sciendo

Proceedings on Privacy Enhancing Technologies ; 2020 (4):297–320

Elizabeth C. Crites, Mary Maller, Sarah Meiklejohn, and Rebekah Mercer

Reputable List Curation from Decentralized Voting

Abstract: Token-curated registries (TCRs) are a mechanism by which a set of users are able to jointly curate a reputable list about real-world information. Entries in the registry may have any form, so this primitive has been proposed for use — and deployed — in a variety of decentralized applications, ranging from the simple joint creation of lists to helping to prevent the spread of misinformation online. Despite this interest, the security of this primitive is not well understood, and indeed existing constructions do not achieve strong or provable notions of security or privacy. In this paper, we provide a formal cryptographic treatment of TCRs as well as a construction that provably hides the votes cast by individual curators. Along the way, we provide a model and proof of security for an underlying voting scheme, which may be of independent interest. We also demonstrate, via an implementation and evaluation, that our construction is practical enough to be deployed even on a constrained decentralized platform like Ethereum.

DOI 10.2478/popets-2020-0074
Received 2020-02-29; revised 2020-06-15; accepted 2020-06-16.

1 Introduction

In recent years, decentralization has been viewed as an increasingly attractive alternative to the existing power structures in place in much of society, in which one party or a small set of parties are trusted — in a largely opaque manner — to make decisions that have far-reaching impact. This movement is exemplified by the rise of cryptocurrency and decentralized computing platforms like Bitcoin and Ethereum, in which everyone acts collectively to agree on the state of a ledger of transactions. While decentralization does lower the trust that must be placed in a small set of authorities, operating entirely without authorities is also problematic. For example, one could broadly attribute the rise of misinformation campaigns online to the lack of authoritative sources of information, or at least a disagreement about who these authoritative sources should be [13].

For many of the envisaged applications of decentralized platforms, it is important to have access to information about real-world events. For example, a flight insurance program, or *smart contract*, needs to know real departure times. This can be achieved via *oracles*, which are themselves smart contracts responsible for bringing external information into the system. Using an *authenticated data feed* [22–24], websites that provide real-time information can feed it into the platform in a way that ensures authenticity. Other solutions include those employed in the Augur[1] and Gnosis[2] prediction markets, which maintain that if enough users say the same thing, then what they say becomes the truth. Users are incentivized to act by a reward that is provided if their information is later accepted as truthful. These solutions have the disadvantage that they rely on the wisdom of the crowd, which can be gamed if there is incentive to misrepresent the truth and is subject to Sybil attacks if access to tokens is not controlled. The advantage is that they do not rely on authoritative external websites being willing to create custom data feeds.

One natural relative of an oracle is the idea of a *token-curated registry* [12], or TCR. In a TCR, a set of *curators*, each in possession of some tokens, are tasked with maintaining a list (or registry) of entries. Services apply to have entries included in this list, and curators decide whether or not they belong there. If all curators agree that an entry belongs, then they take no action and eventually it is included. If, on the other hand, even a single curator thinks an entry doesn't belong, it can challenge its inclusion, in which case all other curators vote to decide its fate. The curators thus act as a semi-authoritative set of entities for this particular list and, as with oracles, can be incentivized to vote truthfully via a built-in reward structure. A full description of how TCRs operate can be found in Section 3.2.

The proposed applications of TCRs are broad and range from simple uses of lists to more complex ones, such as having a consortium of news organizations iden-

Elizabeth C. Crites: University College London, email: e.crites@ucl.ac.uk
Mary Maller: Ethereum Foundation, email: mary.maller@ethereum.org
Sarah Meiklejohn: University College London and IC3, email: s.meiklejohn@ucl.ac.uk
Rebekah Mercer: O(1) Labs, email: rebekahmercer0@gmail.com

1 https://www.augur.net/
2 https://gnosis.io/

tify real images and articles in order to prevent the spread of misinformation online. In fact, this example is a reality: the New York Times is leading the News Provenance project,[3] which worked with IBM's Hyperledger Fabric to create a decentralized prototype designed to provide verifiable and user-friendly signals about the authenticity of news media online. The Civil project,[4] which allows curators to decide which content creators should be allowed in its newsroom, is backed by a TCR that is currently running on Ethereum.[5] One could also imagine using TCRs to have browser vendors jointly curate lists of valid Certificate Transparency logs, rather than the current situation in which they maintain these lists separately.[6][7] In all of these deployment scenarios, the business interests, social relationships, and potential conflicts of the participants make it essential that curation decisions are kept secret, so that parties can vote honestly without worrying about retaliation or bribery. This is especially crucial in contexts where smaller and less established companies, which are more vulnerable to this type of pressure, are taking on this coordination role. On the other hand, the fact that the set of potential curators is known means we do not have to worry about Sybil attacks.

Despite the growing interest in and deployment of TCRs, there are few existing solutions today. The solutions that do exist either reveal votes in the clear [8], which again puts curators at risk of being pressured to vote in a given direction, or rely on specialized hardware [10]. Prior to this paper, it was not known which security properties are important for TCRs, or the extent to which existing constructions satisfy these properties.

In this paper, we provide a formal cryptographic treatment of token-curated registries, including a model capturing their requirements (Section 4), a provably secure construction (Section 5), and a prototype implementation (Section 7). As the above description suggests, the core of our TCR is a voting protocol. While it might seem like a matter of just choosing an existing protocol from the voting literature, there are challenges to this approach. First, the voting protocol must have the appropriate formal cryptographic model and proof of security, or else we must provide them ourselves. Recent progress has been made in formalizing voting primitives and, in particular, modeling ballot privacy [6]. Nevertheless, almost all voting protocols operate in the presence of semi-trusted voting authorities. These parties are relied upon to take actions such as tallying the individual votes, and may need some additional capability (e.g., randomization or private state) in order to compute the tally in a privacy-preserving way. Even when authorities do not need to be trusted to achieve integrity, they often still need to be trusted to achieve privacy, as in the case of Helios [1]. In order to deploy a TCR as a smart contract operating on a decentralized platform, the contract must function as an (untrusted) authority. Given the constraints of the platform, this means it cannot maintain any private state and must operate deterministically; for voting, this means we require a protocol that allows anyone to compute the tally once everyone has voted. This property is known as *self-tallying* [19]. Moreover, all computations performed by the contract come at a high cost (in terms of the *gas* paid to execute them; see Section 3.1 for background on how Ethereum operates). We must thus use only lightweight cryptographic primitives, but still achieve provable security. (Of course, we could also operate a TCR using a platform other than a blockchain, or even on a private blockchain in which computation would not be priced as high. If our solution works on a constrained platform like Ethereum, however, it would also work here, so we design for the worst-case scenario.) We resolve these issues by borrowing several ideas from the voting literature, most notably a self-tallying protocol due to Hao et al. [16], but substantially adapt them to fit this setting. Specifically, our contributions are as follows:

– We provide a formal cryptographic model for TCRs, in terms of the two security properties that they require: *vote secrecy* and *dispute freeness*. These capture the notions that the scheme should not reveal the individual votes, and that it should be verifiable whether or not users have followed the protocol. Our definitions include a formal game-based treatment of interactions with smart contracts and of the voting mechanism inherent in TCRs, both of which may be of independent interest.
– We provide the first TCR construction that is provably secure. In particular, the security and privacy of our TCR can be proved under Decisional Diffie-Hellman (DDH) in the random oracle model, and we can run it using a transparent (i.e., public-coin) setup. Our proofs of security cover a gap in terms of the voting literature, as they imply the security

3 https://www.newsprovenanceproject.com/
4 https://civil.co/
5 https://registry.civil.co/
6 https://valid.apple.com/ct/log_list/current_log_list.json
7 https://www.gstatic.com/ct/log_list/v2/log_list.json

of the original Hao et al. [16] protocol. We are not able to prove concurrent vote secrecy for our protocol here, but are able to prove it for a subsequent version [9].

– We provide an implementation of our protocol and evaluate its performance on the Ethereum platform. We find that even on a restrictive platform like Ethereum, the voting protocol costs only 12 cents per participating curator.

2 Related Work

We are aware of two proposed TCR constructions based in industry. Consensys' PLCR ("Partial Lock Commit Reveal") protocol [8] is very efficient, as it uses a two-round commit-and-reveal approach (i.e., a first-round vote consists of a hash of a vote and a random nonce, and a second-round vote reveals this vote and nonce), but this particular solution cannot satisfy any notion of vote secrecy given that votes are revealed in the clear. The secret voting protocol due to Enigma [10] focuses on secrecy, but relies on trusted hardware (e.g., Intel SGX) to securely tally the votes, rather than allowing this to be done in the clear in an untrusted manner. It is also not clear what implications this has for a notion like dispute freeness.

Beyond constructions, Falk and Tsoukalas [17] considered the incentive mechanisms inherent in TCRs from a game-theoretic perspective, to understand whether or not the reward structure provides participants with an incentive to act truthfully. Asgaonkar and Krishnamachari [2] also consider the payoffs inherent in a TCR and the behavior of a rational potential challenger. Finally, Ito and Tanaka [18] propose incorporating curator reputation into the TCR to determine the reward that individual curators receive.

We also look more broadly at the voting literature, as the core of our token-curated registry is a voting protocol. Kiayias and Yung [19] were the first to demonstrate that the three properties we need (vote secrecy, dispute freeness, and self-tallying) could be achieved. Their protocol requires only three rounds of communication, but the computational cost per voter depends on the total number of voters. Groth [14] proposed a scheme with the same properties and a constant and low computational cost per voter, but the number of rounds is $n + 1$, where n is the number of voters. Hao, Ryan, and Zielinski [16] introduced a protocol that resolved this by requiring only two rounds and lower computational costs than those in Groth's scheme. One caveat of all of these protocols is that they are not appropriate in large-scale elections, but only in a *boardroom* setting, in which the number of voters is limited. This also fits the needs of a token-curated registry, however, in which the set of possible voters is limited to users who (1) possess a specific token and (2) have chosen to use that token to act as curators. Of these protocols, the one by Hao et al. is the best candidate for usage as a smart contract, as demonstrated by a follow-up work featuring an Ethereum-based implementation [20]. Their protocol, however, lacks a formal proof of security. Along with our enhancements that are needed to use this protocol within a TCR, this is a gap that we fill in this paper.

3 Background

3.1 Smart contracts

The first deployed cryptocurrency, Bitcoin, was introduced in January 2009. In Bitcoin, a *blockchain* structure maintains a ledger of all transactions that have ever taken place. The Bitcoin scripting language is designed to enable the atomic transfer of funds from one set of parties to another; as such, it is relatively simple and restrictive. In contrast to Bitcoin, Ethereum uses a scripting language that is (almost) Turing-complete; currently, the most common choice is Solidity. This is designed to enable *smart contracts*, which are programs that are deployed and executed on top of the Ethereum blockchain. Smart contracts accept inputs, perform computations, and maintain state in a globally visible way; they must also operate deterministically so that every node can agree on a contract's state. The only limitation in terms of the programs Solidity produces is their complexity, as every operation consumes a certain amount of *gas*. This is a subcurrency priced in ether, the native currency of the Ethereum blockchain, and acts to limit the computation or storage that an individual contract can use, as this computation and storage must be replicated by every node in the network. As of this writing, each block produced in Ethereum has a gas limit of 10 million.

In their most simplified form, Ethereum transactions contain a destination address, a signature σ authorizing the transaction with respect to the public key pk of the sender, a gas limit, a gas price, an amount amt in ether, and an optional data field data. The destina-

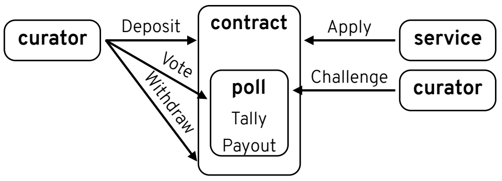

Fig. 1. The participants in a token-curated registry (TCR) and the algorithms they run. Users become curators by depositing coins in a smart contract, and can later withdraw those coins when they no longer wish to act in this role. Services can apply to have entries added to the registry, and curators can challenge these additions as desired. This creates a poll, in which other curators can cast their votes to decide whether or not the entry should be included. Once the poll is closed, the contract can tally the results and reward the curators who voted with the majority decision.

tion address can be either *externally owned*, meaning it is controlled by another user, or it can be a contract address, which points to the code of some smart contract and its associated storage. If the destination address is externally owned, the effect of the transaction is to transfer amt in ether to the user in control of this address. If the destination address is a contract, the effect is that the contract code is executed on any inputs specified in the data field data. This may result in updates to the contract state and/or the creation of additional transactions, subject to the specified gas limit (meaning if the user has not paid enough gas for these operations the transaction may fail and have no effect). Miners may also choose to reject the transaction if it does not offer a high enough gas price. In what follows, we assume the participant has always paid enough gas, so omit the limit and price, but revisit their role in Section 7.

3.2 Token-curated registries

A token-curated registry, or TCR for short, is a mechanism designed to allow people in possession of some relevant tokens, also known as *curators*, to collectively make decisions about which types of entries belong on a given list or registry. A TCR has two main types of participants: *services*, who apply to have entries added to the registry, and curators, who decide on the content that can be added. The interplay between these participants can be seen in Figure 1. We provide a formal treatment of the algorithms they run in Section 4, but cover them at a high level here.

When a service applies to have its entry added, it puts down some deposit, and its application is registered in the system. Curators then have some amount of time to decide if they are happy with the entry being added to the registry. If they are, they do nothing, and after the elapsed time the entry is added and the deposit is returned to the service. If, on the other hand, one curator is unhappy with this entry, they can challenge its addition. This means they also place some deposit and open a poll, which is essentially a referendum on whether or not this entry should be added. (Some TCR models consider broader voting options, but for simplicity we stick with with a simple binary approach.) Other curators have some time available to vote, and once this time has passed the results of the vote are tallied and the winner is determined according to the rules of the poll (e.g., a simple majority). If the vote is on the side of the service, they get back their deposit and some portion of the challenger's deposit as well. The remainder of the challenger's deposit is split between the curators who voted with the majority; i.e., voted in favor of the service. If instead the vote is against the service, the situation is reversed: the challenger gets back their deposit and some portion of the service's deposit, and the remainder of the service's deposit is split between the curators who voted on the side of the challenger.

4 Definitions

4.1 Preliminaries

If x is a binary string then $|x|$ denotes its bit length. If S is a finite set then $|S|$ denotes its size and $x \xleftarrow{\$} S$ denotes sampling a member uniformly from S and assigning it to x. We use $\lambda \in \mathbb{N}$ to denote the security parameter and 1^λ to denote its unary representation.

Algorithms are randomized unless explicitly noted otherwise. "PT" stands for "probabilistic polynomial time." We use $\vec{y} \leftarrow A(\vec{x}; r)$ to denote running algorithm A on inputs $\vec{x}$ and randomness r and assigning its output to $\vec{y}$. We use $\vec{y} \xleftarrow{\$} A(\vec{x})$ to denote $y \leftarrow A(x; r)$ for uniformly random r. The set of values that have non-zero probability of being output by A on input $\vec{x}$ is denoted by $[A(\vec{x})]$. For two functions $f, g : \mathbb{N} \to [0, 1]$, $f(\lambda) \approx g(\lambda)$ denotes $|f(\lambda) - g(\lambda)| = \lambda^{-\omega(1)}$. We use code-based games in security definitions and proofs [4]. A game $\mathsf{Sec}_{\mathcal{A}}(\lambda)$, played with respect to a security notion Sec and adversary $\mathcal{A}$, has a MAIN procedure whose output is the output of the game. The notation $\Pr[\mathsf{Sec}_{\mathcal{A}}(\lambda)]$ denotes the probability that this output is 1. We denote relations using the notation

$R = \{(\phi, w) : \langle\text{properties that } (\phi, w) \text{ satisfy}\rangle\}$ where ϕ is the public instance and w is the private witness.

The security of our TCR relies on the Decisional Diffie-Hellman (DDH) assumption, which states that (g, g^x, g^y, g^{xy}) is indistinguishable from (g, g^x, g^y, g^z) for $x, y, z \xleftarrow{\$} \mathbb{F}$, where $\mathbb{F}$ is a finite field. It also relies on the security of a sigma protocol ($\mathsf{Prove}, \mathsf{Verify}$); i.e., a three-round interactive protocol. We then make this non-interactive using the Fiat-Shamir heuristic [11], which introduces a reliance on the random oracle model. We require the sigma protocol to satisfy two properties: special honest verifier zero-knowledge (SHVZK) and 2-special soundness [15]. This means the corresponding non-interactive proof satisfies zero knowledge and knowledge soundness. All of these properties are standard, but we include their definitions for completeness in Appendix A.

