that a secret key sk_i of C_i, $i \in \mathcal{J}$, can be reconstructed as follows: $\mathsf{sk}_i := \sum_{j\in[\mathcal{J}]} \gamma_i \cdot s_{i,j}$, where $\gamma_i := \prod_{\ell\in\mathcal{J}\setminus\{i\}} \frac{\ell}{\ell-i}$.

Round 4: Each committee member C_i does the following:

- For $j \in \mathcal{J}, j \neq i$, check if $g^{s_{j,i}} = \prod_{\ell=0}^{t} (A_{j,\ell})^{i^\ell}$. If not, post a complaint against C_j together with the evidence $(s_{j,i}, s'_{j,i})$ on the blockchain, such that $g^{s_{j,i}} h^{s'_{j,i}} = \prod_{\ell=0}^{t} (E_{j,\ell})^{i^\ell}$ and $g^{s_{j,i}} \neq \prod_{\ell=0}^{t} (A_{j,\ell})^{i^\ell}$.

Round 5: Each committee member C_i does the following:

- If there is a valid complain against C_j, $j \in \mathcal{J}$, then post the share $s_{j,i}$ on the blockchain. (Everyone can reconstruct $\mathsf{sk}_j := \sum_{i\in[\mathcal{J}]} \gamma_j \cdot s_{j,i}$, where $\gamma_j := \prod_{\ell\in\mathcal{J}\setminus\{j\}} \frac{\ell}{\ell-j}$, and calculate $A_{j,0} := g^{\mathsf{sk}_j}$.)
- Return the election public key as $\mathsf{pk} := \prod_{j\in[\mathcal{J}]} A_{j,0}$.

Figure 4: Distributed key generation Π_{DKG}

[7] See implementation here:
https://github.com/input-output-hk/treasury-crypto/.../protocol/keygen

[8] See implementation here:
https://github.com/input-output-hk/treasury-crypto/../protocol/nizk/ElgamalDecrNIZK.scala

$$\mathsf{NIZK}\{(\mathsf{pk}, C, M), (\mathsf{sk}) : M = \mathsf{Dec}'_{\mathsf{sk}}(C) \wedge \mathsf{pk} = g^{\mathsf{sk}}\}$$

Statement: Public key, $\mathsf{pk} := h \in \mathbb{G}$, ciphertext $C := (C_1, C_2)$, and the plaintext $M := g^m \in \mathbb{G}$
Witness: $\mathsf{sk} \in \mathbb{Z}_p$

Prover:

- Pick random $w \leftarrow \mathbb{Z}_p$;
- Compute $D = (C_1)^{sk}$, $A_1 := g^w$, and $A_2 := (C_1)^w$
- Compute $e = \mathsf{hash}(C, D, A_1, A_2)$ and $z = \mathsf{sk} * e + w$
- Return $\pi := (A_1, A_2, z)$

Verifier:

- Compute $D = C_2 \cdot M^{-1}$
- Compute $e = \mathsf{hash}(C, D, A_1, A_2)$
- Verify that:
  - $g^z = h^e \cdot A_1$, and
  - $(C_1)^z = D^e \cdot A_2$

Figure 5: Non-Interactive Zero Knowledge proof for correct ElGamal decryption

2.3 Ballots Casting

Let m be the number of experts. Let $e_i^{(m)} \in \{0,1\}^m$ be a unit vector, where its i-th, $i \in [1,..,m]$, coordinate is 1 and the rest coordinates are 0. Similarly, let $e_j^{(3)} \in \{0,1\}^3$ be a unit vector. Denote $(e_i^{(m)}, e_j^{(3)})$ as the concatenation of $e_i^{(m)}$ and $e_j^{(3)}$. We also abuse the notation to denote $e_0^{(\ell)}$ as an ℓ-vector containing all 0's.

The expert's choice is represented by one of the unit vectors $(e_1^{(3)}, e_2^{(3)}, e_3^{(3)})$, where $e_1^{(3)}$ stands for 'Abstain', $e_2^{(3)}$ stands for 'Yes', and $e_3^{(3)}$ stands for 'No'.

The voter's choice is represented by the concatenation of two unit vectors $(e_i^{(m)}, e_j^{(3)})$, where $e_i^{(m)}$, $i \in [0, m]$ stands for delegation choice and $e_i^{(3)}$, $i \in [0, 2]$ stands for voting choice. If the voter wants to vote directly, he creates a unit vector $(e_0^{(m)}, e_j^{(3)}), j \in \{1, 2, 3\}$. Otherwise, if the voter wants to delegate his voting power to E_i, he sets $(e_i^{(m)},\ e_0^{(3)})$, where $i \in [1, m]$ is an index of a registered expert.

Before publishing voter's/expert's choices on the blockchain they are encrypted. Let us denote a coordinate-wise encryption of $e_i^{(\ell)}$ as $\mathsf{Enc}_{\mathsf{pk}}(e_i^{(\ell)})$, i.e.

$$\mathsf{Enc}_{\mathsf{pk}}(e_i^{(\ell)}) = \mathsf{Enc}_{\mathsf{pk}}(e_{i,1}^{(\ell)}), \ldots, \mathsf{Enc}_{\mathsf{pk}}(e_{i,\ell}^{(1)}),$$

where $e_i^{(\ell)} = (e_{i,1}^{(\ell)}, \ldots, e_{i,\ell}^{(\ell)})$ and pk is a shared election public key generated during the DKG stage.

