

Kite: How to Delegate Voting Power Privately

Kamilla Nazirkhanova
Stanford University

X. Pilli Cruz-De Jesus
Stanford University

Vrushank Gunjur
Stanford University

Dan Boneh
Stanford University

ABSTRACT

Ensuring the privacy of votes in an election is crucial for the integrity of a democratic process. Often, voting power is delegated to representatives (e.g., in congress) who subsequently vote on behalf of voters on specific issues. This delegation model is also widely used in Decentralized Autonomous Organizations (DAOs). Although several existing voting systems used in DAOs support private voting, they only offer public delegation. In this paper, we introduce Kite, a new protocol that enables *private* delegation of voting power for DAO members. Voters can freely delegate, revoke, and re-delegate their power without revealing any information about who they delegated to. Even the delegate does not learn who delegated to them. The only information that is recorded publicly is that the voter delegated or re-delegated their vote to someone. Kite accommodates both public and private voting for the delegates themselves. We analyze the security of our protocol within the Universal Composability (UC) framework. We implement Kite as an extension to the existing Governor Bravo smart contract on the Ethereum blockchain, that is widely used for DAO governance. Furthermore, we provide an evaluation of our implementation that demonstrates the practicality of the protocol. The most expensive operation is delegation due to the required zero-knowledge proofs. On a consumer-grade laptop, delegation takes between 7 and 167 seconds depending on the requested level of privacy.

KEYWORDS

DAO, Private voting, Proxy Voting, Delegation Privacy

1 INTRODUCTION

Decentralized autonomous organizations (DAOs) consist of a loosely affiliated group of individuals who collectively oversee and manage a shared treasury. Anyone can submit a proposal, and the DAO members vote. If the proposal is accepted, it is executed by the smart contract that manages the DAO. The proliferation of DAOs has generated renewed interest in new voting mechanisms for DAOs [21], and has underscored the importance of privacy in voting.

The ability to participate privately in an election, without revealing one’s vote, is essential for a well-functioning democratic process. A recent example is illustrated in the voting procedure of the Nouns DAO [28]. This DAO, like many other DAOs, is using a voting system where every voter’s vote is visible for everyone to see. Participants noticed the following behavior:

“Nouners many times aren’t voting for what they believe is best. Instead, they feel trapped in quid pro quo voting, afraid that their vote could reflect poorly on their image, and/or affect the likelihood of getting their own proposals through. Conversely

it occurs that Nouners vote in favor or against a proposal based on how the proposer voted for their past proposals.”

As a result, the Nouns DAO is looking to transition to an end-to-end verifiable voting system where everyone can vote in private.

The research community has been exploring private digital voting systems for a long time, starting with the work of Chaum [14] in 1981. Some protocols are based on homomorphic encryption [6, Ch.3], some are based on mix nets [6, Ch.6], some are based on blind signatures [6, Ch.2], and some are based on other mechanisms. Modern protocols stress the notion of end-to-end verifiability, where every voter can verify that its vote was counted as cast [4, 5, 13]. We refer to [6] for a survey of the area. More recently, some papers study private voting in the context of blockchains [18, 26].

In a typical voting system, private or not, every voter casts a ballot, these ballots are then tabulated, and the final results are published. However, this is not how voting works in DAOs. Since members do not have the desire or ability to vote on every proposal, the two most widely used governance protocols on Ethereum — Compound’s *Governor Bravo* [1] and Open Zeppelin’s *Governor* [19] — support *proxy voting*. In proxy voting, a voter can optionally delegate their voting power to a delegate, who votes on proposals on behalf of the voter. These delegations are recorded publicly on chain. In addition, the delegate’s voting history is also recorded publicly on chain. The latter transparency allows a voter Alice to hold her delegate accountable for their voting record. In a liquid democracy [8], Alice can revoke her delegation at any time and delegate to someone else as often and as many times as she wants. Indeed, this logic is supported by DAO governance contracts.