4.2 Smart contracts

To model interactions with smart contracts formally, we consider that every algorithm Alg run by a participant in the network outputs a transaction tx. This means that at some point the participant runs an algorithm $\mathsf{tx} \xleftarrow{\$} \mathsf{FormTx}(\mathsf{sk}, \mathsf{rcpt}, \mathsf{amt}, \mathsf{data})$ that outputs a transaction signed using sk and destined for the recipient rcpt, and carrying amt in ether and some data data to be provided to the contract. If the sender and recipient are implicit from the context, then we use the shorthand $\mathsf{tx} \xleftarrow{\$} \mathsf{FormTx}(\mathsf{amt}, \mathsf{data})$. There is then a corresponding function $\mathsf{Process\_Alg}$ in the smart contract that takes this transaction as input, verifies that it has the correct form and is properly signed, and (if so) uses it to update the state of the contract, in terms of its functions and associated storage. We must also consider how an adversary $\mathcal{A}$ can interact with smart contracts inside of a security game, given that the adversary can interact with the contract itself, see all of the interactions that honest participants have with it, and see all of its internal state and function calls. We model this by providing $\mathcal{A}$ with access to three classes of oracles, which abstractly behave as follows:

- AP.Alg allows the adversary to interact with the contract via its own participants, according to some specified algorithm. This oracle uses $\mathsf{Process\_Alg}$ to process the adversary's input, which is meant to be the output of running Alg, on behalf of the contract.
- HP.Alg allows the adversary to instruct some honest participant i to interact with the contract, according to some specified algorithm. This means the adversary provides any necessary inputs, and the oracle then runs Alg for participant i according to these inputs and uses $\mathsf{Process\_Alg}$ to process the corresponding output on behalf of the contract.
- CP allows the adversary to view the entire state of the contract. We do not use this oracle explicitly in our games below, since the adversary can see all information about the contract whenever it wants, but leave it there as a reminder of this ability.

In our definitions below, we allow the adversary to interact with many *sessions* of the contract concurrently; i.e., many different configurations. We denote the number of honest participants by n, the contract session index by j, and the participant index by i; when querying the oracles defined above, the adversary must always specify the session j. While both game specifications allow for multiple sessions, our current proof of vote secrecy requires $j = 1$. In a subsequent version of our protocol, however, we are able to prove concurrent vote secrecy [9]. We assume all participants, including the contract, are stateful, and denote their state by state. For the sake of readability and succinctness, we ignore the potential for the adversary to corrupt honest participants; instead, we allow it to control arbitrarily many adversarial participants but to only observe honest participants. We leave the ability to handle corruptions, for both our model and our construction, as an interesting open question.

4.3 Token-curated registries

Formally, we consider a token-curated registry (TCR) to be defined by several algorithms, which correspond to the ones in Figure 1. First, we define three algorithms associated with voting.

- $\mathsf{tx}_{\mathsf{vote1}} \xleftarrow{\$} \mathsf{Vote1}(\mathsf{contract}, \mathsf{poll}, \mathsf{wgt}, \mathsf{vote})$ is run by a curator wishing to contribute some weight wgt and vote vote to some poll poll contained in the contract $\mathsf{contract}$.
- $\mathsf{tx}_{\mathsf{vote2}} \xleftarrow{\$} \mathsf{Vote2}(\mathsf{contract}, \mathsf{poll})$ is run by a curator in the second round of voting for poll in $\mathsf{contract}$.
- $\mathsf{Tally}(\mathsf{poll}, \mathsf{tally})$ is run by the contract in order to tally the results of the vote in the poll poll. To be more efficient, it optionally takes in a proposed tally tally (computed, for example, by one of the curators), which it can then verify is the correct one.

The main reason that our voting protocol proceeds in rounds is that we need a fixed set of voters in order to achieve the *self-tallying* property that says that the tally can be computed by any third party (in our case, by the contract) once everyone has voted. In our construction, voters commit to their vote and "register" their interest in voting in the first round. In the second round, once the set of registered voters is fixed, voters can vote again, this time using the relevant information from the first round to form a self-tallying vote. The tally can then be computed using these votes cast in the second round, while the votes cast in the first round (which the voters prove are the same) can be used to pay voters on the winning side. More general constructions could be modelled by having an interactive Vote protocol (with possibly more than two rounds). We also define algorithms associated with the TCR more broadly.

- $\mathsf{tx}_{\mathsf{dep}} \xleftarrow{\$} \mathsf{Deposit}(\mathsf{contract}, \mathsf{amt})$ is run by a user wishing to become a curator in the TCR by creating some initial deposit of tokens amt.
- $\mathsf{tx}_{\mathsf{app}} \xleftarrow{\$} \mathsf{Apply}(\mathsf{contract}, \mathsf{entry})$ is run by a service wishing to add an entry entry to the registry.
- $\mathsf{tx}_{\mathsf{chal}} \xleftarrow{\$} \mathsf{Challenge}(\mathsf{contract}, \mathsf{entry})$ is run by a curator wishing to challenge the addition of the entry entry to the registry.
- $\mathsf{tx}_{\mathsf{with}} \xleftarrow{\$} \mathsf{Withdraw}(\mathsf{contract}, \mathsf{amt})$ is run by a curator wishing to withdraw some amount amt of their deposited tokens.
- $\mathsf{Payout}(\mathsf{poll})$ is run by the contract to pay the curators who voted according to the poll outcome.

4.3.1 Vote secrecy

Vote secrecy says that an adversary cannot learn the contents of a user's vote, beyond what it can infer from their weighting. A formal vote secrecy game is in Figure 2, assuming for notational simplicity that there is only one poll poll so it does not need to be specified. Intuitively, it proceeds as follows. The adversary is given a set of contract configurations $\mathsf{contracts}$, which it is free to interact with concurrently, and a set of the public keys $\{\mathsf{pk}_i\}_i$ belonging to honest participants (line 4). All contracts start in an initial state, meaning their storage fields are empty, with only the parameters initialized.

The adversary is then free to have both its own and honest participants deposit tokens to become curators; it is also free to create arbitrary applications and challenges, and have its own and honest participants vote. The real contract has timers indicating when it should move from the first to the second round of voting, but here we do this manually: the first time the adversary calls either $\mathrm{AP}.\mathsf{Vote2}$ or $\mathrm{HP}.\mathsf{Vote2}$ the voting flag is set to be 1, to signal that the set of voters is fixed and the second round has started (lines 6 and 13).

The main question is how honest voters should vote. If they all vote for the secret bit b (line 2), then the adversary is clearly able to guess b and win the game (line 5), since it can see in the final tally if everyone has voted for 0 or 1. To prevent this trivial type of victory, we thus ensure that the final tally is the same regardless of the bit b, following Benaloh [5]. To do this, we have the adversary provide its own bit $b_{\mathcal{A}}$ as input to $\mathrm{HP}.\mathsf{Vote1}$, which signals whether it wants the voter to vote "for" ($b_{\mathcal{A}} = 1$) or "against" ($b_{\mathcal{A}} = 0$) the secret bit b. This is the same as voting for the bit $b_{\mathcal{A}}$ EQ b, which is what the voters do (line 10). We then keep track of how many times the adversary has used $b_{\mathcal{A}} = 0$ and $b_{\mathcal{A}} = 1$, using a variable $\mathsf{vote\_count}$ (lines 8 and 9). If it has used them an equal number of times, meaning $\mathsf{vote\_count} = 0$, then we have the same number of votes for b and $\neg b$, so the outcome is the same regardless of b. If they are not equal at the start of the voting round, then $\mathcal{A}$ automatically loses the game (line 12).

Finally, to prevent another trivial way for the adversary to learn how people voted, we prevent it from instructing honest participants to withdraw (line 16), as this reveals their balance. This prevents the adversary from instructing a participant to deposit a certain amount of coins, having them vote once, and then instructing them to withdraw their coins and seeing if the amount is the same (indicating they voted on the losing side) or is more than what they deposited (indicating that they voted on the winning side). In practice, this means that participants would perhaps need to vote some minimum number of times before withdrawing, in order to prevent these types of inference attacks.

Definition 4.1. *Define* $\mathbf{Adv}^{secrecy}_{\mathcal{A}}(\lambda) = 2\Pr[\mathsf{G}^{secrecy}_{\mathcal{A}}(\lambda)] - 1$*, where this game is defined as in Figure 2 (with the descriptions of all calls in which the oracle honestly follows the protocol omitted). Then the TCR satisfies* vote secrecy *if for all PT adversaries* $\mathcal{A}$ *there exists a negligible function* $\nu(\cdot)$ *such that* $\mathbf{Adv}^{secrecy}_{\mathcal{A}}(\lambda) < \nu(\lambda)$*.*

```
$\underline{\text{MAIN } \mathsf{G}^{\mathbf{secrecy}}_{\mathcal{A}}(\lambda)}$
$\mathsf{vote\_count} \leftarrow \vec{0}$
$b \xleftarrow{\$} \{0,1\}$
$(\mathsf{pk}_i, \mathsf{sk}_i) \xleftarrow{\$} \mathsf{KeyGen}(1^\lambda)\ \forall i \in [n]$
$b' \xleftarrow{\$} \mathcal{A}^{\mathrm{AP,HP,CP}}(1^\lambda, \mathsf{contracts}, \{\mathsf{pk}_i\}_i)$
return $(b' = b)$

$\underline{\mathrm{AP}.\mathsf{Vote2}(j, \mathsf{tx_{vote2}})}$
if $(\mathsf{contracts}[j].\mathsf{vote\_flag} = 0)$
  $\mathsf{contracts}[j].\mathsf{vote\_flag} \leftarrow 1$
$\mathsf{contracts}[j].\mathsf{Process\_Vote2}(\mathsf{tx_{vote2}})$

$\underline{\mathrm{HP}.\mathsf{Vote1}(j, i, b_{\mathcal{A}})}$
if $(b_{\mathcal{A}} = 0)$ $\mathsf{vote\_count}[j]$ $-= 1$
else $\mathsf{vote\_count}[j]$ $+= 1$
$\mathsf{tx_{vote1}} \xleftarrow{\$} \mathsf{Vote1}(\mathsf{contracts}[j], b_{\mathcal{A}}\ \mathbf{EQ}\ b)$
$\mathsf{contracts}[j].\mathsf{Process\_Vote1}(\mathsf{tx_{vote1}})$

$\underline{\mathrm{HP}.\mathsf{Vote2}(j, i)}$
if $(\mathsf{vote\_count}[j] \neq 0)$ return 0
if $(\mathsf{contracts}[j].\mathsf{vote\_flag} = 0)$
  $\mathsf{contracts}[j].\mathsf{vote\_flag} \leftarrow 1$
$\mathsf{tx_{vote2}} \xleftarrow{\$} \mathsf{Vote2}(\mathsf{contracts}[j])$
$\mathsf{contracts}[j].\mathsf{Process\_Vote2}(\mathsf{tx_{vote2}})$

$\underline{\mathrm{HP}.\mathsf{Withdraw}(\cdot,\cdot)}$
return $\perp$
```

Fig. 2. The TCR vote secrecy game.

```
    $\underline{\text{MAIN } \mathsf{G}^{\mathbf{dispute}}_{\mathcal{A}}(\lambda)}$
1   $\mathsf{tally_1}, \mathsf{tally_2} \leftarrow \vec{0}$
2   $(\mathsf{pk}_i, \mathsf{sk}_i) \xleftarrow{\$} \mathsf{KeyGen}(1^\lambda)\ \forall i \in [n]$
3   $\mathsf{done} \xleftarrow{\$} \mathcal{A}^{\mathrm{AP,HP,CP}}(1^\lambda, \mathsf{contracts}, \{\mathsf{pk}_i\}_i)$
4   $b_{\mathsf{tally}}[j] \leftarrow (\mathsf{tally_1}[j] \neq \mathsf{tally_2}[j]$
      $\vee\ \mathsf{tally_1}[j], \mathsf{tally_2}[j] \neq \mathsf{contracts}[j].\mathsf{tally})\ \forall j$
5   $b_{\mathsf{done}}[j] \leftarrow (\mathsf{contracts}[j].\mathsf{outcome} \neq \perp)\ \forall j$
6   return $(\exists j\ :\ b_{\mathsf{done}}[j] \wedge b_{\mathsf{tally}}[j])$

    $\underline{\mathrm{AP}.\mathsf{Vote1}(j, \mathsf{tx_{vote1}})}$
7   $\mathsf{contracts}[j].\mathsf{Process\_Vote1}(\mathsf{tx_{vote1}})$
8   $V \leftarrow \mathsf{Ext}(\mathsf{tx_{vote1}})$
9   $\mathsf{tally_1}[j] \leftarrow \mathsf{tally_1}[j] \oplus V$

    $\underline{\mathrm{AP}.\mathsf{Vote2}(j, \mathsf{tx_{vote2}})}$
10  if $(\mathsf{contracts}[j].\mathsf{vote\_flag} = 0)$
      $\mathsf{contracts}[j].\mathsf{vote\_flag} \leftarrow 1$
11  $\mathsf{contracts}[j].\mathsf{Process\_Vote2}(\mathsf{tx_{vote2}})$
12  $V \leftarrow \mathsf{Ext}(\mathsf{tx_{vote2}})$
13  $\mathsf{tally_2}[j] \leftarrow \mathsf{tally_2}[j] \oplus V$

    $\underline{\mathrm{HP}.\mathsf{Vote1}(j, i, b_{\mathcal{A}})}$
14  $\mathsf{vote} \leftarrow b_{\mathcal{A}}\ \mathbf{EQ}\ b$
15  $\mathsf{tx_{vote1}} \xleftarrow{\$} \mathsf{Vote1}(\mathsf{contracts}[j], \mathsf{vote})$
16  $\mathsf{contracts}[j].\mathsf{Process\_Vote1}(\mathsf{tx_{vote1}})$
17  $\mathsf{tally_1}[j] \leftarrow \mathsf{tally_1}[j] \oplus f(\mathsf{vote})$
18  $\mathsf{tally_2}[j] \leftarrow \mathsf{tally_2}[j] \oplus f(\mathsf{vote})$
```

Fig. 3. The TCR dispute freeness game.

4.3.2 Dispute freeness

Dispute freeness says that an adversary cannot misbehave within the protocol without public detection; i.e., it is publicly verifiable whether or not everyone followed the protocol. Unlike in a traditional voting scenario, it is already publicly verifiable whether or not the contract (which acts as the election official) follows the protocol, since its code and state transitions are globally visible. We thus need to consider only whether or not the individual curators behave. This means considering two types of misbehavior: one in which the adversary tries to change its vote halfway through the voting protocol (so between Vote1 and Vote2), and one in which it tries to bias the outcome of the vote by voting for something other than 0 or 1. A formal dispute freeness game is in Figure 3 (again, assuming for notational simplicity that poll does not need to be given as input). Intuitively, it proceeds as follows. As in the vote secrecy game, the adversary is given a set of initial contract configurations contracts and the public keys $\{\mathsf{pk}_i\}_i$ belonging to honest participants. It is then free to interact with the contract via AP and HP, and the game keeps track of the voting round in the same way as the vote secrecy game.

In order to detect misbehavior, the game keeps track of two tallies: tally_1 based on the votes indicated in the first round of voting, and tally_2 based on the votes in the second round. The adversary then wins the game if for a vote that completes successfully, meaning it has a defined outcome (line 5), either of these tallies is different from the official tally ($\mathsf{contracts}[j].\mathsf{tally}$) kept by the contract, or if they are different from each other (line 4). Keeping track of the votes in both the first and second round is easy for honest participants, since these are known so can just be added to the tallies (lines 17 and 18), although we allow the tally to incorporate a function of the vote $f(\mathsf{vote})$ rather than just the vote itself. (In our construction, for example, this function is $f(x) = g^x$.) For adversarial participants, we must rely on the ability to *extract* the intended vote in each round. This means we assume the existence of an extractor Ext that, given the transaction provided by the adversary, can output $V = f(\mathsf{vote})$, where vote is the vote intended by the adversary in that round (lines 8-9 and 12-13).