Since tally is calculated homomorphically by summing up encrypted unit vectors, it is crucial to verify that encryptions are formed correctly (i.e., that they indeed encrypt unit vectors). To do so, each voter/expert creates a special zero-knowledge proof for each published encrypted unit vector that proves it is correct. The corresponding ZK proof is described in section [2.4]. Note that a voter/expert publishes a separate unit vector for each submitted proposal.

The protocol for ballot casting is depicted in Figure 6.[9]

[9] See implementation here:
https://github.com/input-output-hk/treasury-crypto/../protocol/voting/Voter.scala

Preparation phase:

- Retrieve shared election public key pk generated with Π_{DKG} as desceibed in Figure 4.

Ballots casting phase:

- Upon issuing a voting ballot, an expert E_i does the following:
  - For each submitted proposal $p_k \in \mathcal{P}$:
    - create a unit vector $e^{(3)}_{k,\ell}$ according to his choice (e.g, $e^{(3)}_{k,1}$ for "Abstain", $e^{(3)}_{k,2}$ for "Yes", and $e^{(3)}_{k,3}$ for "No");
    - pick randomness $r_{k,1}, r_{k,2}, r_{k,3} \leftarrow \mathbb{Z}_p$ and compute $c_{k,t} \leftarrow \mathsf{Enc}_{\mathsf{pk}}(e^{(3)}_{k,t}; r_{k,t})$, $t \in [3]$;
    - produce a unit vector proof π_k showing that $\{c_{k,t}\}_{t\in[3]}$ encrypts a unit vector[a].
  - Send transaction with $(\mathit{Ballot}, (\mathsf{E}_i, \{\{c_{k,t}\}_{t\in[3]}, \pi_k\}_{k\in|\mathcal{P}|}))$ to the blockchain.
- Upon issuing a voting ballot, a voter V_i does the following:
  - For each submitted proposal $p_k \in \mathcal{P}$:
    - create a unit vector $e^{(m+3)}_{k,\ell}$ so that:
      - if V_i wants to delegate, then $e^{(m+3)}_{k,\ell} := (e^{(m)}_{k,i}, e^{(3)}_0)$, where $i \in [1, m]$ is an index of an expert;
      - otherwise, if V_i wants to vote directly, then $e^{(m+3)}_{k,\ell} := (e^{(m)}_0, e^{(3)}_{k,i})$, where $i \in \{1, 2, 3\}$ depends on the choice (Abstain, Yes, or No correspondingly);
    - pick randomness $r_{k,1}, \ldots, r_{k,m+3} \leftarrow \mathbb{Z}_q$ and compute $u_{k,t} \leftarrow \mathsf{Enc}_{\mathsf{pk}}(e^{(m+3)}_{k,t}; r_{k,t})$, $t \in [m + 3]$;
    - produce a unit vector proof π_k showing that $\{c_{k,t}\}_{t\in[m+3]}$ encrypts a unit vector.
  - Send transaction with $(\mathit{Ballot}, (\mathsf{V}_i, \{\{c_{k,t}\}_{t\in[m+3]}, \pi_k\}_{k\in|\mathcal{P}|}))$ to the blockchain.

[a] Due to complexity of the NIZK proof, we described it in a separate section [2.4]

Figure 6: Ballots casting

2.4 Unit Vector ZK Proof

We denote a unit vector of length n as $\mathbf{e}^{(n)}_i = (e_{i,0}, \ldots, e_{i,n-1})$, where its i-th coordinate is 1 and the rest coordinates are 0. Note that in this section we diverge from the previously established notation, where $\mathbf{e}^{(n)}_0$ was defined as a vector of zeroes, now we do not consider zero-vectors at all and the lower index signifies the position of the "1" bit in the vector, e.g., $\mathbf{e}^{(5)}_0 = (10000), \mathbf{e}^{(5)}_4 = (00001)$.

Conventionally, to show a vector of ElGamal ciphertexts element-wise encrypt a unit vector, Chaum-Pedersen proofs [3] are used to show each of the ciphertexts encrypts either 0 or 1 (via Sigma OR composition) and the product of all ciphertexts encrypts 1[10]. Such kind of proof is used in many well-known voting schemes, e.g., Helios. However, the proof size is linear in the length of the unit vector, and thus the communication overhead is quite significant when the unit vector length becomes larger.

In this section, we propose a novel special honest verifier ZK (SHVZK)[11] proof for a unit vector that allows a prover to convince a verifier that the vector of ciphertexts $(C_0, \ldots, C_{n-1})$ encrypts a unit vector $\mathbf{e}^{(n)}_i$, $i \in [0, n-1]$ with $O(\log n)$ proof size. Without loss of generality, assume n is a perfect power of 2. If not, we append $\mathsf{Enc}_{\mathsf{pk}}(0; 0)$ (i.e., trivial ciphertexts) to make the total number of ciphertexts to be the next power of 2. We transform the proposed SHVZK protocol to a non-interactive ZK (NIZK) using the Fiat-Shamir heuristic.

The basic idea of our construction is inspired by [5], where Groth and Kohlweiss proposed a Sigma protocol for the prover to show that he knows how to open one out of many commitments. The key idea behind our construction is that there exists a data-oblivious algorithm that can take input as $i \in [0, .., n-1]$ and output the unit vector $\mathbf{e}^{(n)}_i$. Let $i_1, \ldots, i_{\log n}$ be the binary representation of i. The algorithm is depicted in Fig. 7.