Concretely, in Nouns a substantial voting power is held by delegates and plays a pivotal role in determining the outcomes of most proposals. From Nouns’ inception in August 2021 to January 2025, an average of 68% of votes were cast by delegates. This influence has consistently grown over time, with the average proposal in December 2024 seeing 78% of votes coming from delegates. Moreover, delegates command a larger portion of the voting power, accounting for 34.7%, in contrast to regular voters at 25.8%, while the remainder of the voting power falls under the control of the treasury [3].

These numbers suggest that addressing quid pro quo issues in DAO governance requires governance systems that support private delegation, and possibly private voting for delegates.

Our work. We design Kite, a voting system that supports *private* delegation for DAOs governance. Alice can delegate her voting power to a delegate David so that no one, not even David, will know that Alice delegated to David. Moreover, Alice can revoke her delegation at any time and re-delegate to someone else, without David’s knowledge. When the delegate David votes, Kite supports two options: either public voting, so that voters can hold David

accountable for his voting record, or fully private voting for delegates.

We assume the existence of a “computing bulletin board” accessible to all parties. This board allows parties to post and read messages, as well as perform computations. All messages are authenticated, potentially via signatures, and cannot be erased. Clearly, a secure blockchain, more specifically, a smart contract implemented on a secure blockchain, can act as such a bulletin board.

Kite has three types of participants: voters, delegates, and a tally committee that we also call a trusted authority. These participants interact with the voting system using the following functions: (the detailed implementation of these functions is described in Section 3).

Setup: initiated by the trusted authority. The trusted authority produces public parameters that are used in subsequent subprotocols and sets up the on-chain contract.

Delegate Registration/Unregistration: called by a voter/delegate who wishes to become a delegate/stop being a delegate, respectively. It is executed on the smart contract. It takes the voter’s/delegate’s address as input and updates its status to ‘delegate’/‘voter’.

Delegation/Undelegation: called by a voter who wants to delegate/undelegate their voting tokens to a delegate. Delegation takes the voter’s and delegate’s addresses. The on-chain function updates a public data structure, which reveals nothing about the delegate. Undelegation takes the voter’s address and their previous delegation identifier as input. Upon execution, the on-chain function updates the public data structure, indicating that the voter has undelegated their voting tokens. Nothing else is revealed. Note that there is no need for relays as the fact that voter delegated/undelegated is public while the delegate’s identity remains hidden.

Election Setup: called by a voter who wants to submit a proposal to a vote. It requires the voter’s address, election ID, and a description of the election as input. Upon execution, the on-chain function returns the election parameters, which then become publicly available to all participants.

Election Start: called by the election creator, a voter who previously initiated the election setup. It requires the voter’s address and the election ID as input. The on-chain function then returns a commitment to the token distribution as it stands at the start of the election. This is important because the token delegations might change during the election window, but the system uses the recorded delegations at the start of the election.

Voting: called by a delegate wishing to cast a vote. It requires the election ID, the delegate’s address, and their vote as input. Upon execution, the on-chain function returns updated election parameters, which include an updated encrypted tally.

Tally: called by the trusted authority at the end of the election. It requires the trusted authority’s secret key for the encryption scheme, the election ID and the encrypted tallies of the options as input. The function then decrypts the result and calls the on-chain function, which returns the tallies in the clear, thereby making the results public for everyone.

In Section 4 we analyze the security of Kite using the Universal Composability (UC) framework [11]. In particular, we use a variant

called (SUC) [12]. In Section 4 we first defined an ideal voting functionality. We then utilize the composition theorem to prove the security of our protocol in a hybrid settings where we rely on a provided computing bulletin board functionality, which we also formally define. In our setup, this bulletin board is implemented by a blockchain. Our formalism for the bulletin board simply abstracts the properties of the blockchain that are needed to prove security of our voting protocol.

Implementation. We developed a Solidity proof-of-concept implementation of the proposed protocol. Our implementation extends the Governor Bravo smart contract [1] to make it support private delegation with public voting. We designed our user interface to mimic that of Nouns DAO, as our protocol addresses the quid pro quo voting challenges that they face. By extending a standard governance contract, we expect that our implementation will be applicable to many DAOs.