Definition 4.2. *Define* $\mathbf{Adv}^{dispute}_{\mathcal{A},\mathsf{Ext},f}(\lambda) = \Pr[\mathsf{G}^{dispute}_{\mathcal{A},\mathsf{Ext},f}(\lambda)]$*, where this game is defined as in Figure 3 with respect to a function* $f(\cdot)$ *(with the descriptions of all calls in which the oracle honestly follows the protocol omitted, and* HP.Vote2 *behaving as it does in the game in Definition 4.1). Then the TCR satisfies* dispute freeness *if there exists an extractor* Ext *such that for all PT adversaries* $\mathcal{A}$ *there exists a negligible function* $\nu(\cdot)$ *such that* $\mathbf{Adv}^{dispute}_{\mathcal{A},\mathsf{Ext},f}(\lambda) < \nu(\lambda)$*.*

While vote secrecy and dispute freeness are the only properties we define and prove formally for our TCR, there are several other properties (e.g., coercion resistance) that may be desirable or even necessary for such a system. We discuss these properties in Section 5.3, along with how we can provide financial disincentives for these additional types of misbehavior.

5 Construction

5.1 Design overview

At a high level, our token-curated registry operates as a single smart contract; one can imagine the structure being identical to that of a PLCR (Partial Lock Commit Reveal) contract [8], but with our Vote1 and Vote2 replacing their respective commit and reveal rounds. Users become curators by depositing some number of tokens into the contract, which creates a commitment C and makes them available for challenging and voting. As curators earn tokens, they can update the balance in this commitment.

The voting protocol is the core of our token-curated registry, and proceeds in two rounds. In the first round, users "register" their interest in participating in the vote by providing a registration key $c_0 = g_0^x$ and placing a deposit. They also commit to their intended vote by forming a commitment c_1 that uses x as randomness. In addition to making sure that voters can't change their minds, this commitment c_1 is also useful in allowing the contract to later pay the curator for their vote.

In the second round, the set of voters is now fixed as the set of registered curators from the first round. We follow Hao et al. [16] in having each voter i compute a base Y_i that is a combination of the registration keys of other voters. They then commit to their vote again, this time using Y_i as a base, to form a value c_2. (The original Hao et al. protocol involves sending c_0 in the first round and c_2 in the second round, along with proofs that these are correctly formed.) These values are formulated so that $\prod_i Y_i^{x_i} = 1$, which means $\prod_i c_{i,2} = g_1^{\sum_i \mathsf{vote}_i}$ (in the unweighted case), where $c_{i,2}$ denotes the c_2 component of the i-th voter's vote. This makes the voting protocol *self-tallying*, since with this value the contract can compute the discrete logarithm by brute force to get the tally (which is efficient for a relatively small number of voters). Voters also provide an extra pair (c_3, c_4), which enables us to prove dispute freeness without rewinding, and a proof that all of these values have been formed correctly. If this proof is valid, the contract returns the deposit sent in the first round. If the proof is invalid or a voter doesn't send a second-round vote, the contract keeps the deposit as a form of punishment, since it cannot complete the vote.

Once the contract computes the tally, it determines the outcome of the vote according to the built-in rules. It can easily send the service and challenger deposits to the right places, as these are public, so the only question left is how it pays the voters who were in the majority. In particular, say the goal is to pay one token if $\mathsf{vote} = \mathsf{outcome}$, and 0 otherwise. To do this without knowing vote, we *arithmetize* this expression: for two boolean values b_1 and b_2, the boolean expression $(b_1 = b_2)$ can be represented numerically as $1 - b_1 - b_2 + 2b_1b_2$. For vote and outcome, this means we need to form $1 - \mathsf{vote} - \mathsf{outcome} + 2 \cdot \mathsf{vote} \cdot \mathsf{outcome}$. Since outcome is known, we can homomorphically form a commitment to this expression using a commitment to vote given by the curator, and then fold the result into C; this adds 1 to the balance in C if the expression evaluates to 1 (meaning $\mathsf{vote} = \mathsf{outcome}$) and 0 otherwise, as desired. While this would work with either c_1 or c_2, it is important to use c_1 for this purpose, as the base Y has a discrete logarithm that is unknown to the curator (and indeed unknown to anyone unless all voters collude) so they would no longer be able to update the commitment.

5.2 System design

We describe the algorithms that comprise our TCR below, in terms of the different operations that are required. As they do not involve any significant cryptographic operations, we omit the formal descriptions of Apply and Challenge. We can think of Apply as placing a deposit and (if the deposit is high enough) adding entry to a list of potential registry entries and starting a timer indicating how long the curators have to challenge its inclusion. We can think of Challenge as placing a deposit, opening a new poll to allow other curators to vote, and starting a timer indicating how long they have to do so.

We use one extra algorithm in addition to the ones given in our model, $\mathsf{tx}_{\mathsf{up}} \xleftarrow{\$} \mathsf{Update}()$, which is used to update a lower bound stored by the contract on the number of tokens a curator has. The only effect this has on our model is that we would want to replace Withdraw with Update in the vote secrecy game (Figure 2) as the one algorithm that the adversary can't query honest participants on, as it might reveal a curator's success in voting (whereas running Withdraw now reveals no information). A formal specification of all of the user-facing algorithms can be seen in Figure 4 and of the contract-only algorithms in Figure 5.

TCR contract setup. A contract designed to support a TCR must maintain several fields. In this section, we limit ourselves to the ones that are necessary for the cryptographic operations of the contract, meaning we ignore elements like timers. The contract is initialized with randomly generated fixed generators g_0, g_1, h_0, and h_1, and it is assumed that no adversary knows a discrete logarithm relation between them and that they are in a group in which DDH holds. This means that our TCR operates with a transparent (i.e., public-coin) setup. It is also initialized with empty maps `curators` and `voters`, which respectively keep track of all curators and all voters within a given poll. For ease of exposition, we consider a contract with only a single poll; one with multiple polls would still have only one `curators` map but one `voters` map per poll.

Joining the TCR. In order to become a curator, a user must first deposit tokens into the TCR contract. This means running the Deposit algorithm seen in Figure 4. The amount amt that they deposit determines their *weight*, which they commit to (and prove that they've committed to, using π_{wgt}) in C. They prove this using the relation R_{wgt}:

$$R_{\mathsf{wgt}} = \left(\ ((C, \mathsf{wgt}), r) : \quad C = g_1^{\mathsf{wgt}} h_0^r \ \right)$$

We describe a zero-knowledge argument of knowledge for R_{wgt} in Section 6.

To process this, the contract first uses π_{wgt} to check that C really is a commitment to the weight. If it is satisfied, it registers the user (using $\mathsf{tx}_{\mathsf{dep}}[\mathsf{pk}]$, the public key used in the transaction) in `curators`, with associated commitment C and weight wgt. The initial value in the commitment is the initial amount sent, which is public, but as the curator votes the balance may change, so it must be stored in committed form to avoid revealing information about their votes. Later, the stored value wgt thus serves as a lower bound on the actual balance stored in C.

Updating weights. Initially, since the amount is sent on-chain and thus publicly known, the weight of each participant is also known. Curators may lose tokens only by unsuccessfully challenging an entry, but this is done in a public way so this lower bound can be updated by the contract itself. The contract is unaware, however, of how many tokens curators gain, since as we see below this is determined based on whether or not their hidden votes are the same as the majority. This means the contract may need to be told a new lower bound from time to time, as specified in Figure 4. A curator does this in the same way as when they run Deposit: they simply tell the contract their current number of tokens amt, and prove that this is the value contained in C. The contract can then check that the proof verifies. If so, it increases the stored wgt value to amt. This approach may reveal information about the curator's success in voting, but it has the advantage that it uses a simple proof of knowledge, rather than general range proofs (which are much more expensive to verify). We thus would want to ensure that curators cannot update this lower bound too frequently (e.g., if they update after every single vote then they reveal how they voted every time), by making sure that they update only after they have voted some fixed number of times.

Withdrawing tokens. At some point, a participant may wish to stop acting as a curator, or to withdraw some portion of their tokens. To do this, they simply send the amount they want to withdraw to the contract, as specified in Figure 4. If the contract can see this is greater than or equal to the lower bound wgt, it sends them this amount and decreases their token store (both in terms of their committed tokens and their lower bound). If this token store goes to zero, the contract could additionally remove this curator from the list.

Voting. At some point, a service runs Apply for some entry. If every curator is happy to see entry in the registry, nothing happens and after some amount of time the entry is added. If instead a curator runs Challenge, this opens up the chance for other curators to vote on whether or not they think entry belongs in the registry.

To vote, curators first use Vote1, as specified in Figure 4. This means picking a random x, and forming g_0^x and a commitment to their vote vote using x as ran-

```
Deposit(amt)
$r \xleftarrow{\$} \mathbb{F}$; $C \leftarrow g_1^{\mathsf{amt}} h_0^r$
$\pi_{\mathsf{wgt}} \leftarrow \mathsf{Prove}(R_{\mathsf{wgt}}, (C, \mathsf{amt}), r)$
return $\mathsf{FormTx}(\mathsf{amt}, (C, \pi_{\mathsf{wgt}}))$
```

```
Process_Deposit(tx_dep)
$(C, \pi_{\mathsf{wgt}}) \leftarrow \mathsf{tx}_{\mathsf{dep}}[\mathsf{data}]$; $\mathsf{pk} \leftarrow \mathsf{tx}_{\mathsf{dep}}[\mathsf{pk}]$
$b_{\mathsf{wgt}} \leftarrow \mathsf{Verify}(R_{\mathsf{wgt}}, (C, \mathsf{tx}_{\mathsf{dep}}[\mathsf{amt}]), \pi_{\mathsf{wgt}})$
if ($b_{\mathsf{wgt}}$)
    add $\mathsf{pk} \mapsto (C, \mathsf{tx}_{\mathsf{dep}}[\mathsf{amt}])$ to curators
```

```
Update()
$\pi_{\mathsf{wgt}} \leftarrow \mathsf{Prove}(R_{\mathsf{wgt}}, (C, \mathsf{amt}), r)$
return $\mathsf{FormTx}(0, (\mathsf{amt}, \pi_{\mathsf{wgt}}))$
```

```
Process_Update(tx_up)
$(\mathsf{amt}, \pi_{\mathsf{wgt}}) \leftarrow \mathsf{tx}_{\mathsf{up}}[\mathsf{data}]$; $\mathsf{pk} \leftarrow \mathsf{tx}_{\mathsf{up}}[\mathsf{pk}]$
$(C, \mathsf{wgt}) \leftarrow$ curators[pk]
$b_{\mathsf{wgt}} \leftarrow \mathsf{Verify}(R_{\mathsf{wgt}}, (C, \mathsf{amt}), \pi_{\mathsf{wgt}})$
if ($b_{\mathsf{wgt}}$) $\mathsf{wgt} \leftarrow \mathsf{amt}$
```

```
Withdraw(amt)
return $\mathsf{FormTx}(0, \mathsf{amt})$
```

```
Process_Withdraw(tx_with)
$\mathsf{amt} \leftarrow \mathsf{tx}_{\mathsf{with}}[\mathsf{data}]$; $\mathsf{pk} \leftarrow \mathsf{tx}_{\mathsf{with}}[\mathsf{pk}]$
$(C, \mathsf{wgt}) \leftarrow$ curators[pk]
if ($\mathsf{amt} \leq \mathsf{wgt}$)
    $C \leftarrow C \cdot g_1^{-\mathsf{amt}}$; $\mathsf{wgt} \leftarrow \mathsf{wgt} - \mathsf{amt}$
    update curators[pk] $\leftarrow (C, \mathsf{wgt})$
    send amt to pk
```

```
Vote1(vote, wgt)
$x \xleftarrow{\$} \mathbb{F}$; $(c_0, c_1) \leftarrow (g_0^x, g_1^{\mathsf{vote}} h_0^x)$
$\pi_1 \leftarrow \mathsf{Prove}(R_{\mathsf{Vote1}}, c_0, x)$
add $(\mathsf{vote}, x, c_0, c_1)$ to state
return $\mathsf{FormTx}(\mathsf{amt}_{\mathsf{dep}}, (c_0, c_1, \pi_1, \mathsf{wgt}))$
```

```
Process_Vote1(tx_vote1)
$(c_0, c_1, \pi_1, \mathsf{wgt}) \leftarrow \mathsf{tx}_{\mathsf{vote1}}[\mathsf{data}]$; $\mathsf{pk} \leftarrow \mathsf{tx}_{\mathsf{vote1}}[\mathsf{pk}]$
$b_1 \leftarrow \mathsf{Verify}(R_{\mathsf{Vote1}}, c_0, \pi_1)$
$b \leftarrow (\mathsf{wgt} \leq$ curators[pk][wgt]$)$
if ($b \wedge b_1$)
    add $(c_0, c_1, \mathsf{wgt})$ to voters[pk][data]
```

```
Vote2(j)
$(c_{i,0}, c_{i,1}, \mathsf{wgt}_i) \leftarrow$ voters[pk_i][data] $\forall i$
$s \xleftarrow{\$} \mathbb{F}$; $Y \leftarrow \prod_{0 \leq i < j, j < k \leq m} c_{i,0} c_{k,0}^{-1}$
$(\mathsf{vote}, x, c_0, c_1) \leftarrow$ state
$c_2 \leftarrow g_1^{\mathsf{vote}} Y^x$
$(c_3, c_4) \leftarrow (g_0^s, (h_0^{-1} Y)^x h_1^s)$
$\pi_2 \xleftarrow{\$} \mathsf{Prove}(R_{\mathsf{Vote2}}, (Y, \{c_i\}_{i=0}^4), (\mathsf{vote}, x, s))$
return $\mathsf{FormTx}(0, (c_2, c_3, c_4, \pi_2))$
```

```
Process_Vote2(j, tx_vote2)
$(c_2, c_3, c_4, \pi_2) \leftarrow \mathsf{tx}_{\mathsf{vote2}}[\mathsf{data}]$; $\mathsf{pk} \leftarrow \mathsf{tx}_{\mathsf{vote2}}[\mathsf{pk}]$
$(c_{i,0}, c_{i,1}, \mathsf{wgt}_i) \leftarrow$ voters[pk_i][data] $\forall i$
$Y \leftarrow \prod_{0 \leq i < j, j < k \leq m} c_{i,0} c_{k,0}^{-1}$
$(c_0, c_1, \mathsf{wgt}) \leftarrow$ voters[pk][data]
$b_2 \leftarrow \mathsf{Verify}(R_{\mathsf{Vote2}}, (Y, \{c_i\}_{i=0}^4), \pi_2)$
if ($b_2$)
    tallyG $\leftarrow$ tallyG $\cdot\, c_2^{\mathsf{wgt}}$
    send $\mathsf{amt}_{\mathsf{dep}}$ to pk
```

Fig. 4. The core user-facing cryptographic algorithms that comprise the TCR, in terms of the algorithm run by the user (on the left-hand side) and the processing of the output of this algorithm run by the contract (on the right-hand side).

domness. (The pair (c_0, c_1) has the form of an ElGamal ciphertext, but no one knows the discrete logarithm of h_0 with respect to g_0 so no one can decrypt it.) The curator also demonstrates knowledge of the logarithm of c_0; i.e., they provide a proof for the relation R_{Vote1}:

$$R_{\mathsf{Vote1}} = \{\ (c_0, x) : \quad c_0 = g_0^x\ \}$$

They send this to the contract, along with a deposit $\mathsf{amt}_{\mathsf{dep}}$ that acts to promise they'll come back to vote in the second round, and the number of tokens wgt that they want to put behind their vote. The contract then verifies the proof and checks that the participant has enough tokens, and if so it stores the sent values in voters, associated with the same public key.

At the end of the first round of voting, the contract fixes the set of participants to be all keys in voters, after ensuring that all their first-round votes are distinct (i.e., that they use different values for c_0). To achieve the self-tallying property, we follow Hao et al. [16] in fixing a specific group element for each participant to use in the second round; in particular, if there are m voters then we define

$$Y_j \leftarrow \prod_{0 \leq i < j, j < k \leq m} c_{i,0}^{\mathsf{wgt}_i} c_{k,0}^{-\mathsf{wgt}_k}$$

for all j, $1 \leq j \leq m$ (where j represents the j-th public key, and the ordering on keys can be either lexicographical or in the order they were received).

```
Tally()
find total such that $g^{\text{total}} = \texttt{tallyG}$
tally ← total

Payout()
for all pk ∈ voters
    $U_{\mathbf{pk}} \leftarrow \left(g_1^{1-\texttt{outcome}} c_{\mathbf{pk},1}^{2\cdot\texttt{outcome}-1}\right)$
    curators[pk][C] ← curators[pk][C] · $U_{\mathbf{pk}}$
```

Fig. 5. The internal smart contract functions.