Intuitively, we let the prover first bit-wisely commit the binary presentation of $i \in [0, n-1]$ for the unit vector $\mathbf{e}^{(n)}_i$. The prover then shows that each of the commitments of $(i_1, \ldots, i_{\log n})$ indeed contain 0

[10] This approach is implemented in https://github.com/input-output-hk/treasury-crypto/.../protocol/nizk/unitvectornizk. But note that it is not used in the final protocol.

[11] See implementation here: https://github.com/input-output-hk/treasury-crypto/.../protocol/nizk/shvzk

The algorithm that maps $i \in [0, n-1]$ to $\mathbf{e}_i^{(n)}$

```
Input: index $i = (i_1, \ldots, i_{\log n}) \in \{0,1\}^{\log n}$
Output: unit vector $\mathbf{e}_i^{(n)} = (e_{i,0}, \ldots, e_{i,n-1}) \in \{0,1\}^n$
1. For $\ell \in [\log n]$, set $b_{\ell,0} := 1 - i_\ell$ and $b_{\ell,1} := i_\ell$;
2. For $j \in [0, n-1]$, set $e_{i,j} := \prod_{\ell=1}^{\log n} b_{\ell, j_\ell}$, where $j_1, \ldots, j_{\log n}$ is the binary representation of $j$;
3. Return $\mathbf{e}_i^{(n)} = (e_{i,0}, \ldots, e_{i,n-1})$;
```

Figure 7: The algorithm that maps $i \in [0, n-1]$ to $\mathbf{e}_i^{(n)}$

or 1, using the Sigma protocol proposed in Section 2.3 of [5]. Note that in the 3rd move of such a Sigma protocol, the prover reveals a degree-1 polynomial of the committed message. Denote $z_{\ell,1} := i_\ell x + \beta_\ell$, $\ell \in [\log n]$ as the corresponding degree-1 polynomials, where β_ℓ are chosen by the prover and x is chosen by the verifier. By linearity, we can also define $z_{\ell,0} := x - z_{\ell,1} = (1 - i_\ell)x - \beta_\ell$, $\ell \in [\log n]$. According to the algorithm described in Fig. 7, for $j \in [0, n-1]$, let $j_1, \ldots, j_{\log n}$ be the binary representation of j, and the product $\prod_{\ell=1}^{\log n} z_{\ell, j_\ell}$ can be viewed as a degree-$(\log n)$ polynomial of the form

$$p_j(x) = e_{i,j} x^{\log n} + \sum_{k=0}^{\log n - 1} p_{j,k} x^k$$

for some $p_{j,k}$, $k \in [0, \log n - 1]$. We then use batch verification to show that each of C_j indeed encrypts $e_{i,j}$. More specifically, for a randomly chosen $y \leftarrow \mathbb{Z}_p$, let $E_j := (C_j)^{x^{\log n}} \cdot \mathsf{Enc}(-p_j(x); 0)$; the prover needs to show that $E := \prod_{j=0}^{n-1} (E_j)^{y^j} \cdot \prod_{k=0}^{\log n - 1} (D_k)^{x^k}$ encrypts 0, where $D_\ell := \mathsf{Enc}_{\mathsf{pk}}(\sum_{j=0}^{n-1} (p_{j,\ell} \cdot y^j); R_\ell)$, $\ell \in [0, \log n - 1]$ with fresh randomness $R_\ell \in \mathbb{Z}_p$. The construction is depicted in Fig. 8. Both the prover and the verifier shares a common reference string (CRS), which is a Pedersen commitment key that can be generated using random oracle. The prover first commits to each bits of the binary representation of i, and the commitments are denoted as I_ℓ, $\ell \in [\log n]$. Subsequently, it produces B_ℓ, A_ℓ as the first move of the Sigma protocol in Sec. 2.3 of [5] showing I_ℓ commits to 0 or 1. Jumping ahead, later the prover will receive a challenge $x \leftarrow \{0,1\}^\lambda$, and then it computes the third move of the Sigma protocols by producing $\{z_\ell, w_\ell, v_\ell\}_{\ell=1}^{\log n}$. To enable batch verification, before that, the prover is given another challenge $y \leftarrow \{0,1\}^\lambda$ in the second move.

The verification consists of two parts (Fig. 9). In the first part, the verifier checks the following equations to ensure that I_ℓ commits to 0 or 1.

- $(I_\ell)^x \cdot B_\ell = \mathsf{Com}_{\mathsf{ck}}(z_\ell; w_\ell)$
- $(I_\ell)^{x - z_\ell} \cdot A_\ell = \mathsf{Com}_{\mathsf{ck}}(0; v_\ell)$

In the second part, the verifier checks if

$$\prod_{j=0}^{n-1} \left((C_j)^{x^{\log n}} \cdot \mathsf{Enc}_{\mathsf{pk}}(-\prod_{\ell=1}^{\log n} z_{\ell, j_\ell}; 0) \right)^{y^j} \cdot \prod_{\ell=0}^{\log n - 1} (D_\ell)^{x^\ell}$$

is encryption of 0 by asking the prover to reveal the randomness.