In DAO governance, the voting power of each voter is determined by the number of voting tokens they own. To manage these voting tokens, we utilize an ERC-20 contract [15], a widely-used standard for fungible tokens on the Ethereum network. Importantly, Kite requires a mechanism that enables locking tokens, namely temporarily restricting them from being transferred or used. This feature is crucial in the voting scenario to prevent double voting; without it, Alice could potentially vote once, transfer her tokens to a new account, and vote again, effectively reusing the same voting power to vote twice.

Our implementation uses zero-knowledge SNARKS [7] and we provide implementations of all the necessary circuits. In Section 5 we describe the many techniques we used to optimize proving time and reduce on-chain verification gas costs. Our implementation uses the Noir zk-SNARK framework¹ as the underlying ZK system. Finally, in Section 6 we describe the performance of the system.

1.1 Related Work

Several works had previously studied proxy voting for general voting systems. To the best of our knowledge, existing work does not consider the specific challenges and opportunities that come up in the context of DAO governance.

A number of works design cryptographic proxy voting systems [22–24, 30] and discuss various security properties such as coercion-resistance and delegation privacy, in addition to the standard vote privacy, running tally privacy, robustness, and others.

Kulyk et al. [22] explored incorporating proxy voting into Helios [4], a well-known open-source web-based voting system designed for verifiable elections. This extension additionally supports private delegation. More specifically, a voter sends a delegation token to their chosen proxy over an anonymous channel. The proxy can later use that delegation token to cast a vote. Note that the proxy learns the fact it was delegated to but not the identity of the voter that delegated. In [23], Kulyk et al. addressed the challenge of building a coercion-resistant proxy voting scheme that additionally satisfies delegation privacy. In this scheme, delegation privacy is achieved by using delegation servers that facilitate communication between voters and proxies. This scheme also requires anonymous

¹<https://noir-lang.org/>

channels. In a follow-up work of Kulyk et al. [24], boardroom voting scheme [25] was extended to ensure delegation privacy. In this scheme, a voter selects a random field element to serve as a delegation token, then secret shares the Feldman commitment to that element among a set of proxies. Each proxy additionally receives a random value, but the proxy chosen for delegation receives the token in the clear, which it can verify against the later reconstructed Feldman commitment.

We note that, unlike [22, 23], Kite neither requires additional parties to facilitate delegation nor relies on anonymous channels. Moreover, unlike [24], our scheme does not require the delegator to communicate with every voter. Furthermore, in Kite, the delegate learns nothing, not even the fact that they were delegated to. To the best of our knowledge, this work is also the first to consider the blockchain settings and provide an implementation as a DAO governance contract.

2 PRELIMINARIES

In this section, we outline the cryptographic primitives used in our system.

2.1 Additively Homomorphic Encryption

For our purposes, we require an **asymmetric encryption scheme** Enc comprised of five algorithms (Enc.Gen, Enc.E, Enc.D, Enc.Add, Enc.Rerand) such that:

- $\text{Enc.Gen}(1^\lambda) \rightarrow (\text{pk}, \text{sk})$: generate a public and private key pair.
- $\text{Enc.E}(\text{pk}, m; r) \rightarrow ct$: encrypt a message m with public key pk with randomness r .
- $\text{Enc.D}(\text{sk}, ct) \rightarrow m$: decrypt ciphertext ct using secret key sk .
- $\text{Enc.Add}(\text{pk}, ct_1, ct_2) \rightarrow ct_+$: homomorphically add ciphertexts ct_1 and ct_2 .
- $\text{Enc.Rerand}(\text{pk}, ct, r') \rightarrow ct'$: re-randomize ct with new randomness r' .

We only require that the scheme be semantically secure against chosen plaintext attack, also known as CPA-secure [9, Ch. 5].

2.2 Digital Signature Scheme

A **digital signature scheme** Sig is a triple of algorithms (Sig.Gen, Sig.Sign, Sig.Verify) such that:

- $\text{Sig.Gen}(1^\lambda) \rightarrow (\text{pk}, \text{sk})$: generate a public and private key pair.
- $\text{Sig.Sign}(\text{sk}, m) \rightarrow \sigma$: sign message m with secret key sk .
- $\text{Sig.Verify}(\text{pk}, m, \sigma) \rightarrow b$: verify signature σ .

We require that the scheme be existentially unforgeable under a chosen message attack [9, Ch. 13].