In the second round, the j-th voter can compute their value Y_j. They now provide another value, c_2, that is a commitment to their vote, still using x as the randomness but this time using Y as the base instead of h_0. They also include another ElGamal-style encryption (c_3, c_4) (but again one for which no one knows the decryption key), which is used only in our proof of dispute freeness. They then prove a relation R_{Vote2} to demonstrate that (1) the same value x is used as randomness in the first and second rounds; (2) the commitments c_1 and c_2 from the first and second round contain the same vote and (3) that this vote is either a 0 or a 1; and (4) that c_3 and c_4 are also formed correctly.

$$\begin{aligned}R_{\mathsf{Vote2}} =\{&((Y, c_0, c_1, c_2, c_3, c_4), (\mathsf{vote}, x, s)) : c_0 = g_0^x \wedge \\ &((c_1, c_2) = (h_0^x, Y^x) \vee (c_1 g_1^{-1}, c_2 g_1^{-1}) = (h_0^x, Y^x)) \\ &\wedge\ (c_3, c_4) = (g_0^s, (h_0^{-1} Y)^x h_1^s)\}\end{aligned}$$

A zero-knowledge argument of knowledge for R_{Vote2} is specified in Section 6.

To process this, the contract checks the proof, using the data from both rounds of voting, as shown in Figure 4. If the proof verifies, then the contract folds the value c_2 into its tally and returns the deposit from the first round to the participant. If not, or if the user never sends the second-round transaction in the first place, the deposit acts as a penalty fee and also could be used to reimburse gas costs for honest participants. If needed, the contract could also deduct from the user's deposited tokens to further penalize them, especially after repeat offenses (at which point the user would eventually be stripped of their tokens and removed as a curator).

Tallying and paying out. Finally, the smart contract tallies the result, as seen in Figure 5. The running tally has already been computed by the contract during Process_Vote2, and we argue now that this process is self-tallying; i.e., the contract can compute the tally without any help.

Lemma 5.1. *After the second-round transactions of all m voters have been processed,* $\texttt{tallyG} = g_1^{\sum_{i=1}^m \mathsf{wgt}_i \mathsf{vote}_i}$.

Proof. According to how the Y_j values were computed,

$$\begin{aligned}Y_j &= \prod_{i<j,j<k} c_{i,0}^{\mathsf{wgt}_i} c_{k,0}^{-\mathsf{wgt}_k} \\ &= \prod_{i<j,j<k} g_0^{x_i \mathsf{wgt}_i} g_0^{-x_k \mathsf{wgt}_k} \\ &= g_0^{\sum_{i<j} x_i \mathsf{wgt}_i - \sum_{j<i} x_i \mathsf{wgt}_i}.\end{aligned}$$

It is thus the case that

$$\begin{aligned}\prod_{j=1}^m Y_j^{x_j \mathsf{wgt}_j} &= g_0^{\sum_j x_j \mathsf{wgt}_j (\sum_{i<j} x_i \mathsf{wgt}_i - \sum_{j<i} x_i \mathsf{wgt}_i)} \\ &= g_0^{\sum_j \sum_{i<j} x_i x_j \mathsf{wgt}_i \mathsf{wgt}_j - \sum_j \sum_{j<i} x_i x_j \mathsf{wgt}_i \mathsf{wgt}_j} \\ &= g_0^{\sum_j \sum_{i<j} x_i x_j \mathsf{wgt}_i \mathsf{wgt}_j - \sum_j \sum_{i<j} x_j x_i \mathsf{wgt}_j \mathsf{wgt}_i} \\ &= 1.\end{aligned}$$

After all m voters have run Vote2 we then get

$$\begin{aligned}\texttt{tallyG} &= \prod_{i=1}^m c_{i,2}^{\mathsf{wgt}_i} \\ &= \prod_{i=1}^m g_1^{\mathsf{wgt}_i \mathsf{vote}_i} Y_i^{\mathsf{wgt}_i x_i} \\ &= g_1^{\sum_{i=1}^m \mathsf{wgt}_i \mathsf{vote}_i}\end{aligned}$$

as desired. □

Thus, finding the real tally means finding the discrete logarithm of tallyG, which can be achieved by brute force. While this is a potentially expensive computation (especially to do on-chain), it is made significantly cheaper by restricting the allowable weights and the number of voters, which can be parameters built into the contract. For example, if the only allowable weight is 1, then the maximum value in the exponent is the number of voters. Additionally, a volunteer could compute the tally off-chain and submit it to the smart contract, which could verify the correctness of this tally by confirming that $g_1^{\mathsf{tally}} = \texttt{tallyG}$.

The contract then sets a variable outcome according to the value of tally and the voting policy for the poll. For example, if the policy states that a simple majority wins, then if $\texttt{tally} > \frac{1}{2}\sum_{i=1}^m \mathsf{wgt}_i$ the smart contract sets $\texttt{outcome} \leftarrow 1$, and otherwise $\texttt{outcome} \leftarrow 0$.

Finally, the contract must update the tokens of each curator according to how they voted. (In addition, it sends the public deposits created in Apply and Challenge to the service or challenger, according to the outcome of the vote.) As seen in Figure 5 and discussed in Section 5.1, we can use c_1, which acts as a commitment to the boolean vote vote, to form a commitment U_{pk} to the outcome of the boolean expression ($\mathsf{vote} = \texttt{outcome}$); i.e., U_{pk} is a commitment to 1 if $\mathsf{vote} = \texttt{outcome}$ and a commitment to 0 otherwise. We can then add this value into their committed balance $C = g_0^{\mathsf{wgt}} h_1^r$ by multiplying the two commitments together, which means the curator earns one token for voting "correctly" and nothing otherwise. If a different payout structure were desired, this could be achieved by manipulating U_{pk} appropriately (e.g., squaring it if the reward should be two tokens). Importantly, the reward for a curator is not proportional to the weight they used in the vote.

Since the curator knows both the original randomness used in C and the randomness used in c_1, as well as how much they were rewarded, they can update their locally stored weight and randomness so that they still know the opening of the commitment C, which is needed to run Update. This means updating the value of (amt, r) as follows:

	outcome = 0	**outcome = 1**
$\mathsf{vote} = 0$	$(\mathsf{amt} + 1, r - x)$	$(\mathsf{amt}, r + x)$
$\mathsf{vote} = 1$	$(\mathsf{amt}, r - x)$	$(\mathsf{amt} + 1, r + x)$

5.3 Security

We now argue why our construction achieves the notions of security defined in Section 4. For notational simplicity, our proofs assume all weights are equal to 1, but could be modified to allow for arbitrary weights.

Theorem 5.2. *If* $(\mathsf{Prove}, \mathsf{Verify})$ *is a zero-knowledge argument of knowledge and DDH holds, then the construction above satisfies vote secrecy, as specified in Definition 4.1.*

Our full proof of vote secrecy is quite involved, and can be found in Appendix C. Intuitively, we must transition from the honest vote secrecy game to a game in which all information about the votes of honest participants is hidden, at which point the adversary can have no advantage. In terms of honest participants, there must be at least two in order to satisfy the requirement that $\mathsf{vote\_count} = 0$, so we consider the case of two honest participants as the most hostile (but our results generalize to any even number of honest participants, which is again required in the game).

We first transition from real to simulated proofs, which is indistinguishable by zero-knowledge. Using DDH, we now target an honest pair of participants and embed extra randomness into their c_2 values at the same time. This means, however, that we embed the DDH challenge into their c_0 values, and thus don't know their values for x, which in turn makes it difficult for them to compute Y^x. Luckily, we don't need to compute these values until the second round, at which point all of the first-round votes have been sent. We can thus extract from the adversary's first-round proof of knowledge to get their values of x, at which point we can use them to simulate Y^x. In order to argue that the extractor can successfully extract these values across all adversarial participants, we must restrict the adversary to operating in a single session; i.e., we restrict $j = 1$ and cannot prove concurrent vote secrecy for this protocol (but can for a subsequent version [9]).

We next switch to decoupling the randomness between c_0 and c_1. While this is an easier task, as c_1 has the form of a normal Pedersen commitment, we still embed a DDH challenge into c_0, which still makes it difficult to compute the Y^x term in c_2. We must thus proceed voter by voter to ensure that the discrete logarithm of Y is known at every step, since we can extract the contribution to it for each adversarial participant and form it ourselves for the other honest participants. We then switch to decoupling the randomness between c_3 and c_4, which is significantly easier. Finally, we reach a game in which the c_1 and c_4 values both use independent randomness, which perfectly hides the information they contain, and the c_2 values use extra randomness that is correlated across pairs of honest participants. We can nevertheless argue that the distributions are identical for both cases, $b = 0$ or $b = 1$, which proves the theorem.

Theorem 5.3. *If* $(\mathsf{Prove}, \mathsf{Verify})$ *is an argument of knowledge, then the construction above satisfies dispute freeness, as specified in Definition 4.2.*

Our proof of this theorem can be found in Appendix D, and in contrast is quite simple. This is due to the fact that the (c_0, c_1) and (c_3, c_4) pairs are formed as ElGamal ciphertexts, albeit one for which no real-world participant knows the decryption key. Our reduction can instead form the parameters (in particular, h_0 and h_1) so that it does know the decryption key, which allows it to recover g_1^{vote} from (c_0, c_1) and $(h_0^{-1} Y)^x$ from (c_3, c_4). It can furthermore remove the h_0^{-x} term here, as it knows the discrete logarithm of h_0 with respect to g_0 (and knows $c_0 = g_0^x$), to recover Y^x, and in turn g_1^{vote} from c_2. If these are not equal to each other, or not equal to 1 ($\mathsf{vote} = 0$) or g_1 ($\mathsf{vote} = 1$), then this instance is not in the language. On the other hand, in order for the vote to complete successfully, all proofs must verify, so the reduction can output the instance and proof to successfully break knowledge soundness.

Aside from the obvious implications of these two security properties, they also combine to prevent more subtle attacks. For example, we can consider a *front-running* attack in which an adversary observes the transactions of other participants before sending its own to potentially change its mind about its own vote (e.g., to try to earn tokens by voting with the majority). These attacks are impossible based on our two security properties: vote secrecy ensures that the adversary doesn't learn anything about the votes of others in the first round (so can't use any information to its advantage), and dispute freeness ensures that the adversary must stick to its original vote in the second round.

There are also some properties, however, that are not covered in our model. As discussed above, we do not prevent an adversary from thwarting the voting process by not sending its second-round vote or otherwise provide robustness, but we do provide financial disincentives for this behavior in the form of a deposit refunded only after a valid second-round vote is cast (and perhaps harsher penalties if there is repeated misbehavior). More crucially, we currently do not provide any notion of *receipt-freeness*, or coercion resistance, as voters can easily prove they voted a certain way by revealing x. This is quite important for some of the potential applications of TCRs, in which bribery is a real threat that could undermine the quality of the registry.

One approach we could take would be to disincentivize coercion by again applying financial penalties, in this case if someone can demonstrate that a participant revealed their vote. For example, if a bribed curator reveals x and someone submits this to the contract, it could check that this is the correct x and take some penalty fee from their deposited tokens, or even remove them as a curator altogether, which would in turn make them less attractive as a target for bribery. This approach could also be extended to more sophisticated methods for revealing x (e.g., the contract could check a proof of knowledge of x), although it is unlikely to be able to handle all of them. Another option would be to have the other curators manually inspect any evidence of bribery and submit votes indicating whether or not they think it is valid, and having the contract apply a penalty if a sufficient fraction of them agree that it is. Thus, while we currently do not provide any cryptographic guarantee about coercion resistance, this can again be addressed with an incentive-driven approach, and we leave it as interesting future work to see if it can be addressed more rigorously.

6 Arguments of Knowledge for our Construction

6.1 Proving R_{wgt} and R_{Vote1}

To construct our zero-knowledge arguments for R_{wgt}, R_{Vote1}, and R_{Vote2} we use two main building blocks: a proof of discrete logarithm and a proof that one out of two values is a commitment to 0. These can both be achieved using sigma protocols and then applying the Fiat-Shamir heuristic [11] to obtain a non-interactive proof. The protocols are both standard but are included for completeness in Figure 7 (in Appendix A).

To prove R_{wgt}, which is used in Deposit, a prover must demonstrate knowledge of r such that $C = g_1^{\mathsf{wgt}} h_0^r$, for known wgt. This can be achieved using a proof of knowledge of discrete logarithm (the left-hand side of Figure 7), where the prover's input is r and the shared input is $h_0^r = g_1^{-\mathsf{wgt}} C$.

To prove R_{Vote1}, which is used in Vote1, a prover must demonstrate knowledge of x such that $c_0 = g_0^x$. Again, this can be achieved using a proof of knowledge of discrete logarithm (the left-hand side of Figure 7), where the prover's input is x and the shared input is c_0.

6.2 Proving R_{Vote2}

To prove R_{Vote2}, which is used in Vote2, a prover must demonstrate that $\{c_i\}_{i=0}^4$ are formed correctly. More specifically, it needs to show knowledge of (vote, x, s) such that (1) $c_0 = g_0^x$, (2) $(c_1, c_2) = (h_0^x, Y^x)$ or $(c_1 g_1^{-1}, c_2 g_1^{-1}) = (h_0^x, Y^x)$, and (3) $(c_3, c_4) = (g_0^s, (h_0^{-1} Y)^x h_1^s)$. Rather than prove this directly, we claim that — assuming computing discrete logarithms is hard — this follows if we instead provide a proof for the following expanded relation.

$$R' = \left\{ \begin{array}{ll} ((Y, \{A_i\}_{i=0}^5), (\{x_i\}_{i=0}^2, b)) : & A_0 = g_0^{x_0} \\ & A_1 = g_1^b h_0^{x_1} \\ & A_2 = g_1^b Y^{x_1} \\ & A_3 = (h_0 Y^{-1})^{x_0} \\ & A_4 = h_1^{x_2} \\ & A_5 = g_0^{x_2} \end{array} \right\}$$

where

$$A_0 = c_0 \quad A_1 = c_1 \quad A_2 = c_2 \quad A_3 = c_1c_2^{-1}$$
$$A_4 = c_1c_2^{-1}c_4 \quad A_5 = c_3.$$

Using a prover for R', our argument for R_{Vote2} is then quite simple: the prover and verifier independently calculate the instance for R' using the c_i values, and the prover calculates the witness using $b = \mathsf{vote}$, $x_0 = x_1 = x$, and $x_2 = s$. More formally,

```
Prove($R_{\mathsf{Vote2}}, (Y, \{c_i\}_{i=0}^4), (\mathsf{vote}, x, s)$)
$\phi \leftarrow (Y, c_0, c_1, c_2, c_1c_2^{-1}, c_1c_2^{-1}c_4, c_3)$
$w \leftarrow (x, s, \mathsf{vote})$
$\pi \leftarrow \mathsf{Prove}(R', \phi, w)$
return $\pi$

Verify($R_{\mathsf{Vote2}}, (Y, \{c_i\}_{i=0}^4), \pi$)
$\phi \leftarrow (Y, c_0, c_1, c_2, c_1c_2^{-1}, c_1c_2^{-1}c_4, c_3)$
return Verify($R', \phi, \pi$)
```

We now must prove that it suffices to prove R_{Vote2} by giving an argument of knowledge for R'.

Lemma 6.1. *Suppose that* (Prove, Verify) *is a zero-knowledge argument of knowledge for R'. Then our proof above is a zero-knowledge argument of knowledge for R_{Vote2}.*

Proof. Zero knowledge is achieved directly by using the simulator for R'. Suppose that there exists a PT emulator $\mathcal{X}$ such that for any PT adversary $\mathcal{A}$, if $\mathcal{A}$ has a non-negligible advantage of outputting a valid proof for R' then $\mathcal{X}$ has an overwhelming probability of outputting a valid witness for R'. We construct an emulator $\mathcal{X}_{\mathsf{vote}}$ and an adversary $\mathcal{B}$ such that $\mathcal{X}_{\mathsf{vote}}$ outputs a valid witness for R_{Vote2}.