Unit vector ZK argument (Prover)

CRS: the commitment key $\mathsf{ck} = hash(\mathsf{pk})$
Statement: the public key pk and the ciphertexts $C_0 := \mathsf{Enc}_{\mathsf{pk}}(e_{i,0}; r_0), \ldots, C_{m-1} := \mathsf{Enc}_{\mathsf{pk}}(e_{i,m-1}; r_{m-1})$
Witness: the unit vector $\mathbf{e}_i^{(m)} \in \{0,1\}^m$ and the randomness $r_0, \ldots, r_{m-1} \in \mathbb{Z}_p$

Prover:

- If the number of ciphertexts m is not a perfect power of 2, extend the set with $C_j := \mathsf{Enc}_{\mathsf{pk}}(0; 0), j \in [m, .., n-1]$, where n is a perfect power of 2
- Let $\{i_k\}_{k \in [0,..,log(n)-1]}$ be a binary representation of the index i (e.g. if $i = 3$ and $n = 5$, its binary represantation is ”00011”, so that $i_0 = 0$, $i_1 = 0$, $i_2 = 0$, $i_3 = 1$, $i_4 = 1$)
- For $\ell = 0, \ldots, \log n - 1$ do the following:
  - Pick random $\alpha_\ell, \beta_\ell, \gamma_\ell, \delta_\ell \leftarrow \mathbb{Z}_p$;
  - Compute $I_\ell := \mathsf{Com}_{\mathsf{ck}}(i_\ell; \alpha_\ell)$, $B_\ell := \mathsf{Com}_{\mathsf{ck}}(\beta_\ell; \gamma_\ell)$ and $A_\ell := \mathsf{Com}_{\mathsf{ck}}(i_\ell \cdot \beta_\ell; \delta_\ell)$;
- Compute first verifier challange (using Fiat-Shamir heuristic):

$$c_y = hash(\mathsf{pk} \mid \{C_l\}_{\ell=0}^{n-1} \mid \{I_\ell, B_\ell, A_\ell\}_{\ell=0}^{\log n - 1})$$

- For $j = 0, \ldots, n-1$ compute polynomials $p_j(x)$ of the following form:

$$p_j(x) = e_{i,j} x^{\log n} + \sum_{k=0}^{\log n - 1} p_{j,k} x^k,$$

  where $e_{i,j}$ is a j-th bit in the vector $\mathbf{e}_i^{(m)}$ and $p_{j,k}, k \in [0, .., log(n) - 1]$ is a k-th coefficient of the polynomial p_j.
  A polynomials $p_j(x)$ can be constructed using the following procedure:
  - Let $j_k, k \in [0, .., \log n - 1]$ be a binary representation of the index j
  - Compute $p_j(x) = \prod_{\ell=0}^{\log n - 1} z_\ell^{j_\ell}$, where $j_\ell \in \{0, 1\}$ and

$$z_\ell^1 = i_\ell x + \beta$$
$$z_\ell^0 = x - z_\ell^1 = (1 - i_\ell)x - \beta$$

- For $\ell = 0, \ldots, \log n - 1$ compute D_ℓ as follows:
  - Pick random $R_\ell \leftarrow \mathbb{Z}_p$, and
  - Compute $D_\ell := \mathsf{Enc}_{\mathsf{pk}}\big(\sum_{j=0}^{n-1}(p_{j,\ell} \cdot y^j); R_\ell\big)$
- Compute second verifier challange (using Fiat-Shamir heuristic):

$$c_x = hash(\mathsf{pk} \mid \{C_l\}_{\ell=0}^{n-1} \mid \{I_\ell, B_\ell, A_\ell\}_{\ell=0}^{\log n - 1}) \mid \{D_l\}_{\ell=0}^{\log n - 1})$$

- For $\ell = 0, \ldots, \log n - 1$ compute z_ℓ, w_ℓ and v_ℓ as follows:
  - $z_\ell := i_\ell \cdot c_x + \beta_\ell$
  - $w_\ell := \alpha_\ell \cdot c_x + \gamma_\ell$
  - $v_\ell := \alpha_\ell (c_x - z_\ell) + \delta_\ell$
- Compute $R := \sum_{j=0}^{n-1}(r_j \cdot (c_x)^{\log n} \cdot (c_y)^j) + \sum_{\ell=0}^{\log n - 1}(R_\ell \cdot (c_x)^\ell)$
- Return proof $\pi := (\{I_\ell, B_\ell, A_\ell\}_{\ell=0}^{\log n - 1}),\ \{D_l\}_{\ell=0}^{\log n - 1}),\ \{z_\ell, w_\ell, v_\ell\}_{\ell=0}^{\log n - 1},\ R)$

Figure 8: Unit vector ZK argument: proof creation

Unit vector ZK argument (Verifier)

CRS: the commitment key $\mathsf{ck} = hash(\mathsf{pk})$
Statement: the public key pk and the ciphertexts $C_0 := \mathsf{Enc}_{\mathsf{pk}}(e_{i,0}; r_0), \ldots, C_{m-1} := \mathsf{Enc}_{\mathsf{pk}}(e_{i,m-1}; r_{m-1})$
Proof: $\pi := (\{I_\ell, B_\ell, A_\ell\}_{\ell=0}^{\log n-1}),\ \{D_l\}_{\ell=0}^{\log n-1}),\ \{z_\ell, w_\ell, v_\ell\}_{\ell=0}^{\log n-1},\ R)$

Verification:

- If the number of ciphertexts m is not a perfect power of 2, extend the set with $C_j := \mathsf{Enc}_{\mathsf{pk}}(0; 0), j \in [m, .., n-1]$, where n is a perfect power of 2
- Compute first verifier challenge: $c_y = hash(\mathsf{pk} \mid \{C_l\}_{\ell=0}^{n-1} \mid \{I_\ell, B_\ell, A_\ell\}_{\ell=0}^{\log n-1})$
- Compute second verifier challenge: $c_x = hash(\mathsf{pk} \mid \{C_l\}_{\ell=0}^{n-1} \mid \{I_\ell, B_\ell, A_\ell\}_{\ell=0}^{\log n-1}) \mid \{D_l\}_{\ell=0}^{\log n-1})$
- Verify that for $\ell = 0, \ldots, \log n - 1$ the following equations are true:
  - $(I_\ell)^{c_x} \cdot B_\ell = \mathsf{Com}_{\mathsf{ck}}(z_\ell; w_\ell)$
  - $(I_\ell)^{c_x - z_\ell} \cdot A_\ell = \mathsf{Com}_{\mathsf{ck}}(0; v_\ell)$
- Verify the following equation is true:

$$\prod_{j=0}^{n-1} \left((C_j)^{(c_x)^{\log n}} \cdot \mathsf{Enc}_{\mathsf{pk}}(-\prod_{\ell=0}^{\log n-1} z_\ell^{j_\ell}; 0)\right)^{(c_y)^j} \cdot \prod_{\ell=0}^{\log n-1} (D_\ell)^{(c_x)^\ell} = \mathsf{Enc}_{\mathsf{pk}}(0; R),$$

where $z_j^1 = z_j$, $z_j^0 = c_x - z_j$, and $j_\ell \in \{0, 1\}$ is a binary representation of j, $\ell \in [0, .., \log n - 1]$.

Figure 9: Unit vector ZK argument: proof verification

2.5 Tally Protocol

At the tally stage, committee members jointly decrypt the results of the voting without revealing personal choices of experts and voters.

Let denote as $\mathcal{B}^E := \{B^E_1, \ldots, B^E_m\}$ a set of ballots received from experts and as $\mathcal{B}^V := \{B^V_1, \ldots, B^V_n\}$ a set of ballots received from voters. Recall that a ballot contains encrypted unit vectors with choices (one unit vector for each proposal). Unit vectors of experts are 3-bits long, while unit vectors of voters are $(m+3)$-bits long, where m is the number of registered experts.

Without loss of generality, let assume that we have only one proposal, so that a ballot contains only one encrypted unit vector. Let denote bit-wise encryptions of a unit vector of a voter V_i as $c_{v_i,j},\ j \in [0,..,m{+}2]$ and an encrypted unit vector of an expert E_i as $c_{e_i,j},\ j \in [0,..2]$. Note that $c_{v_i,j},\ j \in [0,..,m-1]$ encrypt bits of a voter's unit vector that signify his delegation choice, while the rest of encrypted bits $c_{v_i,j}, j\ \in [m, m+1, m+2]$ signify his voting choice ("Abstain","Yes","No").

Joint decryption is accomplished in two steps. At the first step, the number of delegations is calculated and jointly decrypted. To do so, the number of delegations for each expert is calculated homomorphically by summing up delegations part of voter's unit vectors (i.e., first m bits). Then each committe member produces a decryption share with his secret key, that was used to generate shared election public key. Given decryption shares from all members, the number of delegations to each expert can be decrypted.

At the second step, the number of votes for each choice ("Abstain","Yes","No") is calculated homomorphically by summing up corresponding part of voter's and expert's unit vectors (weighed by the voting power for voters and delegated voting power for experts). Then, committe members produce decryption shares, which allows everyone to decrypt and verify the final tally.

The full protocol is described in Fig. 10 and Fig. 11.[12] Note that in case of several proposals, the protocol is performed for each proposal separately in parallel (for better efficiency, committee members may combine published data into a single transaction).

[12] See implementation here:
https://github.com/input-output-hk/treasury-crypto/../protocol/voting/Tally.scala
https://github.com/input-output-hk/treasury-crypto/../protocol/decryption/DecryptionManager.scala

Entities:

Committee members $\mathcal{C} := \{C_1, \ldots, C_l\}$. Note that here we refer to the members that were qualified after the DKG stage.

Voters $\mathcal{V} := \{V_1, \ldots, V_n\}$.

Experts $\mathcal{E} := \{E_1, \ldots, E_m\}$

Input data:

Set of voter's ballots: $C_V := \{c_{v_1}, \ldots, c_{v_n}\}$, where $c_{v_i} = \{c_{v_i,0}, \ldots, c_{v_i,m+2}\}$ is an encrypted unit vector with choice of a voter V_i.[a]

Set of expert's ballots: $C_E := \{c_{e_1}, \ldots, c_{e_m}\}$, where $c_{e_i} = \{c_{e_i,0}, c_{e_i,1}, c_{e_i,2}\}$ is an encrypted unit vector with choice of an expert E_i.

Round 1 (delegation decryption): Each committee member C_j does the following:

- For $i = 1, \ldots, m$ compute homomorphically the number of delegations d_{e_i} for each expert E_i:

$$d_{e_i} = \prod_{l=0}^{n-1} (c_{v_l,i})^{\alpha_l},$$

  where α_l is a voting power of the voter V_l (amount of deposited stake).