2.3 Hash Functions and Merkle Trees

A **hash function** H with output length l is a pair of algorithms (H.Gen, H) such that:

- $\text{H.Gen}(1^\lambda)$: generate key s .
- $\text{H}(s, x)$: output a string $x' \in \{0, 1\}^l$.

We require that the hash function is collision resistant, meaning it is infeasible to find two different inputs that produce the same

hash value [9, Ch. 8]. Additionally, we will use hash functions to construct hash trees, also known as Merkle trees [9, Ch. 8].

A Merkle Tree, denoted as MT, consists of three algorithms (MT.GetRoot, MT.GetProof, MT.Verify) such that:

- $\text{MT.GetRoot}(T) \rightarrow R$: compute the root R of the tree T
- $\text{MT.GetProof}(T, i) \rightarrow \pi_i$: compute the proof π_i for the leaf with index i
- $\text{MT.Verify}(T[i], i, \pi_i, R) \rightarrow b$: verify the proof π_i for the leaf $T[i]$ with index i against the root R

2.4 Succinct Non-interactive Zero-Knowledge Arguments

A **Succinct Non-interactive Zero-Knowledge Argument of Knowledge** for a relation \mathcal{R} consists of three algorithms (\mathcal{G} , zkProve, zkVerify) such that:

- $\mathcal{G}(1^\lambda) \rightarrow \text{priv}, \text{vgrs}$: generate a public verifier-generated reference string and corresponding private verification coins.
- $\text{zkProve}(R, w, \text{vgrs}) \rightarrow \pi$: generates a proof π for a statement R given a witness w .
- $\text{zkVerify}(\text{priv}, R, \pi) \rightarrow b$: verifies the validity of proof π .

We require it to be complete, succinct, and zero-knowledge (as defined in Appendix A). Importantly, we require it to be knowledge-sound, however, we need a straightline extractor [16], as we further prove the security of our protocol in the universal composability framework.

In the subsequent section, public and private string inputs are omitted in the verifier and prover algorithms. Throughout our voting protocol, we make use of zero-knowledge proofs for different relations. For brevity, we specify the relation as a subscript.

3 PROTOCOL

In Kite, we identify three types of entities: a set of voters p_1, p_2, \dots, p_n , a trusted authority TA, and an on-chain contract. The protocol is designed to allow voters to either vote directly or delegate their voting power. TA is trusted to tally the election results correctly and not reveal them prematurely. We discuss possible relaxations to the trust in the TA in Section 5.

Let us begin with a high-level overview of the design of the delegation and voting protocols. The voting power of each voter is indicated by the number of governance ERC20 tokens that they own. A voter who wishes to vote on proposals must register as a delegate by calling a corresponding function at the on-chain contract. The contract operates on a secure blockchain and is responsible for tracking all participants, their status (either as voters or delegates), and the encrypted voting power of every delegate. We consider multiple-choice voting (specifically, two options in the following example) but also discuss how to adapt our protocol for ranked-choice voting.

Suppose there are only three voters, denoted p_1, p_2 , and p_3 , each holding t_1, t_2 , and t_3 tokens respectively. If p_2 and p_3 are registered as delegates, the on-chain contract would store the delegate's voting power list as $(\text{Enc}(0), \text{Enc}(t_2), \text{Enc}(t_3))$, using an additively homomorphic encryption scheme. When voter p_i decides to delegate their voting tokens to another voter p_j , voter i creates a vector of all zeros except for the entry at index j , which is set to

the number of voting tokens held by p_i . This vector is then encrypted using the homomorphic encryption scheme and posted to the blockchain. The on-chain contract homomorphically adds this encrypted vector to the current list of delegate voting powers. Continuing with our example, if p_1 delegates to p_3 , they would post $(\text{Enc}(0), \text{Enc}(0), \text{Enc}(t_1))$ to the contract. The contract then updates the delegates' voting power list to $(\text{Enc}(0), \text{Enc}(t_2), \text{Enc}(t_3 + t_1))$ using the additive homomorphism. Additionally, it marks p_1 as a delegator and locks their tokens, preventing them from voting directly with their tokens.