We first focus on the values $x_0, x_1 \in \mathbb{F}$ and $b \in \{0, 1\}$ output by $\mathcal{X}$ (after rewinding once), which are such that

$$c_0 = g_0^{x_0} \qquad c_1 = g_1^b h_0^{x_1}$$
$$c_2 = g_1^b Y^{x_1} \qquad c_1c_2^{-1} = (h_0Y^{-1})^{x_0}.$$

Observe that $h_0^{x_1}Y^{-x_1} = (h_0Y^{-1})^{x_0}$ and thus $x_0 = x_1$.

Next, we focus on the value $x_2 \in \mathbb{F}$ output by $\mathcal{X}$ (again, after rewinding once), which is such that

$$c_1c_2^{-1}c_4 = h_1^{x_2} \text{ and } c_3 = g_0^{x_2}.$$

By our argument above, $c_1c_2^{-1} = h_0^xY^{-x}$, and so

$$c_4 = h_0^{-x}Y^xh_1^{x_2}.$$

Thus the emulator has found b, x_0, x_2 such that

$$(c_0, c_1, c_2, c_3, c_4) = (g_0^{x_0}, g_1^b h_0^{x_0}, g_1^b Y^{x_0}, g_0^{x_2}, h_0^{-x_0}Y^{x_0}h_1^{x_2}),$$

as required for R_{Vote2}. □

```
Prover's input: $(\{x_i\}_{i=0}^2, b)$ such that
$A_0 = g_0^{x_0}, A_1 = g_1^b h_0^{x_1}, A_2 = g_1^b Y^{x_1}, b \in \{0,1\}$
$A_3 = (h_0Y^{-1})^{x_0}$, $A_4 = h_1^{x_2}, A_5 = g_0^{x_2}$

Shared input: $(Y, \{A_i\}_{i=0}^5)$

Prove ↦ Verify :
$r_0, r_1, s_0, s_1, s_2 \xleftarrow{\$} \mathbb{F}$
$R_0, R_1, R_2, R_3 \leftarrow g_0^{r_0}, (h_0Y^{-1})^{r_0}, h_1^{r_1}, g_0^{r_1}$
$S_0, T \leftarrow g_1^{s_0}h_0^{s_1}, g_1^{s_0 b}h_0^{s_2}$
$S_1 \leftarrow g_1^{s_0}Y^{s_1}$
send $R_0, R_1, R_2, R_3, S_0, S_1, T$

Verify ↦ Prove :
send $a_0, a_1, a_2 \xleftarrow{\$} \mathbb{F}$

Prove ↦ Verify :
$d_0 \leftarrow r_0 + a_0x_0$
$d_1 \leftarrow r_1 + a_1x_2$
$u, v, w \leftarrow s_0 + a_2b, s_1 + a_2x_1, s_2 + x_1(a_2 - u)$
send $d_0, d_1, u, v, w$

Verify :
check $g_0^{d_0} = R_0A_0^{a_0}$
check $(h_0Y^{-1})^{d_0} = R_1A_3^{a_0}$
check $h_1^{d_1} = R_2A_4^{a_1}$
check $g_0^{d_1} = R_3A_5^{a_1}$
check $g_1^u h_0^v = S_0A_1^{a_2}$
check $g_1^u Y^v = S_1A_2^{a_2}$
check $h_0^w = TA_1^{a_2-u}$
return 1 if checks pass, else 0
```

Fig. 6. Zero-knowledge argument of knowledge for R'.

6.2.1 Proving R'

Now that we have convinced ourselves that we can prove R_{Vote2} by proving R', we need only construct a zero-knowledge argument for R'. This is in Figure 6 and is essentially a combination of the building blocks above, using a one-out-of-two proof for A_1 and A_2 and a proof of discrete logarithm for the rest.

Lemma 6.2. *If discrete logarithm holds, then the argument presented in Figure 6 for the relation R' satisfies zero knowledge and 2-special soundness.*

We provide a proof of Lemma 6.2 in Appendix B. Given that the proof of R' is a straightforward combination of standard building blocks, its security follows relatively easily from theirs.

6.2.2 Efficiency

In terms of the overall efficiency of proving R_{Vote2}, both the prover and the verifier must compute two group exponentiations (one for c_2^{-1} and one for Y^{-1}) to translate the instance for R_{Vote2} into an instance for R'.

To prove R', from Figure 6 we see that the proof consists of 7 group elements and 3 field elements. Computing R_0, R_1, R_2, R_3 requires 4 exponentiations, and computing S_0, S_1, T requires an additional 5 exponentiations, so the prover computes 9 group exponentiations in total for R'. Across all checks, the verifier computes 16 group exponentiations in proving R'.

Putting everything together, we see that for R_{Vote2}, proofs consist of 7 group elements and 3 field elements, the prover computes 11 group exponentiations, and the verifier computes 18 group exponentiations. We discuss the practicality of this for a smart contract-based setting in Section 7.

7 Implementation

We implement Deposit, Update, Vote1, and Vote2 on the curator side, and each of the corresponding verification functions on the smart contract side. The curator code is written in 200 lines of Rust, using the bn crate for elliptic curve arithmetic; specifically, we use 256-bit primes and the BN256 G1 curve. The verifier code is written in 200 lines of Solidity. The costs associated with each of these phases are in Table 1.

As of the Ethereum Istanbul upgrade in December 2019, the gas cost for performing one exponentiation, or `ECMUL`, is 6,000 gas [7] (as compared to the 40,000 gas it was before this upgrade [21]). As we see in Table 1, this dramatic decrease brings our protocol into the realm of practicality, even on a constrained platform like Ethereum. Each transaction in Ethereum is associated with a transaction cost, which in some cases is more than the gas consumption of our verification functions. This cost must be paid with each transaction, so if a verifier is calling several functions at once, this cost can be amortized across the functions. In practice we do not expect curators to be able to benefit from this amortization, since votes must be placed by distinct curators and voting proceeds in sequential phases. Nevertheless, for the purposes of illustration we show the cost both with and without the transaction cost.

At the time of writing, the standard gas cost is 2 GWei/gas,[8] and 1 ether is the equivalent of 223 USD. This means that each individual phase costs up to 7 cents, and that it costs under 12 cents to cast a vote (running Vote1 and Vote2).

8 Conclusions

In this paper, we provided the first cryptographic construction of a token-curated registry, which we prove secure under the DDH assumption in a new model that also encapsulates electronic voting. In particular, our construction is the first to achieve a provable notion of vote secrecy, and our proofs of security also imply the security of an existing voting protocol due to Hao et al. [16]. Our protocol is quite minimal, as demonstrated by its inexpensive implementation on top of the Ethereum platform. Nevertheless, it raises the question of whether or not there are new proof techniques (e.g., ways to shift computation from the verifier to the prover other than SNARKs) that might be developed to further lower costs.

Acknowledgements

This work was done in part while Sarah Meiklejohn was visiting the Simons Institute for the Theory of Computing, and the first three authors were supported in part by EPSRC Grant EP/N028104/1.

References

[1] B. Adida. Helios: Web-based open-audit voting. In P. C. van Oorschot, editor, *USENIX Security 2008*, pages 335–348, San Jose, CA, USA, July 28 – Aug. 1, 2008. USENIX Association.

[2] A. Asgaonkar and B. Krishnamachari. Token curated registries - a game theoretic approach, 2018. https://arxiv.org/pdf/1809.01756.pdf.

[3] F. Bao, R. H. Deng, and H. Zhu. Variations of Diffie-Hellman problem. In S. Qing, D. Gollmann, and J. Zhou, editors, *ICICS 03*, volume 2836 of *LNCS*, pages 301–312, Huhehaote, China, Oct. 10–13, 2003. Springer, Heidelberg, Germany.

8 https://ethgasstation.info/

Stage	Time (μs)	Verification (gas)	Verification and tx (gas)	Verification and tx (USD)
Deposit	328	38,659	60,507	0.027
Update	328	18,641	40,489	0.018
Vote1	656	58,677	80,525	0.036
Vote2	3546	130,696	156,192	0.070
Total	4854	246,673	337,713	0.151

Table 1. The average runtime (in microseconds and averaged over 300 runs) and gas costs of each stage, as well as the total costs across all the stages. The conversion to USD uses rates of 2 GWei/gas and 223 USD/ether.

[4] M. Bellare and P. Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In S. Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 409–426, St. Petersburg, Russia, May 28 – June 1, 2006. Springer, Heidelberg, Germany.

[5] J. D. C. Benaloh. *Verifiable Secret-Ballot Elections*. PhD thesis, Yale University, 1987.

[6] D. Bernhard, V. Cortier, D. Galindo, O. Pereira, and B. Warinschi. SoK: A comprehensive analysis of game-based ballot privacy definitions. In *2015 IEEE Symposium on Security and Privacy*, pages 499–516, San Jose, CA, USA, May 17–21, 2015. IEEE Computer Society Press.

[7] A. S. Cardozo and Z. Williamson. EIP 1108: Reduce alt_bn128 precompile gas costs, 2018. https://eips.ethereum.org/EIPS/eip-1108.

[8] ConsenSys. Partial-lock commit-reveal voting. https://github.com/ConsenSys/PLCRVoting.

[9] E. C. Crites, M. Maller, S. Meiklejohn, and R. Mercer. Reputable list curation from decentralized voting, 2020. https://eprint.iacr.org/2020/709.pdf.

[10] Enigma. Secret voting: An update & code walkthrough. https://blog.enigma.co/secret-voting-an-update-code-walkthrough-605e8635e725.

[11] A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In A. M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 186–194, Santa Barbara, CA, USA, Aug. 1987. Springer, Heidelberg, Germany.

[12] M. Goldin. Token-curated registries 1.0, Sept. 2017. https://medium.com/@ilovebagels/token-curated-registries-1-0-61a232f8dac7.

[13] R. Gray. Lies, propaganda and fake news: a challenge for our age, Mar. 2017. http://www.bbc.com/future/story/20170301-lies-propaganda-and-fake-news-a-grand-challenge-of-our-age.

[14] J. Groth. Efficient maximal privacy in boardroom voting and anonymous broadcast. In A. Juels, editor, *FC 2004*, volume 3110 of *LNCS*, pages 90–104, Key West, USA, Feb. 9–12, 2004. Springer, Heidelberg, Germany.

[15] J. Groth and M. Kohlweiss. One-out-of-many proofs: Or how to leak a secret and spend a coin. In E. Oswald and M. Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 253–280, Sofia, Bulgaria, Apr. 26–30, 2015. Springer, Heidelberg, Germany.

[16] F. Hao, P. Y. A. Ryan, and P. Zielinski. Anonymous voting by two-round public discussion. *IET Information Security*, 4:62–67(5), June 2010.

[17] B. Hemenway Falk and G. Tsoukalas. Token-weighted crowdsourcing, Dec. 2018. The Wharton School Research Paper.

[18] K. Ito and H. Tanaka. Token-curated registry with citation graph, 2019. https://arxiv.org/pdf/1906.03300.pdf.

[19] A. Kiayias and M. Yung. Self-tallying elections and perfect ballot secrecy. In D. Naccache and P. Paillier, editors, *PKC 2002*, volume 2274 of *LNCS*, pages 141–158, Paris, France, Feb. 12–14, 2002. Springer, Heidelberg, Germany.

[20] P. McCorry, S. F. Shahandashti, and F. Hao. A smart contract for boardroom voting with maximum voter privacy. In A. Kiayias, editor, *FC 2017*, volume 10322 of *LNCS*, pages 357–375, Sliema, Malta, Apr. 3–7, 2017. Springer, Heidelberg, Germany.

[21] C. Reitwiessner. EIP 196: Precompiled contracts for addition and scalar multiplication on the elliptic curve alt_bn128, 2017. https://eips.ethereum.org/EIPS/eip-196.

[22] H. Ritzdorf, K. Wüst, A. Gervais, G. Felley, and S. Capkun. TLS-N: Non-repudiation over TLS enablign ubiquitous content signing. In *NDSS 2018*, San Diego, CA, USA, Feb. 18-21, 2018. The Internet Society.

[23] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi. Town crier: An authenticated data feed for smart contracts. In E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, editors, *ACM CCS 2016*, pages 270–282, Vienna, Austria, Oct. 24–28, 2016. ACM Press.

[24] F. Zhang, S. K. D. Maram, H. Malvai, S. Goldfeder, and A. Juels. Deco: Liberating web data using decentralized oracles for TLS, 2019. https://arxiv.org/pdf/1909.00938.pdf.

A Standard definitions and building blocks for zero-knowledge proofs

We begin by providing the standard definitions of security for zero-knowledge proofs, starting with special honest verifier zero-knowledge and 2-special soundness. These can be defined formally (omitting public parameters that are given to each party) as follows:

Definition A.1 (SHVZK)**.** *A sigma protocol* $(\mathsf{Prove},\mathsf{Verify})$ *for a relation R satisfies* special honest verifier zero knowledge (SHVZK) *if there exists a PT simulator* Sim *such that for all interactive PT adversaries $\mathcal{A}$,*

$$\begin{aligned}&\Pr[(\phi,w,x)\xleftarrow{\$}\mathcal{A}(1^\lambda); a\xleftarrow{\$}\mathsf{Prove}(\phi,w);\\ &\qquad\qquad z\xleftarrow{\$}\mathsf{Prove}(x):\mathcal{A}(a,z)=1]\\ &\approx \Pr[(\phi,w,x)\xleftarrow{\$}\mathcal{A}(1^\lambda);(a,z)\xleftarrow{\$}\mathsf{Sim}(\phi,x):\mathcal{A}(a,z)=1],\end{aligned}$$

where $\mathcal{A}$ outputs (ϕ,w,x) such that $(\phi,w)\in R$ and $x\in\{0,1\}^\lambda$. If the distributions are identical, the protocol satisfies perfect *SHVZK.*

Definition A.2 (2-special soundness)**.** *A sigma protocol* $(\mathsf{Prove},\mathsf{Verify})$ *for a relation R satisfies* 2-special soundness *if there exists an efficient extractor $\mathcal{X}$ that can compute the witness given two accepting transcripts with the same initial message. Formally, for all PT $\mathcal{A}$,*

$$\begin{aligned}&\Pr[(\phi,a,x_1,z_1,x_2,z_2)\xleftarrow{\$}\mathcal{A}(1^\lambda);\\ &\quad w\leftarrow\mathcal{X}(1^\lambda,\phi,a,x_1,z_1,x_2,z_2)\;:\;(\phi,w)\in R]\approx 1,\end{aligned}$$

where $\mathcal{A}$ outputs distinct $x_1,x_2\in\{0,1\}^\lambda$ and both transcripts are accepting; i.e., $\mathsf{Verify}(1^\lambda,\phi,a,x_i,z_i)=1$ *for $i=1,2$. If the probability is exactly 1, the protocol satisfies* perfect *2-special soundness.*

We now provide in Figure 7 the full specification of two building blocks that are useful in the proofs in Section 6: a proof of discrete logarithm and a proof that one out of two values is a commitment to 0.

Lemma A.3. *The arguments presented in Figure 7 satisfy zero knowledge and 2-special soundness.*

Proof. We start with the proof of knowledge of discrete logarithm. To show that it satisfies zero knowledge, consider a simulator that, given an instance Y, chooses random $a,d\in\mathbb{F}$, sets $R=g^dY^{-a}$, and returns (Y,R,a,d). Then in both the real and simulated transcripts R is distributed uniformly at random, given the random r in the real transcript and the random d in the simulated transcript. Similarly, a is distributed uniformly at random. Given R and a there is a unique d that satisfies the verifier's equations. Thus the real and simulated transcripts are indistinguishable. To show 2-special soundness, consider an extractor that, given two accepting transcripts (Y,R,a_1,d_1) and (Y,R,a_2,d_2), returns $x=\frac{d_1-d_2}{a_1-a_2}$. We argue that x is always the discrete logarithm of Y. To see this observe that $RY^{a_1}=g^{d_1}$ and $RY^{a_2}=g^{d_2}$ implies that $Y^{a_1-a_2}=g^{d_1-d_2}$ and hence $Y=g^{\frac{d_1-d_2}{a_2-a_1}}$.