- For $i = 1, \ldots, m$ compute decryption shares for delegation sums as follows:
  - Parse d_{e_i} to $(d_{e_i,1}, d_{e_i,2})$ (recall that c_{e_i} and, correspondingly, d_{e_i} are ElGamal ciphertexts, which comprise of 2 group elements [2.1.2])
  - Compute $D_{j,e_i} = (d_{e_i,1})^{sk_j}$, where sk_j is a secret key[b] of the committee member C_j, and a proof π_i that the share is generated correctly (see Fig. 12 for NIZK description):

$$\pi_{j,i} \leftarrow \mathsf{NIZK}\left\{ \begin{array}{l} (\mathsf{pk}_j, d_{e_i,1}, D_{j,e_i}), (\mathsf{sk}_j) : \\ D_{j,e_i} = (d_{e_i,1})^{sk_j} \ \wedge pk_j = g^{sk_j} \ \wedge \ (\mathsf{pk}_j, \mathsf{sk}_j) \in \mathcal{R}_{\mathrm{PKE}} \end{array} \right\}$$

- Publish $\{D_{j,e_i}, \pi_i\}_{i \in [m]}$ to the blockchain

Committee members $C_j, j \in \mathcal{F}$, that failed to submit decryption shares are disqualified from further participation.

Round 2 (delegation decryption shares recovery):
Each qualified committee member $C_i,\ i \in \mathcal{J}$ does the following:

- If there are such committee members $C_j,\ j \in \mathcal{F}$, who failed to post valid decryption shares, then C_i posts to the blockchain shares $s_{j,i},\ j \in \mathcal{F}$ of C_j's secret key obtained during the Round 1 of the DKG protocol (Figure 4).
- Everyone can reconstruct a secret key[c] of C_j: $sk_j := \sum_{i \in [\mathcal{J}]} \gamma_j \cdot s_{j,i}$, where $\gamma_j := \prod_{\ell \in \mathcal{J} \setminus \{j\}} \frac{\ell}{\ell - j}$, and reconstruct his decryption shares D_{j,e_i}.

After obtaining all decryption shares D_{j,e_i} from all committee members $C_j,\ j \in [1,..,l]$, everyone can decrypt delegation results as follows:

- For $i = 1, \ldots, m$, where m is the number of experts compute $D_{e_i} = \prod_{j=1}^{l} D_{j,e_i}$, where l is the number of committee members.
- Compute number of delegations to each expert E_i as $dlg_i = DLOG_g(d_{e_i,2} \cdot (D_{e_i})^{-1})$, where $d_{e_i,2}$ is obtained during the Round 1 and $DLOG$ is a discrete logarithm for the resulting group element. Note that the logarithm is the number of delegations so it can be effectively found by brute force.

Round 3 (tally results decryption): see continuation on Fig. 11

[a] We consider only one proposal per ballot, so there is only one encrypted unit vector

[b] Here we refer to the secret key sk_j that was used to generate shared election public key and for which a corresponding pk_j was registered by C_j.

[c] Note that there should be more than $l/2$ shares, where l is the number of committee members, to be able to reconstruct the secret key

Figure 10: Tally (Rounds 1,2)

Round 3 (tally results decryption): Each qualified committee member $C_j,\ j \in \mathcal{J}$ does the following:

- For $i = 0,..,2$ compute homomorphically the tally result r_i for each choice $i \in \{0,1,2\}$ ("Abstain","Yes","No" correspondingly) as follows:

$$r_i^E = \prod_{j=1}^{m} (c_{e_j,m+i})^{dlg_j}, \quad r_i^V = \prod_{j=1}^{n} (c_{v_j,m+i})^{\alpha_j}$$

$$r_i = r_i^E \cdot r_i^V$$

  where dlg_j is a delegated voting power to the expert E_j and α_j is a voting power of the voter V_j (amount of deposited stake).

- For $i = 0,..,2$ compute decryption shares for tally results as follows:
  - Parse r_i to $(r_{i,1}, r_{i,2})$ (recall that r_i is an ElGamal ciphertext, which comprises 2 group elements [2.1.2])
  - Compute decryption share $R_{j,i} = (r_{i,1})^{sk_j}$, where sk_j is a secret key of the committee member C_j, and a proof $\pi_{j,i}$ that the share is generated correctly (see Fig. 12 for NIZK description):

$$\pi_{j,i} \leftarrow \mathsf{NIZK}\left\{ \begin{array}{l} (\mathsf{pk}_j, r_{i,1}, R_{j,i}), (\mathsf{sk}_j) : \\ R_{j,i} = (r_{i,1})^{sk_j} \ \wedge pk_j = g^{sk_j} \ \wedge \ (\mathsf{pk}_j, \mathsf{sk}_j) \in \mathcal{R}_{\mathrm{PKE}} \end{array} \right\}$$

- Publish $\{R_{j,i}, \pi_{j,i}\}_{i \in [0,..,2]}$ to the blockchain

Committee members $C_j, j \in \mathcal{F}'$, that failed to submit decryption shares are disqualified from further participation.

Round 4 (tally decryption shares recovery):
Each qualified committee member $C_i,\ i \in \mathcal{J}'$ does the following:

- If there are such committee members $C_j,\ j \in \mathcal{F}'$, who failed to post valid decryption shares, then C_i posts to the blockchain shares $s_{j,i},\ j \in \mathcal{F}'$ of C_j's secret key obtained during the Round 1 of the DKG protocol (Figure 4).

Everyone can reconstruct a secret key of $C_j,\ j \in \mathcal{F}'$: $sk_j := \sum_{i \in [\mathcal{J}']} \gamma_j \cdot s_{j,i}$, where $\gamma_j := \prod_{\ell \in \mathcal{J}' \setminus \{j\}} \frac{\ell}{\ell - j}$

Everyone reconstructs missing decryption shares of disqualified committee members $C_j,\ j \in \mathcal{F} \cup \mathcal{F}'$ as $R_{j,i} = (r_{i,1})^{sk_j}$ for $i \in [0,..,2]$.