In Kite, both public and private voting scenarios are supported. Let us first describe public voting. For example, if voter p_3 decides to vote for option 1, they can submit their vote openly. The on-chain contract then adds their encrypted total voting power, which is $\text{Enc}(t_3 + t_1)$, into the tally for option 1 using the additive homomorphism.

The private voting setting requires a different approach. Here, a voter must submit an encrypted vote count for each option. Specifically, when p_3 votes for option 1, it submits encryptions of zero for all options except the first. However, since p_3 's encrypted total voting power $\text{Enc}(t_3 + t_1)$ is already accessible on the blockchain, they cannot simply send this for option 1, as it would compromise the privacy of their vote. Instead, p_3 re-randomizes their encrypted voting power before posting it for option 1. If there are two voting options available (yes or no), p_3 casts their vote for option 1 by submitting $(\text{Rerand}(\text{Enc}(t_3 + t_1)), \text{Enc}(0))$. Now CPA-security of the encryption scheme ensures vote privacy.

At the end of the election, TA decrypts the final tally for each option and publishes it on the blockchain.

The high-level description so far omits a crucial detail: every encryption sent by voters must be paired with a proof of correctness. Otherwise, a malicious voter might submit an ill-formed ciphertext, potentially affecting the election outcome. Kite utilizes two types of proofs: a proof of correct delegation and a proof of correct vote (for the private setting).

In the next few subsections we walk through all the subprotocols of Kite. Along the way we describe the relations that are used in our zero-knowledge proofs to prove that all posted data is well-formed. Glossary 1 provided a list of all the parameters used.

REMARK. *Note that Kite can be modified to support ranked-choice voting. The public voting scenario is trivial, so we focus on private voting. Assume there are n total voting options and m is the number of options a voter can rank on their ballot. We maintain a tally vector with $n \cdot m$ entries, each corresponding to an option-rank pair. For example, in the case of $n = 5$ and $m = 3$, if a voter ranks options as $(1, 4, 5)$, they must submit an encryption vector where the entries corresponding to the pairs $(1, 1)$, $(4, 2)$, and $(5, 3)$ contain the encrypted voting power, while all other entries are zero.*

3.1 Setup

In this subprotocol (illustrated in Alg.1), TA begins by generating public keys for the encryption and signature schemes. The authority then creates L_T , list of the number of voting tokens held by each eligible voter. This list is transformed into a Merkle tree, and its root R_T is signed by TA. This step is crucial as participants, at various points, need to prove their token holdings. Verifying this

Global contract parameters	
L_{eid}	List of election identifiers
t_v	Number of voting tokens owned by v
L_T	List of the number of tokens owned by eligible voters
R_T	Merkle tree root of L_T
L_d	List of delegates' encrypted voting power
L_{did}	List of delegation identifiers
R_{eid}	Merkle tree root of L_d at the beginning of election eid
lock	Lock map
σ_{TA}	Trusted authority's signature on R_T
Election parameters	
eid	Election identifier
vote	Vote map
$desc_{eid}$	Description of election eid
E_i^{eid}	Encryption of the number of votes for option i in election eid
D_i^{eid}	The number of votes for option i in election eid
Other notation	
pk_P^Q, sk_P^Q	Public and secret keys of party Q for primitive P

Table 1: Glossary

through Merkle tree and signature checks simplifies the process, as it avoids the need for direct verification of the blockchain state and computation of balances. TA is also tasked with updating the Merkle tree root in response to any changes in the token list. Following these steps, the trusted authority proceeds with the on-chain contract setup, executing a series of initializations. Once finished, the contract logs (writes on the blockchain) the parameters.