Prover's input: x **such that** $Y=g^x$	**Prover's input:** $b\in\{0,1\}$, x **such that** $C=g^bh^x$
Shared input: Y	**Shared input:** C
<u>Prove $\mapsto$ Verify</u>	<u>Prove $\mapsto$ Verify</u>
$r\xleftarrow{\$}\mathbb{F}$	$s_0,s_1,s_2\xleftarrow{\$}\mathbb{F}$
send $R\leftarrow g^r$	$S\leftarrow g^{s_0}h^{s_1}$
	$T\leftarrow g^{s_0b}h^{s_2}$
	send S,T
<u>Verify $\mapsto$ Prove</u>	<u>Verify $\mapsto$ Prove</u>
send $a\xleftarrow{\$}\mathbb{F}$	send $a\xleftarrow{\$}\mathbb{F}$
<u>Prove $\mapsto$ Verify</u>	<u>Prove $\mapsto$ Verify</u>
send $d\leftarrow r+ax$	$u\leftarrow s_0+ab$
	$v\leftarrow s_1+ax$
	$w\leftarrow s_2+x(a-u)$
	send u,v,w
<u>Verify</u>	<u>Verify</u>
return $(RY^a=g^d)$	return $(SC^a=g^uh^v)\wedge$ $(TC^{a-u}=h^w)$

Fig. 7. Standard zero-knowledge argument of knowledge for discrete logarithm (on the left-hand side) and for a committed value containing zero or one (on the right-hand side).

For the one-of-two proof, consider a simulator that, given an instance C, chooses random $a,u,v,w\in\mathbb{F}$ and sets $S=g^uh^vC^{-a}$ and $T=h^wC^{u-a}$. Then in both the real and simulated transcripts u, S, T and u are distributed uniformly at random due to the random s_0, s_1 and s_2 in the real transcript and the random v and w in the simulated transcript. Similarly, a is distributed uniformly at random. Given S,T,a,u there is a unique v,w that satisfies the verifier's equations. Thus the real and simulated transcripts are indistinguishable. To show 2-special soundness, consider an extractor that, given two accepting transcripts (C,S,T,a_1,u_1,v_1,w_1) and (C,S,T,a_2,u_2,v_2,w_2), returns

$$\begin{aligned}b=0,\;x&=\frac{(w_1-w_2)}{a_1-u_1-a_2+u_2}\text{if}(a_1-a_2)\neq(u_1-u_2)\\ b=1,\;x&=\frac{(v_1-v_2)}{a_1-a_2}\text{if}(a_1-a_2)=(u_1-u_2)\end{aligned}$$

We argue that $C=g^bh^x$. To see this observe that if $(a_1-a_2)\neq(u_1-u_2)$ then $TC^{a_1-u_1}=h^{w_1}$ and $TC^{a_2-u_2}=h^{w_2}$ gives us that $C^{a_1-u_1-a_2+u_2}=h^{w_1-w_2}$ and hence

$C = h^{\frac{w_1 - w_2}{a_1 - u_1 - a_2 + u_2}}$. If $(a_1 - a_2) = (u_1 - u_2)$ then $SC^{a_1} = g^{u_1}h^{v_1}$ and $SC^{a_2} = g^{u_2}h^{v_2}$ gives us that $C^{a_1 - a_2} = g^{u_1-u_2}h^{v_1-v_2}$ and hence $C = g^{\frac{u_1-u_2}{a_1-a_2}} h^{\frac{v_1-v_2}{a_1-a_2}} = gh^x$. □

B Proof of Lemma 6.2

Proof. We begin by proving the zero-knowledge property of the argument. To do this we define a simulator that outputs proof elements that are indistinguishable from the outputs of an honest prover. We give this simulator a trapdoor, which is the discrete logarithm relations between g_0, h_0, h_1, and Y. Our simulator is further able to program the random oracle.

Zero-knowledge. Consider the simulator that knows γ_0 such that $(h_0Y^{-1}) = g_0^{\gamma_0}$ and $g_0 = h_1^{\gamma_1}$. It begins by choosing random values

$$a_0, a_1, a_2, d_0, d_1, u, v, w$$

It then sets

$$\begin{array}{ll} R_0 = g_0^{d_0}A_0^{-a_0} & R_1 = R_0^{\gamma_0} \\ R_2 = h_1^{d_1}A_4^{-a_1} & R_3 = R_1^{\gamma_1} \\ S_0 = g_1^u h_0^v A_1^{-a_2} & S_1 = g_1^u Y^v A_2^{-a_2} \\ T = h_0^w A_1^{u-a_2} & \end{array}$$

and programs the random oracle to return a_0, a_1, a_2 on the query $(R_0, R_1, R_2, R_3, S_0, S_1, T)$. Finally it returns

$$(R_0, R_1, R_2, R_3, S_0, S_1, T), (d_0, d_1, u, v, w).$$

To see that the output of the simulator is indistinguishable from the output of the honest prover, first see that the R_0 and R_2 output by both the prover and the simulator are chosen uniformly at random. Moreover, both values of R_1 and R_3 are respectively equal to $R_0^{\gamma_0}$ and $R_1^{\gamma_1}$, so are distributed identically. Similarly, both values of S_0, S_1, T are distributed uniformly at random due to the blinders s_0, s_1, s_2. Then because

$$g_1^u h_0^v = S_0 A_1^{a_2} \wedge g_1^u Y^v = S_1 A_2^{a_2}$$

we have that

$$(h_0Y^{-1})^v = S_0 S_1^{-1} A_1^{a_2} A_2^{-a_2},$$

so there is a unique value v that satisfies the verifier's equations. Given v there are also unique values u and w that satisfies the verifier's equations. Hence the simulator's output u, v, w is distributed identically to the same values output by the prover. The simulator and the prover thus sample proofs from the same distribution and the scheme is zero-knowledge.

Knowledge soundness. We now prove knowledge soundness by designing an extractor $\mathcal{X}$ that rewinds a succeeding adversary at most once. We demonstrate how this extractor either breaks the discrete logarithm assumption or extracts a valid witness.

To begin, $\mathcal{X}$ runs $\mathcal{A}$ to obtain $R_0, R_1, R_2, R_3, S_0, S_1, T$. Then, by rewinding the adversary the extractor obtains, for $i \in \{0, 1\}$,

$$(a_{i,0}, a_{i,1}, a_{i,2})$$

and

$$(d_{i,0}, d_{i,1}, u_i, v_i, w_i)$$

that satisfy the verifier's checks.

From the first check we see that $R_0 = g_0^{-d_{i,0}} A_0^{a_{i,0}}$. Hence, setting $\log_{g_0}(A_0)$ to the indeterminate X_0, we see that

$$a_{0,0}X_0 - d_{0,0} = a_{1,0}X_0 - d_{1,0}$$

and

$$X_0 = \frac{d_{0,0} - d_{1,0}}{a_{0,0} - a_{1,0}}.$$

From the second check we see that $R_1 = (h_0Y^{-1}) - d_{i,0}A_3^{a_{i,0}}$. Hence, setting $\log_{h_0Y^{-1}}(A_3)$ to the indeterminate X_1, we see that

$$a_{0,0}X_1 - d_{0,0} = a_{1,0}X_1 - d_{1,0}$$

and

$$X_1 = \frac{d_{0,0} - d_{1,0}}{a_{0,0} - a_{1,0}} = X_0.$$

The extractor returns $x_0 = \frac{d_{0,0} - d_{1,0}}{a_{0,0} - a_{1,0}}$.

From the third check we see that $R_3 = h_1^{-d_{i,1}} A_4^{a_{i,1}}$. Hence, setting $\log_{h_1}(A_4)$ to the indeterminate X_2, we see that

$$a_{0,1}X_2 - d_{0,1} = a_{1,1}X_2 - d_{1,1}$$

and

$$X_2 = \frac{d_{0,1} - d_{1,1}}{a_{0,1} - a_{1,1}}.$$

From the fourth check we see that $R_4 = g_0 - d_{i,1}A_5^{a_{i,1}}$. Hence, setting $\log_{g_0}(A_5)$ to the indeterminate X_3, we see that

$$a_{0,1}X_3 - d_{0,1} = a_{1,1}X_3 - d_{1,1}$$

and

$$X_3 = \frac{d_{0,1} - d_{1,1}}{a_{0,1} - a_{1,1}} = X_2.$$

The extractor returns $x_2 = \frac{d_{0,1} - d_{1,1}}{a_{0,1} - a_{1,1}}$.

From the fifth check we see that $g_1^{u_i} h_0^{v_i} = S_0 A_1^{a_{i,2}}$ so

$$g_1^{u_0 - u_1} h_0^{v_0 - v_1} = A_1^{a_{0,2} - a_{1,2}}.$$

Thus the extractor obtains (b, x_1) such that $A_1 = g_1^b h_0^{v_i}$ and

$$(b, x_1) = \left(\frac{u_0 - u_1}{a_{0,2} - a_{1,2}}, \frac{v_0 - v_1}{a_{0,2} - a_{1,2}} \right) \tag{1}$$

Similarly, from the sixth check we see that $h_0^{w_i} = T_0 A_1^{a_{i,2} - u_i}$ so

$$h_0^{w_0 - w_1} = A_1^{a_{0,2} - u_0 - a_{1,2} + u_1} = \left(g_1^b h_0^{x_1} \right)^{(a_{0,2} - u_0 - a_{1,2} + u_1)}.$$

Thus, either the extractor finds a non-trivial discrete-log relation between h_0 and g_1, which it then returns to break discrete logarithm, or $b = 0$ (and thus is a valid witness component), or

$$(a_{0,2} - a_{1,2}) = (u_0 - u_1).$$

Looking back on Equation 1, we thus see that

$$b = \frac{u_0 - u_1}{a_{0,2} - a_{1,2}} = \frac{u_0 - u_1}{u_0 - u_1} = 1$$

and thus $b \in \{0, 1\}$ is a valid witness component.

From the seventh check we see that $g_1^{u_i} Y^{v_i} = S_1 A_2^{a_{i,2}}$ so

$$g_1^{u_0 - u_1} Y^{v_0 - v_1} = A_2^{a_{0,2} - a_{1,2}}.$$

Thus the extractor obtains (b', x_1') such that $A_1 = g_1^{b'} Y^{v_i'}$ and

$$(b', x_1') = \left(\frac{u_0 - u_1}{a_{0,2} - a_{1,2}}, \frac{v_0 - v_1}{a_{0,2} - a_{1,2}} \right).$$

These values are exactly equal to (b, x_1). Hence

$$A_2 = g_1^b Y^{x_1}$$

and (x_0, x_1, x_2, b) is a valid witness for R'. □

C Proof of Vote Secrecy (Theorem 5.2)

Let $\mathcal{A}$ be a PT adversary playing game $\mathsf{G}_{\mathcal{A}}^{\text{secrecy}}(\lambda)$, and let n denote the number of honest voters. We provide PT adversaries $\mathcal{B}_0$, $\mathcal{B}_1$, $\mathcal{B}_2$, $\mathcal{B}_4$, and a family of PT adversaries $\mathcal{B}_{i,3}$ for i, $0 \le i \le n$, such that

$$\begin{aligned}
&\mathbf{Adv}_{\mathcal{A}}^{\text{secrecy}}(\lambda) \\
&= 2(\mathbf{Adv}_{\mathcal{B}_0}^{\text{zk}}(\lambda) + \mathbf{Adv}_{\mathcal{B}_1}^{\text{zk}}(\lambda) + \mathbf{Adv}_{\mathcal{B}_1}^{\text{snd}}(\lambda) \\
&+ \frac{n}{2}(\mathbf{Adv}_{\mathcal{B}_2}^{\text{ddh}}(\lambda) + \mathbf{Adv}_{\mathcal{B}_2}^{\text{snd}}(\lambda)) + \sum_{i=1}^{n} \mathbf{Adv}_{\mathcal{B}_{i,3}}^{\text{ddh}}(\lambda) \\
&+ \mathbf{Adv}_{\mathcal{B}_4}^{\text{ddh}}(\lambda)) - 1
\end{aligned}$$

for all $\lambda \in \mathbb{N}$, from which the theorem follows. To do this, we build $\mathcal{B}_0$, $\mathcal{B}_1$, $\mathcal{B}_2$, $\mathcal{B}_4$, and a family $\mathcal{B}_{i,3}$ such that

$$|\Pr[\mathsf{G}_{\mathcal{A}}^{\text{secrecy}}(\lambda)] - \Pr[\mathsf{G}_1^{\mathcal{A}}(\lambda)]| \le \mathbf{Adv}_{\mathcal{B}_0}^{\text{zk}}(\lambda) \tag{2}$$

$$|\Pr[\mathsf{G}_1^{\mathcal{A}}(\lambda)] - \Pr[\mathsf{G}_2^{\mathcal{A}}(\lambda)]| \le \mathbf{Adv}_{\mathcal{B}_1}^{\text{zk}}(\lambda) + \mathbf{Adv}_{\mathcal{B}_1}^{\text{snd}}(\lambda) \tag{3}$$

$$|\Pr[\mathsf{G}_2^{\mathcal{A}}(\lambda)] - \Pr[\mathsf{G}_3^{\mathcal{A}}(\lambda)]| \le \frac{n}{2}(\mathbf{Adv}_{\mathcal{B}_2}^{\text{ddh}}(\lambda) + \mathbf{Adv}_{\mathcal{B}_2}^{\text{snd}}(\lambda)) \tag{4}$$

$$|\Pr[\mathsf{G}_{i,3}^{\mathcal{A}}(\lambda)] - \Pr[\mathsf{G}_{i+1,3}^{\mathcal{A}}(\lambda)]| \le \mathbf{Adv}_{\mathcal{B}_{i,3}}^{\text{ddh}}(\lambda) \tag{5}$$

$$|\Pr[\mathsf{G}_4^{\mathcal{A}}(\lambda)] - \Pr[\mathsf{G}_5^{\mathcal{A}}(\lambda)]| \le \mathbf{Adv}_{\mathcal{B}_4}^{\text{ddh}}(\lambda) \tag{6}$$