After obtaining all decryption shares $R_{j,i}$ from all committee members $C_j,\ j \in [1,..,l]$, everyone can decrypt tally results as follows:

- For $i = 0,..,2$ compute $R_i = \prod_{j=1}^{l} R_{j,i}$, where l is the number of committee members.
- Compute number of votes for each choice $i \in \{0,1,2\}$ as $res_i = DLOG_g(r_{i,2} \cdot (R_i)^{-1})$, where $r_{i,2}$ is computed as in Round 3 and $DLOG$ is a discrete logarithm for the resulting group element. Note that the logarithm is the number of votes so it can be effectively found by brute force.

Figure 11: Tally (Rounds 3,4)

$$\mathsf{NIZK}\{(\mathsf{pk}, C, D), (\mathsf{sk}) : D = (C_1)^{\mathsf{sk}} \wedge \mathsf{pk} = g^{\mathsf{sk}}\}$$

Statement: Public key, $\mathsf{pk} := h \in \mathbb{G}$, ciphertext $C := (C_1, C_2)$, and the decryption share $D := (C_1)^{sk} \in \mathbb{G}$
Witness: $\mathsf{sk} \in \mathbb{Z}_p$

Prover:

- Pick random $w \leftarrow \mathbb{Z}_p$;
- Compute $A_1 := g^w$, and $A_2 := (C_1)^w$
- Compute $e = \mathsf{hash}(C, D, A_1, A_2)$ and $z = \mathsf{sk} * e + w$
- Return $\pi := (A_1, A_2, z)$

Verifier:

- Compute d
- Compute $e = \mathsf{hash}(C, D, A_1, A_2)$
- Verify that:
  - $g^z = h^e \cdot A_1$, and
  - $(C_1)^z = D^e \cdot A_2$

Figure 12: Non-Interactive Zero Knowledge proof for correct ElGamal decryption share (note that this NIZK is basically the same as ElGamal decryption NIZK in Fig. 5 except slightly different interface)

2.6 Distributed Randomness Generation

As it was previously outlined in [Treasury System Overview], randomness generation is done by committee members. It is split into two phases:

1. **Random value commitment**, and
2. **Randomness revealing**.

Commitment phase is done during the post-voting stage. At this phase each qualified[13] committe member generates a random value and submits its encryption (under his private key) to the blockchain. No one else knows the committed random values of committee members. The commitment phase is formilized at Fig. 13.[14]

Randomness revealing phase is done during the pre-voting stage of the next treasury epoch. At this phase committe members reveal their random values together with a proof of their relevance with the commitment. In case some committee member failed to reveal the random value, his secret key is jointly reconstructed by other committee members and then everyone can decrypt the random value. So if a committee member has made a committment of a random value, it will be revealed even if he aborted to do it by himself, which makes impossible to manipulate the final randomness. The revealing phase is formalized at and Fig. 14.

[13] By qualified we mean a committee member that has not been disqualified during any of the previous stages (DKG, Tally)

[14] See implementation here:
https://github.com/input-output-hk/treasury-crypto/../protocol/decryption/RandomnessGenManager.scala

Entities:

Committee members $\mathcal{C} := \{C_1, \ldots, C_l\}$.

Random value commitment. Each qualified committee member $C_j,\ j \in \mathcal{J}'$ does the following:

- Generate a random value $r_j \in \mathbb{F}_p$
- Compute $R_j = \mathsf{Enc}_{\mathsf{pk}_j}(r_j; r')$, where $r' \in \mathbb{F}_p$ is some other random value needed for the ElGamal encryption scheme
- Publish R_j to the blockchain

Figure 13: Random value commitment

Entities:

Committee members $\mathcal{C} := \{C_1, \ldots, C_l\}$.

Round 1 (random values revealing): Each committee member $C_j,\ j \in \mathcal{J}''$, who submitted a random value commitment R_j, does the following:

- Decrypt committed random value: $D_j = \mathsf{Dec}'_{\mathsf{sk}_j}(R_j)$
- Compute a proof π_j that the decryption is correct (see Fig. 5 for NIZK description):

$$\pi_j \leftarrow \mathsf{NIZK}\left\{ \begin{array}{l} (\mathsf{pk}_j, R_j, D_j), (\mathsf{sk}_j) : \\ D_j = \mathsf{Dec}'_{\mathsf{sk}_j}(R_j) \ \wedge pk_j = g^{sk_j} \ \wedge \ (\mathsf{pk}_j, \mathsf{sk}_j) \in \mathcal{R}_{\mathrm{PKE}} \end{array} \right\}$$

- Publish $\{D_j, \pi_j\}$ to the blockchain

Round 2 (randomness decryption shares recovery):
Each qualified committee member $C_i,\ i \in \mathcal{J}''$ does the following:

- If there are such committee members $C_j,\ j \in \mathcal{F}''$, which submitted commitments to random value but failed to reveal them, then C_i posts to the blockchain shares $s_{j,i},\ j \in \mathcal{F}''$ of C_j's secret key obtained during the Round 1 of the DKG protocol (Figure 4).