Algorithm 1 Setup

```

1: function AUTHORITY SETUP( $L_T$ )                                // called by the trusted authority TA
2:    $pk_{Enc}^{TA}, sk_{Enc}^{TA} \leftarrow \text{Enc.Gen}(1^\lambda)$ 
3:    $pk_{Sig}^{TA}, sk_{Sig}^{TA} \leftarrow \text{Sig.Gen}(1^\lambda)$ 
4:    $R_T \leftarrow \text{MT.GetRoot}(L_T)$ 
5:    $\sigma_T \leftarrow \text{Sig.Sign}(sk_{Sig}^{TA}, R_T)$ 
6:    $L_T, R_T, L_{eid}, L_d, L_{did}, \text{lock}, \text{active} \leftarrow \text{ON CHAIN SETUP}(pk_{Enc}^{TA}, pk_{Sig}^{TA}, L_T, R_T, \sigma_{TA})$ 
7:   store  $sk_{Enc}^{TA}, sk_{Sig}^{TA}$ 
8: end function
9: function ON CHAIN SETUP( $pk_{Enc}^{TA}, pk_{Sig}^{TA}, L_T, R_T, \sigma_{TA}$ )
10:   $L_{eid} \leftarrow \emptyset$                                            // list of election identifiers
11:   $L_{did} \leftarrow \emptyset$                                        // list of delegation identifiers
12:  lock  $\leftarrow \emptyset$                                          // lock map
13:  active  $\leftarrow \emptyset$                                        // active delegate map
14:   $L_d \leftarrow \emptyset$                                          // list of delegates voting power, all-zero vector
15:  if  $\text{Sig.Verify}(pk_{Sig}^{TA}, R_T, \sigma_T) = 1$  then
16:     $R_T \leftarrow R_T$                                            // save  $R_T$  in the contract state
17:    log  $L_T, R_T, L_{eid}, L_d, L_{did}, \text{lock}, \text{active}$ 
18:  else
19:    abort
20:  end if
21: end function
22: function VOTER SETUP( $L_T$ )                                     // called by  $p_i$ 
23:   $t_{p_i} \leftarrow L_T[p_i]$                                      // store the number of tokens  $p_i$  has
24:   $ct \leftarrow \emptyset$                                          // initialize delegation vector
25:  store  $t_{p_i}, ct$ 
26: end function

```

3.2 Delegate Registration and Unregistration

When a voter decides to become a delegate, they call ON CHAIN DELEGATE REGISTRATION, Alg. 2. This triggers the on-chain contract

to mark the voter as an active delegate, simultaneously locking their funds. Again, the locking step ensures the voter cannot reuse the same tokens in the voting process. Additionally, the contract updates the list of delegates' encrypted voting power. It does this by adding the encryption of the voter's power into the corresponding cell in the list L_d .

To unregister as a delegate, the protocol follows a reverse process and calls ON CHAIN DELEGATE UNREGISTRATION, Alg. 3. The on-chain contract marks the delegate as inactive and unlocks their funds. Furthermore, the contract updates the list of delegates' encrypted voting power by subtracting the encryption of the delegate's voting power from the corresponding cell of L_d .

There are four possible state combinations a voter can encounter, stemming from two variations each in active state and locking state:

- (1) unlocked, inactive – voter that has not delegated
- (2) unlocked, active – impossible
- (3) locked, inactive – voter that delegated their tokens
- (4) locked, active – delegate

Algorithm 2 Delegate Registration

```

1: function DELEGATE REGISTRATION( $p_i$ ) // called by  $p_i$ 
2:    $p_i, \text{lock}[p_i], \text{active}[p_i], L_d[p_i] \leftarrow \text{ON CHAIN DELEGATE REGISTRATION}(p_i)$ 
3: end function
4: function ON CHAIN DELEGATE REGISTRATION( $p_i$ )
5:   if  $\text{active}[p_i] = 0 \wedge \text{lock}[p_i] = 0$  then
6:      $\text{lock}(p_i)$  // lock tokens of msg.sender in ERC20 contract
7:      $\text{lock}[p_i] = 1$  // update lock map
8:      $\text{active}[p_i] = 1$  // update active delegate map
9:      $t_{p_i} \leftarrow L_T[p_i]$ 
10:     $L_d[p_i] \leftarrow \text{Enc.E}(\text{pk}_{\text{Enc}}^{\text{TA}}, t_{p_i}; 0)$  // update  $p_i$ 's voting power in the delegate list
11:   end if
12:   log  $p_i, \text{lock}[p_i], \text{active}[p_i], L_d[p_i]$ 
13: end function