$$\Pr[\mathsf{G}_5^{\mathcal{A}}(\lambda)] = 0 \tag{7}$$

```
$\underline{\text{MAIN } \mathsf{G}^{\text{secrecy}}_{\mathcal{A}}(\lambda)}$
1  $\mathsf{vote\_count} \leftarrow \vec{0}$
2  $b \xleftarrow{\$} \{0,1\}$
3  $(\mathsf{pk}_i, \mathsf{sk}_i) \xleftarrow{\$} \mathsf{KeyGen}(1^\lambda)\ \forall i \in [n]$
4  $b' \xleftarrow{\$} \mathcal{A}^{\mathrm{AP,HP,CP}}(1^\lambda, \mathsf{contracts}, \{\mathsf{pk}_i\}_i)$
5  return $(b' = b)$

$\underline{\mathrm{AP}.\mathsf{Vote1}(j, \mathsf{tx}_{\mathsf{vote1}})}$
6  $\mathsf{contracts}[j].\mathsf{Process\_Vote1}(\mathsf{tx}_{\mathsf{vote1}})$

$\underline{\mathrm{AP}.\mathsf{Vote2}(j, \mathsf{tx}_{\mathsf{vote2}})}$
7  if $(\mathsf{contracts}[j].\mathsf{vote\_flag} = 0)$ $\mathsf{contracts}[j].\mathsf{vote\_flag} \leftarrow 1$
8  $\mathsf{contracts}[j].\mathsf{Process\_Vote2}(\mathsf{tx}_{\mathsf{vote2}})$

$\underline{\mathrm{HP}.\mathsf{Vote1}(j, i, b_{\mathcal{A}})}$
9  if $(b_{\mathcal{A}} = 0)$ $\mathsf{vote\_count}[j] \mathrel{-}= 1$
10 else $\mathsf{vote\_count}[j] \mathrel{+}= 1$
11 $\mathsf{vote} \leftarrow (b_{\mathcal{A}}\ \mathbf{EQ}\ b)$
12 $x \xleftarrow{\$} \mathbb{F}$, $\boxed{\mathsf{G}_4 : x, \tilde{r} \xleftarrow{\$} \mathbb{F}}$
13 $c_0 \leftarrow g_0^x$
14 $c_1 \leftarrow g_1^{\mathsf{vote}} h_0^x$, $\boxed{\mathsf{G}_4 : c_1 \leftarrow g_1^{\mathsf{vote}} h_0^{\tilde{r}}}$
15 $\pi_1 \leftarrow \mathsf{Prove}(R_{\mathsf{Vote1}}, c_0, x)$, $\boxed{\mathsf{G}_1 : \pi_1 \leftarrow \mathsf{Sim}(R_{\mathsf{Vote1}}, c_0)}$
16 $\mathsf{tx}_{\mathsf{vote1}} \xleftarrow{\$} \mathsf{FormTx}(\mathsf{amt}_{\mathsf{dep}}, (c_0, c_1, \pi_1, \mathsf{wgt}))$
17 $\mathsf{contracts}[j].\mathsf{Process\_Vote1}(\mathsf{tx}_{\mathsf{vote1}})$

$\underline{\mathrm{HP}.\mathsf{Vote2}(j, i)}$
18 if $(\mathsf{vote\_count}[j] \neq 0)$ return 0
19 if $(\mathsf{contracts}[j].\mathsf{vote\_flag} = 0)$ $\mathsf{contracts}[j].\mathsf{vote\_flag} \leftarrow 1$
20 $s \xleftarrow{\$} \mathbb{F}$, $\boxed{\mathsf{G}_5 : s, \tilde{s} \xleftarrow{\$} \mathbb{F}}$
21 $Y \leftarrow \mathtt{voters}[\mathsf{pk}][Y]$
22 $c_2 \leftarrow g_1^{\mathsf{vote}} Y^x$, $\boxed{\mathsf{G}_3 : c_2 \leftarrow g_1^{\mathsf{vote}} T}$
23 $c_3 \leftarrow g_0^s$
24 $c_4 \leftarrow (h_0^{-1} Y)^x h_1^s$, $\boxed{\mathsf{G}_3 : c_4 \leftarrow (h_0^{-x} T) h_1^s}$,
   $\boxed{\mathsf{G}_4 : c_4 \leftarrow (h_0^{-\tilde{r}} T) h_1^s}$, $\boxed{\mathsf{G}_5 : c_4 \leftarrow (h_0^{-\tilde{r}} T) h_1^{\tilde{s}}}$
25 $\pi_2 \leftarrow \mathsf{Prove}(R_{\mathsf{Vote2}}, (Y, \{c_i\}_{i=0}^4), (\mathsf{vote}, x, s))$,
   $\boxed{\mathsf{G}_2 : \pi_2 \leftarrow \mathsf{Sim}(R_{\mathsf{Vote2}}, (Y, \{c_i\}_{i=0}^4))}$
26 $\mathsf{tx}_{\mathsf{vote2}} \xleftarrow{\$} \mathsf{FormTx}(0, (c_2, c_3, c_4, \pi_2))$
27 $\mathsf{contracts}[j].\mathsf{Process\_Vote2}(\mathsf{tx}_{\mathsf{vote2}})$
```

Fig. 8. The "unrolled" vote secrecy game and, in boxes, the changes introduced by our various game transitions. The value T used in lines 22 and 24 depends on the index of the participant, so we leave a definition of it until the relevant game transition.

Summaries of all of these games are provided in Figure 8, where $\mathsf{G}^{\mathcal{A}}_{0,3}(\lambda) = \mathsf{G}^{\mathcal{A}}_3(\lambda)$ and $\mathsf{G}^{\mathcal{A}}_{n,3}(\lambda) = \mathsf{G}^{\mathcal{A}}_4(\lambda)$. We then have that

$$
\begin{aligned}
&\mathbf{Adv}^{\text{secrecy}}_{\mathcal{A}}(\lambda)\\
&= 2\Pr[\mathsf{G}^{\text{secrecy}}_{\mathcal{A}}(\lambda)] - 1\\
&= 2(\Pr[\mathsf{G}^{\text{secrecy}}_{\mathcal{A}}(\lambda)] - \Pr[\mathsf{G}^{\mathcal{A}}_1(\lambda)] + \Pr[\mathsf{G}^{\mathcal{A}}_1(\lambda)]) - 1\\
&= 2(\mathbf{Adv}^{\text{zk}}_{\mathcal{B}_0}(\lambda) + \Pr[\mathsf{G}^{\mathcal{A}}_1(\lambda)] - \Pr[\mathsf{G}^{\mathcal{A}}_2(\lambda)] + \Pr[\mathsf{G}^{\mathcal{A}}_2(\lambda)]) - 1\\
&= 2(\mathbf{Adv}^{\text{zk}}_{\mathcal{B}_0}(\lambda) + \mathbf{Adv}^{\text{zk}}_{\mathcal{B}_1}(\lambda) + \mathbf{Adv}^{\text{snd}}_{\mathcal{B}_1}(\lambda) + \Pr[\mathsf{G}^{\mathcal{A}}_2(\lambda)]\\
&\qquad - \Pr[\mathsf{G}^{\mathcal{A}}_3(\lambda)] + \Pr[\mathsf{G}^{\mathcal{A}}_3(\lambda)]) - 1\\
&= 2(\mathbf{Adv}^{\text{zk}}_{\mathcal{B}_0}(\lambda) + \mathbf{Adv}^{\text{zk}}_{\mathcal{B}_1}(\lambda) + \mathbf{Adv}^{\text{snd}}_{\mathcal{B}_1}(\lambda) + n/2(\mathbf{Adv}^{\text{ddh}}_{\mathcal{B}_2}(\lambda)\\
&\qquad + \mathbf{Adv}^{\text{snd}}_{\mathcal{B}_2}(\lambda)) + \Pr[\mathsf{G}^{\mathcal{A}}_3(\lambda)] - \Pr[\mathsf{G}^{\mathcal{A}}_4(\lambda)] + \Pr[\mathsf{G}^{\mathcal{A}}_4(\lambda)]) - 1\\
&= 2(\mathbf{Adv}^{\text{zk}}_{\mathcal{B}_0}(\lambda) + \mathbf{Adv}^{\text{zk}}_{\mathcal{B}_1}(\lambda) + \mathbf{Adv}^{\text{snd}}_{\mathcal{B}_1}(\lambda) + n/2(\mathbf{Adv}^{\text{ddh}}_{\mathcal{B}_2}(\lambda)\\
&\qquad + \mathbf{Adv}^{\text{snd}}_{\mathcal{B}_2}(\lambda)) + \textstyle\sum_{i=1}^n \mathbf{Adv}^{\text{ddh}}_{\mathcal{B}_{i,3}}(\lambda) + \Pr[\mathsf{G}^{\mathcal{A}}_4(\lambda)]\\
&\qquad - \Pr[\mathsf{G}^{\mathcal{A}}_5(\lambda)] + \Pr[\mathsf{G}^{\mathcal{A}}_5(\lambda)]) - 1\\
&= 2(\mathbf{Adv}^{\text{zk}}_{\mathcal{B}_0}(\lambda) + \mathbf{Adv}^{\text{zk}}_{\mathcal{B}_1}(\lambda) + \mathbf{Adv}^{\text{snd}}_{\mathcal{B}_1}(\lambda) + n/2(\mathbf{Adv}^{\text{ddh}}_{\mathcal{B}_2}(\lambda)\\
&\qquad + \mathbf{Adv}^{\text{snd}}_{\mathcal{B}_2}(\lambda)) + \textstyle\sum_{i=1}^n \mathbf{Adv}^{\text{ddh}}_{\mathcal{B}_{i,3}}(\lambda) + \mathbf{Adv}^{\text{ddh}}_{\mathcal{B}_4}(\lambda)) - 1.
\end{aligned}
$$

Equation (2): $\mathsf{G}^{\text{secrecy}}_{\mathcal{A}}(\lambda)$ to $\mathsf{G}^{\mathcal{A}}_1(\lambda)$

Intuitively, this follows in a completely straightforward way from the zero-knowledge property of the underlying proofs (here a simple Schnorr proof). Formally, $\mathcal{B}_0$ behaves as follows (omitting all lines that they follow honestly):

```
$\underline{\mathcal{B}_0^{\mathcal{O}}(1^\lambda)}$
15 $\pi_1 \leftarrow \mathcal{O}(c_0, x)$
```

If $\mathcal{B}_0$ gets real proofs from $\mathcal{O}$ then this is identical to $\mathsf{G}^{\text{secrecy}}_{\mathcal{A}}(\lambda)$. If instead $\mathcal{B}_0$ gets simulated proofs from $\mathcal{O}$ then this is identical to $\mathsf{G}^{\mathcal{A}}_1(\lambda)$.

Equation (3): $\mathsf{G}^{\mathcal{A}}_1(\lambda)$ to $\mathsf{G}^{\mathcal{A}}_2(\lambda)$

Intuitively, this also follows from the zero-knowledge property of the proof for R', although here the simulator for R' needs to know the discrete logarithm of Y, so we first need to extract from the adversary's first-round votes to get their contributions. Formally, $\mathcal{B}_1$ behaves as follows (again, omitting all lines that they follow honestly):

```
$\underline{\mathcal{B}_1^{\mathcal{O}}(1^\lambda)}$
25 $\pi_2 \leftarrow \mathcal{O}((Y, \{c_i\}_{i=0}^4), (\mathsf{vote}, x, s))$
```

If $\mathcal{B}_1$ gets real proofs from $\mathcal{O}$ then this is identical to $\mathsf{G}^{\mathcal{A}}_1(\lambda)$. If instead $\mathcal{B}_1$ gets simulated proofs from $\mathcal{O}$ (which are produced with the help of the extractor defined below) then this is identical to $\mathsf{G}^{\mathcal{A}}_2(\lambda)$. In order to run the simulator, we must define an extractor $\mathcal{X}$ that outputs $x_1, \ldots, x_\ell$ such that the adversary's values satisfy $c_{1,0}, \ldots, c_{\ell,0} = g_0^{x_1}, \ldots, g_0^{x_\ell}$. While this would generally be quite difficult, we do not claim to achieve concurrent vote secrecy in our scheme, meaning we must restrict $j = 1$. In the first round, $\mathcal{A}$ must query its random oracle to produce proofs of knowledge of the discrete logarithm of each $c_{i,0}$ for participant i. When $\mathcal{A}$ queries the oracle, $\mathcal{X}$ returns truly random values. The extractor now rewinds $\mathcal{A}$ and runs it again on the same random coins, but programs the random oracle to return different random outputs on $\mathcal{A}$'s queries. For any c_0 values produced by honest participants, $\mathcal{A}$ cannot reuse their proofs of knowledge, because the contract does not accept first-round votes with equal c_0 values. Similarly, because we restrict to only one session, $\mathcal{A}$ cannot reuse its own proofs of knowledge from other sessions here, so all proofs of knowledge must have been produced using access to this random oracle (controlled by $\mathcal{X}$). Thus, by the 2-special soundness of the argument for R_{Vote1}, either $\mathcal{B}_2$ is able to output $\mathcal{A}$'s instance and proof in order to break knowledge soundness, or $\mathcal{X}$ is able to output valid $x_0, \ldots, x_\ell$. It can then provide these to the simulator to allow it to reconstruct the discrete logarithm of Y.

Equation (4): $\mathsf{G}_2^{\mathcal{A}}(\lambda)$ to $\mathsf{G}_3^{\mathcal{A}}(\lambda)$

Intuitively, we start by assuming that there are two honest participants, who are queried at indices i_1 and i_2 (if there are more, then we proceed in a pairwise fashion, which is why we end up with a looseness of $n/2$). We embed DDH challenge terms D_1 and D_2 into their respective c_0 values, and must now be able to form the corresponding c_2 values. While this makes it slightly more difficult to form Y^x, we can extract all of the adversary's contributions to Y, and then embed the remaining $g^{\pm x_{i_1} x_{i_2}}$ terms using the DDH challenge term C. If $C = g^{x_{i_1} x_{i_2}}$ then this is the honest term Y^x, and if it is random then this embeds some extra randomness into c_2. Crucially, because we are embedding the same randomness for both participants, it can still cancel and satisfy the self-tallying requirement. Formally, $\mathcal{B}_2$ behaves as follows:

```
$\mathcal{B}_2(g_0, D_{i_1}, D_{i_2}, C)$
$\eta_0, \eta_1 \xleftarrow{\$} \mathbb{F}$; $h_0, h_1 \leftarrow g_0^{\eta_0}, g_0^{\eta_1}$

AP.Vote1$(j, \mathsf{tx_{vote1}})$
$x_i \leftarrow \mathsf{Ext}(\pi_1)\ \forall i$
// at the end of round 1
$z_{i_1} \leftarrow \sum_{i=1}^{i_1-1} x_i - \sum_{i=i_1+1}^{i_2-1} x_i - \sum_{i=i_2+1}^{m} x_i$
$z_{i_2} \leftarrow \sum_{i=1}^{i_1-1} x_i + \sum_{i=i_1+1}^{i_2-1} x_i - \sum_{i=i_2+1}^{m} x_i$
$T_{i_1} \leftarrow C^{-1} \cdot D_{i_1}^{z_{i_1}}$
$T_{i_2} \leftarrow C \cdot D_{i_2}^{z_{i_2}}$

HP.Vote1$(j, i \in \{i_1, i_2\}, b_{\mathcal{A}})$
13  $c_0 \leftarrow D_i$
14  $c_1 \leftarrow g_1^{\mathsf{vote}_i} D_i^{\eta_0}$

HP.Vote2$(j, i \in \{i_1, i_2\})$
22  $c_2 \leftarrow g_1^{\mathsf{vote}_i} \cdot T_i$
24  $c_4 \leftarrow D_i^{-\eta_0} \cdot T_i \cdot h_1^{s_i}$
25  $\pi_2 \leftarrow \mathsf{Sim}^*((D_i, C, x' = \sum_{k=0,k\neq i_1,i_2}^{m} x_i, \eta_0, \eta_1),$
                                          $c_0, c_1, c_2, c_3, c_4)$
```

where the algorithm Sim^* is a modified simulation algorithm, needed because $\mathcal{B}_2$ does not know the contributions x_{i_1} and x_{i_2} to Y, that behaves as follows:

```
$\mathsf{Sim}^*((D_i, C, x' = \sum_{k=0,k\neq i_1,i_2}^{m} x_i, \eta_0, \eta_1), c_0, c_1, c_2, c_3, c_4)$
$(A_0, A_1, A_2, A_3, A_4, A_5) \leftarrow (c_0, c_1, c_2, c_1 c_2^{-1}, c_1 c_2^{-1} c_4, c_3)$
$(a_0, a_1, a_2, d_0, d_1, u, v, w) \xleftarrow{\$} \mathbb{F}$
$R_0 = g_0^{d_0} A_0^{-a_0}$; $R_1 = (g_0^{d_0} A_0^{-a_0})^{-\eta_0 + x'} (D_k)^{d_0} C$
$R_2 = h_1^{d_1} A_4^{-a_1}$; $R_3 = R_1^{\eta_1}$
$S_0 = g_1^u h_0^v A_1^{-a_2}$; $S_1 = g_1^u Y^v A_2^{-a_2}$
$T = h_0^w A_1^{u-a_2}$
program the random oracle to return $a_0, \ldots, a_2$
on input $(R_0, R_1, R_2, R_3, S_0, S_1, T)$
return $(R_0, R_1, R_2, R_3, S_0, S_1, T), (d_0, d_1, u, v, w)$
```

If $\mathcal{A}$ does not form its c_2 values correctly, then $\mathcal{B}_2$ can output its instance and proof to break knowledge soundness. If it does form them correctly, then we have that

$$\begin{aligned}\log_{g_0}\Big(\prod_{j\neq i_1,i_2} Y_j^{x_j}\Big) &= x_{i_1}(\mathsf{GAND} + \mathsf{G1L2} - \mathsf{LAND}) \\ &\quad + x_{i_2}(\mathsf{GAND} - \mathsf{G1L2} - \mathsf{LAND}) \\ &= (x_{i_1} + x_{i_2})(\mathsf{GAND} - \mathsf{LAND}) \\ &\quad + (x_{i_1} - x_{i_2})\mathsf{G1L2},\end{aligned}$$

where these values are defined as

$$\begin{aligned}\mathsf{LAND} &= \sum_{j<i_1 \wedge i_2} x_j \\ \mathsf{G1L2} &= \sum_{j<i_2, j>i_1} x_j \\ \mathsf{GAND} &= \sum_{j>i_1 \wedge i_2} x_j\end{aligned}$$