Randomness computation. Everyone does the following:

- Reconstruct secret keys of $C_j,\ j \in \mathcal{F}''$: $sk_j := \sum_{i \in [\mathcal{J}']} \gamma_j \cdot s_{j,i}$, where $\gamma_j := \prod_{\ell \in \mathcal{J}'' \setminus \{j\}} \frac{\ell}{\ell - j}$
- Decrypt missing random values of failed committee members $C_j,\ j \in \mathcal{F}''$ as $D_j = \mathsf{Dec}'_{\mathsf{sk}_j}(R_j)$.
- After obtaining all random values D_j from all committee members $C_j,\ j \in \mathcal{J}''$, compute randomness as follows:

$$rand = H(\|_{j \in \mathcal{J}''} D_j),$$

where $\|_{j \in \mathcal{J}''} D_j$ denotes a concatanation of $D_j,\ j \in \mathcal{J}''$.

Figure 14: Randomness revealing

3 Treasury Epoch: Overall Timeline

Fig.15 summarizes the timeline of a treasury epoch. Note that the time in a blockchain setting can be represented as block periods. Each phase of the protocol takes certain amount of blocks to be executed. During these periods, corresponding transactions should be submitted (e.g., registration transaction, ballot transaction, etc.).

The time required for each phase should be evaluated experimentally to assure robust protocol execution. We expect that the voting phase takes most of the epoch time as it requires actions from all voters. Registration phase may also take substantial timeframe as it requres submitting transactions from all participants. On the other hand, all other phases, that involves only committee members, can be relatively short as they are strictly technical and we expect committee members to be well-prepared to perform these phases efficiently.

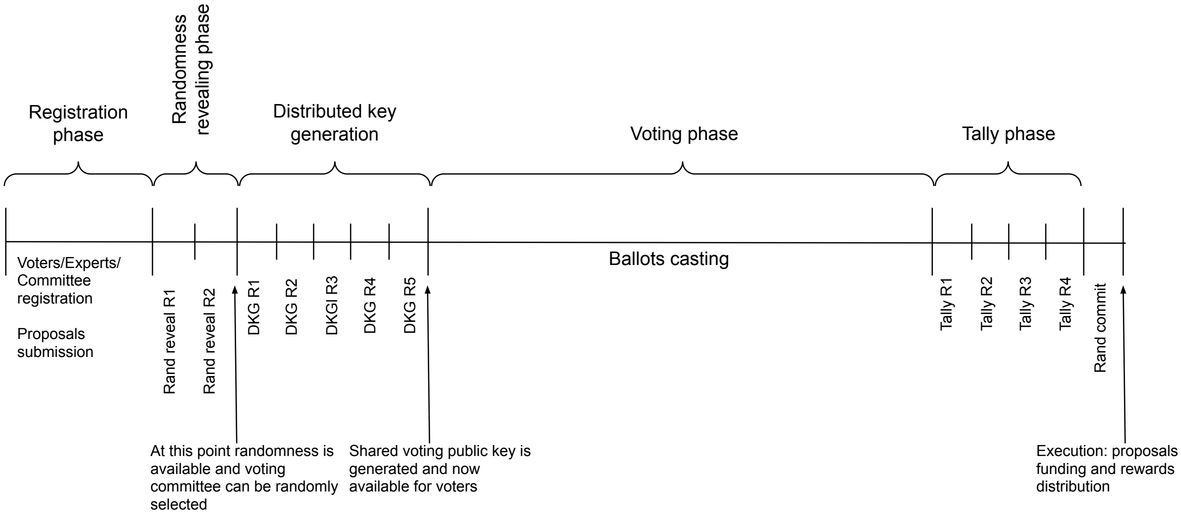
Registration phase
Randomness revealing phase
Distributed key generation
Voting phase
Tally phase
Ballots casting
Voters/Experts/
Committee registration
Proposals submission
Rand reveal R1
Rand reveal R2
DKG R1
DKG R2
DKGl R3
DKG R4
DKG R5
Tally R1
Tally R2
Tally R3
Tally R4
Rand commit
At this point randomness is available and voting committee can be randomly selected
Shared voting public key is generated and now available for voters
Execution: proposals funding and rewards distribution

Figure 15: Treasury epoch timeline

References

[1] Treasury crypto library. https://github.com/input-output-hk/treasury-crypto/.

[2] Treasurycoin prototype. https://github.com/input-output-hk/TreasuryCoin/.

[3] David Chaum and Torben P. Pedersen. Wallet databases with observers. In *CRYPTO '92*, volume 740, pages 89–105, 1993.

[4] Rosario Gennaro, Stanisław Jarecki, Hugo Krawczyk, and Tal Rabin. Secure distributed key generation for discrete-log based cryptosystems. In *EUROCRYPT '99*, pages 295–310. Springer Berlin Heidelberg, 1999.

[5] Jens Groth and Markulf Kohlweiss. *One-Out-of-Many Proofs: Or How to Leak a Secret and Spend a Coin*, pages 253–280. Springer, 2015.

[6] Bingsheng Zhang, Roman Oliynykov, and Hamed Balogun. A treasury system for cryptocurrencies: Enabling better collaborative intelligence, 2018. https://eprint.iacr.org/2018/435.pdf.

[7] Bingsheng Zhang, Roman Oliynykov, and Hamed Balogun. A treasury system for cryptocurrencies: Enabling better collaborative intelligence. ndss symposium, 2019. https://www.ndss-symposium.org/wp-content/uploads/2019/02/ndss2019_02A-2_Zhang_paper.pdf.