```

Algorithm 3 Delegate Unregistration

```

1: function DELEGATE UNREGISTRATION( $p_i$ ) // called by  $p_i$ 
2:    $p_i, \text{lock}[p_i], \text{active}[p_i], L_d[p_i] \leftarrow \text{ON CHAIN DELEGATE UNREGISTRATION}(p_i)$ 
3: end function
4: function ON CHAIN DELEGATE UNREGISTRATION( $p_i$ )
5:   if  $\text{active}[p_i] = 1 \wedge \text{lock}[p_i] = 1$  then
6:      $\text{unlock}(p_i)$  // unlock tokens of msg.sender in ERC20 contract
7:      $\text{lock}[p_i] = 0$  // update lock map
8:      $\text{active}[p_i] = 0$  // update active delegate map
9:      $t_{p_i} \leftarrow L_T[p_i]$ 
10:     $e \leftarrow \text{Enc.E}(\text{pk}_{\text{Enc}}^{\text{TA}}, -t_{p_i}; 0)$ 
11:     $L_d[p_i] \leftarrow \text{Enc.Add}(\text{pk}_{\text{Enc}}^{\text{TA}}, L_d[p_i], e)$ 
12:   end if
13:   log  $p_i, \text{lock}[p_i], \text{active}[p_i], L_d[p_i]$ 
14: end function

```

3.3 Delegation and Undelegation

As previously explained, in the delegation subprotocol, Alg. 4, if a voter p_i wishes to delegate their voting power t_{p_i} to a delegate p_j , they create an encrypted vector ct . It is an encryption of all-zero vector, except for the entry corresponding to p_j . Additionally, the voter is generates a proof that ct is well-formed and that they indeed possess t_{p_i} tokens. This involves a Merkle inclusion proof in the tree with the root R_T , which was earlier uploaded by TA.

The on-chain contract then verifies that the voter's tokens are not locked and checks the proof of correctness. If these checks are passed, it locks the tokens and performs a homomorphic addition of ct to the existing list of encrypted powers of the delegates. This addition ensures that the total power of p_j is accurately updated to include t_{p_i} . Additionally, it computes a commitment to ct , using a hash function. This hash, or commitment, is then added to the list of delegation identifiers. This step is crucial for the undelegation process that may follow later.

During the undelegation process, Alg. 5, a voter essentially reverses the actions taken in the delegation phase. This includes unlocking their tokens and deducting their contributed voting power from the delegate's total. The key to executing the second part correctly lies in the use of delegation identifiers, established earlier.

For undelegation, the voter is required to present the original delegation vector, ct , that they used for delegation. The contract then verifies whether the hash of this is stored in L_{did} . Finding the hash indicates the voter had indeed delegated their tokens using this specific ct . Once confirmed, the contract proceeds to homomorphically subtract ct from L_d and also removes the corresponding delegation identifier $H(ct)$ from L_{did} .

The relation for the corresponding zero-knowledge proof is defined as follows:

$$\mathcal{R}_{\text{del}} := \left\{ (p_j, r), (\text{pk}_{\text{Enc.E}}^{\text{TA}}, ct, t_{p_i}, R_T, \pi_{p_i}, p_i) \mid \begin{aligned} &\text{MT.Verify}(t_{p_i}, p_i, \pi_{p_i}, R_T) = 1 \wedge \\ &ct[p_j] = \text{Enc.E}(\text{pk}_{\text{Enc}}^{\text{TA}}, t_{p_i}; r_j) \wedge \\ &\forall i \neq p_j : ct[i] = \text{Enc.E}(\text{pk}_{\text{Enc}}^{\text{TA}}, 0; r_i) \end{aligned} \right\} \quad (1)$$

In the subsequent Table 2, we delineate the witness and public parameters. Note that the corresponding circuit is linear in the number of delegates. One possible optimization is to select a random subset of delegates of smaller size, which we call the *anonymity set*, and construct ct only for that set. We discuss this in more detail in Section 5.1.