Using these definitions, we also have that $z_{i_1} = \mathsf{LAND} + \mathsf{G1L2} - \mathsf{GAND}$, and $z_{i_2} = \mathsf{LAND} - \mathsf{G1L2} - \mathsf{GAND}$. Combining this and calling c the discrete logarithm of C, we have

$$\begin{aligned}&\log_{g_0}(\prod_{j\neq i_1,i_2} Y_j^{x_j} \cdot T_{i_1} \cdot T_{i_2}) \\ &= (x_{i_1}+x_{i_2})(\mathsf{GAND}-\mathsf{LAND}) + (x_{i_1}-x_{i_2})\mathsf{G1L2} \\ &\quad + c + x_{i_1}z_{i_1} - c + x_{i_2}z_{i_2} \\ &= (x_{i_1}+x_{i_2})(\mathsf{GAND}-\mathsf{LAND}) + (x_{i_1}-x_{i_2})\mathsf{G1L2} \\ &\quad + x_{i_1}(\mathsf{LAND}+\mathsf{G1L2}-\mathsf{GAND}) \\ &\quad + x_{i_2}(\mathsf{LAND}-\mathsf{G1L2}-\mathsf{GAND}) \\ &= (x_{i_1}+x_{i_2})(\mathsf{GAND}-\mathsf{LAND}) + (x_{i_1}-x_{i_2})\mathsf{G1L2} \\ &\quad + (x_{i_1}+x_{i_2})(\mathsf{LAND}-\mathsf{GAND}) + (x_{i_2}-x_{i_1})\mathsf{G1L2} \\ &= 0,\end{aligned}$$

meaning self-tallying holds regardless of the value of C. Furthermore, if $C = g_0^{x_{i_1}x_{i_2}}$ then this is identical to $\mathsf{G}_2^{\mathcal{A}}(\lambda)$ with simulated proofs. This is clear for c_2, and for the proofs observe that (continuing to assume two honest participants, without loss of generality):

$$\begin{aligned}R_1 &= (g_0^{d_0}A_0^{-a_0})^{-\eta_0+\sum_{j\leq i_1} x_j - \sum_{i_1<j, j\neq i_2} x_j}(D_2)^{d_0}C^{-1} \\ &= (g_0^{d_0-a_0x_{i_1}})^{-\eta_0+\sum_{j\leq i_1} x_j - \sum_{i_1<j, j\neq i_2} x_j} g_0^{d_0x_{i_2}} g_0^{-x_{i_1}x_{i_2}} \\ &= (g_0^{d_0-ax_{i_1}})^{-\eta_0+\sum_{j\leq i_1} x_j - \sum_{i_1<j, j\neq i_2} x_j + x_{i_2}} \\ &= R_0^{-\eta_0+\sum_{j\leq i_1} x_j - \sum_{i_1<j} x_j}\end{aligned}$$

so R_1 is distributed identically to the simulated R_1. If instead $C = g_0^r$ for $r \xleftarrow{\$} \mathbb{F}$ then this is identical to $\mathsf{G}_3^{\mathcal{A}}(\lambda)$. We continue to use these values of $T_{i_1} = g_0^{x_{i_1}z_{i_1}-r}$ and $T_{i_2} = g_0^{x_{i_2}z_{i_2}+r}$ (such that $c_{i,2} = g_1^{\mathsf{vote}_i}T_i$) in future games.

Equation (5): $\mathsf{G}_3^{\mathcal{A}}(\lambda)$ to $\mathsf{G}_4^{\mathcal{A}}(\lambda)$

Intuitively, this game hop proceeds in a hybrid fashion, in which we switch for each participant from $c_1 = g_1^{\mathsf{vote}}h_0^x$ to $c_1 = g_1^{\mathsf{vote}}h_0^r$. While ordinarily it might be possible to do this all at once, we are again treating the value x as part of the DDH challenge, and in order to form c_2 for unknown x it is necessary to know all other contributions to Y. Formally, we define $\mathsf{G}_{k,3}^{\mathcal{A}}(\lambda)$ as the game in which the first $n-k$ honest participants are using the values of c_1 as in $\mathsf{G}_3^{\mathcal{A}}(\lambda)$, and the remaining k are using the values of c_1 as in $\mathsf{G}_4^{\mathcal{A}}(\lambda)$; it's then the case that $\mathsf{G}_{0,3}^{\mathcal{A}}(\lambda) = \mathsf{G}_3^{\mathcal{A}}(\lambda)$ and $\mathsf{G}_{n,3}^{\mathcal{A}}(\lambda) = \mathsf{G}_4^{\mathcal{A}}(\lambda)$. Each adversary $\mathcal{B}_{k,3}$ behaves as follows, using the definitions of T_i from the previous game:

```
    $\mathcal{B}_{k,3}(g_0, A_1, A_2, C)$
    $h_0 \leftarrow A_2$

    HP.Vote1$(j, i, b_{\mathcal{A}})$
    for $i \leq k$
13  $c_0 \leftarrow g_0^{x_i}$
14  $c_1 \leftarrow g_1^{\mathsf{vote}_i} A_2^{x_i}$
    for $i = k+1$
13  $c_0 \leftarrow A_1$
14  $c_1 \leftarrow g_1^{\mathsf{vote}_i} C$
    for $i > k+1$
13  $c_0 \leftarrow g_0^{x_i}$
14  $c_1 \leftarrow g_1^{\mathsf{vote}_i} h_0^{\tilde{r}_i}$

    HP.Vote2$(j, i)$
22  $c_2 \leftarrow g_1^{\mathsf{vote}_i} \cdot T_i$
    for $i \leq k$
24  $c_4 \leftarrow A_2^{-x_i} \cdot T_i \cdot h_1^{s_i}$
    for $i = k+1$
24  $c_4 \leftarrow C^{-1} \cdot T_i \cdot h_1^{s_i}$
    for $i > k+1$
24  $c_4 \leftarrow h_0^{-\tilde{r}_i} \cdot T_i \cdot h_1^{s_i}$
```

If $C = g_0^{x_{i_1}x_{i_2}}$ then this is identical to $\mathsf{G}_{k,3}^{\mathcal{A}}(\lambda)$. If instead $C = g_0^r$ for $r \xleftarrow{\$} \mathbb{F}$ then this is identical to $\mathsf{G}_{k+1,3}^{\mathcal{A}}(\lambda)$.

Equation (6): $\mathsf{G}_4^{\mathcal{A}}(\lambda)$ to $\mathsf{G}_5^{\mathcal{A}}(\lambda)$

Intuitively, in this game hop we decouple the randomness between c_3 and c_4. This is easier than it is to decouple the randomness between c_0 and c_1, because the commitment randomness is different from the value x (i.e., it is not used anywhere else). Formally, we use the generalized Decisional Diffie-Hellman assumption [3], which states that it is difficult to distinguish between $(g_1, \ldots, g_\ell, u_1, \ldots, u_\ell)$, for $g_1, \ldots, g_\ell, u_1, \ldots, u_\ell$ are uniformly distributed in $\mathbb{G}^{2\ell}$, and $(g_1, \ldots, g_\ell, g_1^r, \ldots, g_\ell^r)$ for $r \xleftarrow{\$} \mathbb{F}$. This is implied (tightly) by DDH, and for our purposes we use $\ell = n+1$. The adversary $\mathcal{B}_4$ behaves as follows:

```
$\mathcal{B}_4(g_1, \ldots, g_{n+1}, u_1, \ldots, u_{n+1})$
$g_0 \leftarrow g_{n+1}$, $h_1 \leftarrow u_{n+1}$

$\mathrm{HP}.\mathsf{Vote2}(j, i \in [n])$
$c_3 \leftarrow g_i$
$c_4 \leftarrow h_0^{-\tilde{r}_i} \cdot T_i \cdot u_i$
```

If $u_i = g_i^r$ for all i then this is identical to $\mathsf{G}_4^{\mathcal{A}}(\lambda)$, since for $g_i = g_0^{s_i}$ we have that $u_i = h_1^{s_i}$. If instead $u_i = g_0^{\tilde{s}_i}$ for $\tilde{s}_i \xleftarrow{\$} \mathbb{F}$ then this is identical to $\mathsf{G}_5^{\mathcal{A}}(\lambda)$.

Equation (7): $\mathsf{G}_5^{\mathcal{A}}(\lambda)$

Intuitively, each honest participant now uses random values in their computation of c_1 and c_4 that do not appear anywhere else, which completely hides all information there. We can thus focus our attention on c_2 and argue that, despite the correlated randomness across pairs of honest participants, the values are distributed identically whether $b = 0$ or $b = 1$.

More formally, assuming again a single honest pair of participants (and operating across honest pairs to generalize), we have for voter 1:

$$(c_0, c_1, c_2, c_3, c_4) = \\ (g_0^{x_1}, g_1^{\mathsf{vote}_1} h_0^{\tilde{r}_1}, g_1^{\mathsf{vote}_1} g_0^{-r+x_1 z_1}, g_0^{s_1}, h_0^{-\tilde{r}_1} g_0^{-r+x_1 z_1} h_1^{\tilde{s}_1}).$$

For voter 2, we have:

$$(c_0, c_1, c_2, c_3, c_4) = \\ (g_0^{x_2}, g_1^{\mathsf{vote}_2} h_0^{\tilde{r}_2}, g_1^{\mathsf{vote}_2} g_0^{r+x_2 z_2}, g_0^{s_2}, h_0^{-\tilde{r}_2} g_0^{r+x_2 z_2} h_1^{\tilde{s}_2}).$$

Since $\tilde{r}_1, \tilde{r}_2, \tilde{s}_1, \tilde{s}_2$ are random and appear in only a single term, the contents of the c_1 and c_4 elements are hidden. Thus, the adversary's goal is to distinguish between the c_2 elements in the case in which voter 1 voted for 0 (meaning voter 2 must have voted for 1), or voter 1 voted for 1 and voter 2 voted for 0. These two cases can be written as follows for $g_1 = g_0^{\gamma}$:

$$\text{Case 1:}\ \ (g_1, c_2, c_2') = (g_0^{\gamma}, g_0^{-r+x_1 z_1}, g_0^{\gamma+r+x_2 z_2})$$
$$\text{Case 2:}\ \ (g_1, c_2', c_2) = (g_0^{\gamma}, g_0^{\gamma-r+x_1 z_1}, g_0^{r+x_2 z_2})$$

Since the adversary knows z_1 and z_2, this amounts to distinguishing between the tuples

$$(g_0^{\gamma}, g_0^{-r}, g_0^{\gamma+r}) \quad \text{and} \quad (g_0^{\gamma}, g_0^{\gamma-r}, g_0^{r}).$$

Let $\rho = \gamma + r$. Then $-r = \gamma - \rho$, so the two cases have the exact same form, meaning the distributions are identical.

D Proof of Dispute Freeness (Theorem 5.3)

Let $\mathcal{A}$ be a PT adversary playing game $\mathsf{G}_{\mathcal{A}}^{\text{dispute}}(\lambda)$. We know that $\mathcal{A}$ wins if for some j, (1) $\mathsf{tally}_1 \neq \mathsf{tally}_2$, which we call E_1, (2) $\mathsf{tally}_1 \neq \mathsf{contracts.tally}$, which we call E_2, or (3) $\mathsf{tally}_2 \neq \mathsf{contracts.tally}$, which we call E_3. More precisely for how our construction operates, we use $\mathsf{contracts.tally} = \mathtt{tallyG}$. We build a PT adversary $\mathcal{B}$, which implicitly embeds the required extractor Ext, such that

$$\mathbf{Adv}_{\mathcal{A}}^{\text{dispute}}(\lambda) \leq \mathbf{Adv}_{\mathcal{B}}^{\text{snd}}(\lambda).$$

$\mathcal{B}$ behaves as follows:

```
$\mathcal{B}(1^\lambda)$
$Q \leftarrow \emptyset$
$k_0, k_1 \xleftarrow{\$} \mathbb{F}$; $h_0 \leftarrow g_0^{k_0}$, $h_1 \leftarrow g_0^{k_1}$

$\mathrm{AP}.\mathsf{Vote1}(j, \mathsf{tx}_{\mathsf{vote1}})$
$(c_0, c_1, \pi_1, \mathsf{wgt}) \leftarrow \mathsf{tx}_{\mathsf{vote1}}[\mathsf{data}]$
$V_1 \leftarrow c_1 \cdot c_0^{-k_0}$ // code for Ext
$Q[\mathsf{pk}] \leftarrow V_1$

$\mathrm{AP}.\mathsf{Vote2}(j, \mathsf{tx}_{\mathsf{vote2}})$
$(c_2, c_3, c_4, \pi_2) \leftarrow \mathsf{tx}_{\mathsf{vote2}}[\mathsf{data}]$
$\phi \leftarrow (Y, \{c_i\}_{i=0}^4)$
$b_\pi \leftarrow \mathsf{Verify}(R_{\mathsf{Vote2}}, \phi, \pi_2)$
$V_2 \leftarrow c_0^{-k_0} \cdot c_4^{-1} \cdot c_3^{k_1} \cdot c_2$ // code for Ext
if $(b_\pi \wedge V_2 \neq Q[\mathsf{pk}])$ return $(\phi, \pi_2)$ // E1
if $(b_\pi \wedge Q[\mathsf{pk}] \neq 1, g_1)$ return $(\phi, \pi_2)$ // E2
if $(b_\pi \wedge V_2 \neq 1, g_1)$ return $(\phi, \pi_2)$ // E3
```

In the setup, the values h_0 and h_1 are uniformly random, so are distributed identically to their values in the honest setup.

In the first round, $\mathcal{B}$ performs an ElGamal decryption to recover

$$\begin{aligned} V_1 &= c_1 \cdot c_0^{-k_0} \\ &= g_1^{\mathsf{vote}} \cdot h_0^{x} \cdot (g_0^{x})^{-k_0} \\ &= g_1^{\mathsf{vote}} \cdot g_0^{x k_0} \cdot g_0^{-x k_0} \\ &= g_1^{\mathsf{vote}} \end{aligned}$$

for $\mathsf{vote} \in \{0, 1\}$, assuming c_0 and c_1 were formed correctly. If they were not formed correctly, meaning $\mathcal{B}$ does not extract 1 or g_1, then $\phi \notin L_{\mathsf{Vote2}}$. $\mathcal{B}$ can thus output this instance and π_2 to break knowledge soundness (an extractor cannot return a witness if the instance is not in the language).

In the second round, $\mathcal{B}$ again performs an ElGamal decryption to first recover

$$\begin{aligned} c_4^{-1} \cdot c_3^{k_1} &= (h_0 Y^{-1})^x \cdot h_1^{-s} \cdot (g_0^s)^{k_1} \\ &= (h_0 Y^{-1})^x \cdot g_0^{-k_1 s} \cdot g_0^{k_1 s} \\ &= (h_0 Y^{-1})^x \end{aligned}$$

again assuming c_3 and c_4 were computed correctly. It then continues to compute

$$\begin{aligned} V_2 &= c_0^{-k_0} \cdot c_4^{-1} \cdot c_3^{k_1} \cdot c_2 \\ &= g_0^{-xk_0} \cdot (h_0 Y^{-1})^x \cdot g_1^{\mathsf{vote}} \cdot Y^x \\ &= g_0^{-xk_0} \cdot g_0^{xk_0} \cdot Y^{-x} \cdot g_1^{\mathsf{vote}} \cdot Y^x \\ &= g_1^{\mathsf{vote}} \end{aligned}$$

for $\mathsf{vote} \in \{0, 1\}$, assuming c_2 was computed correctly. If any of these values were not computed correctly, meaning $\mathcal{B}$ does not extract 1 or g_1, then again the instance is not in the language so $\mathcal{B}$ can output it and π_2 to break knowledge soundness.

If E_1 happens and $\mathsf{tally}_1 \neq \mathsf{tally}_2$, there must be at least one participant for whom $V_1 \neq V_2$, which means $\mathcal{B}$ succeeds. If E_2 happens and $\mathsf{tally}_1 \neq \mathtt{tallyG}$, then again there must be at least one participant such that $V_1 \neq 1, g_1$, which means $\mathcal{B}$ succeeds. Finally, if E_3 happens and $\mathsf{tally}_2 \neq \mathtt{tallyG}$, then again there must be at least one participant such that $V_2 \neq 1, g_1$, which means $\mathcal{B}$ succeeds. Thus $\mathcal{B}$ succeeds whenever $\mathcal{A}$ does.