Witness	
$p_j \in [m]$	Delegate's address and an index, where m is the number of delegates
$r \in \mathbb{Z}_q^m$	A randomness vector used in encryption
Public Statement	
$\text{pk}_{\text{Enc}}^{\text{TA}} \in \mathbb{G}$	The encryption public key of the trusted authority
$ct \in \mathbb{G}^m$	A delegation vector of encryptions of size m
$t_{p_i} \in \mathbb{Z}_q$	The voting power of a voter p_i
$R_T \in \mathbb{F}_p$	The root of the Merkle tree of voting powers
$\pi_{p_i} \in \mathbb{F}_p^{\log n}$	A Merkle proof for the element with index p_i in the Merkle tree with n elements
$p_i \in [n]$	Voter's address and an index, where m is the number of voters

Table 2: Witness and Public Statement for \mathcal{R}_{del}

3.4 Election Setup and Election Start

Our election subprotocol is structured into two phases. The first phase begins with the election setup, Alg. 6, initiated when the election details are made available on-chain. The second phase

Algorithm 4 Delegation

```

1: function DELEGATION( $p_i, p_j$ )                                // called by  $p_i$ 
2:    $r \xleftarrow{\$} \mathbb{Z}_q^m$                                               //  $m$  is the number of registered delegates
3:   for  $i \in \text{len}(L_d) \wedge i \neq p_j$  do
4:      $ct[i] \leftarrow \text{Enc.E}(\text{pk}_{\text{Enc}}^{\text{TA}}, 0; r_i)$ 
5:   end for
6:    $ct[p_j] \leftarrow \text{Enc.E}(\text{pk}_{\text{Enc}}^{\text{TA}}, t_{p_j}; r_j)$ 
7:    $\pi_{p_i} \leftarrow \text{MT.GetProof}(L_T, p_i)$                     // get a Merkle proof for a leaf with index  $p_i$ 
8:    $\pi \leftarrow \text{zkProve}_{R_{\text{del}}}((\text{pk}_{\text{Enc.E}}^{\text{TA}}, ct, t_{p_i}, R_T, \pi_{p_i}, p_i), (p_j, r))$ 
9:    $p_i, \text{lock}[p_i], L_d, L_{\text{did}}[p_i] \leftarrow \text{ON CHAIN DELEGATION}(p_i, ct, \pi, \pi_{p_i})$ 
10: end function
11: function ON CHAIN DELEGATION( $p_i, ct, \pi, \pi_{p_i}$ )
12:   if  $\text{lock}[p_i] = 0$  then                                     // verify if  $p_i$ 's tokens are not locked
13:     if  $\text{zkVerify}_{R_{\text{del}}}((\text{pk}_{\text{Enc.E}}^{\text{TA}}, ct, t_{p_i}, R_T, \pi_{p_i}, v), \pi) = 1$  then
14:        $\text{lock}(p_i)$                                            // lock tokens of msg.sender in ERC20 contract
15:        $\text{lock}[p_i] = 1$                                        // update lock map
16:        $L_d \leftarrow \text{Enc.Add}(\text{pk}_{\text{Enc}}^{\text{TA}}, L_d, ct)$           // homomorphic addition of encrypted vectors
17:        $L_{\text{did}}[p_i] \leftarrow H(ct)$                         // update the list of delegation identifiers
18:     end if
19:   end if
20:    $\log p_i, \text{lock}[p_i], L_d, L_{\text{did}}[p_i]$ 
21: end function

```

Algorithm 5 Undelegation

```

1: function UNDELEGATION( $p_i, ct$ )                                // called by  $p_i$ 
2:    $p_i, \text{lock}[p_i], L_d, L_{\text{did}}[p_i] \leftarrow \text{ON CHAIN UNDELEGATION}(p_i, ct)$ 
3: end function
4: function ON CHAIN UNDELEGATION( $p_i, ct$ )
5:   if  $\text{lock}[p_i] = 1$  then
6:     if  $L_{\text{did}}[p_i] = H(ct)$  then
7:        $\text{unlock}(p_i)$                                            // unlock tokens of msg.sender in ERC20 contract
8:        $\text{lock}[p_i] = 0$                                        // update lock map
9:        $L_d \leftarrow \text{Enc.Add}(\text{pk}_{\text{Enc}}^{\text{TA}}, L_d, -ct)$           // homomorphic subtraction of encrypted vectors
10:       $L_{\text{did}}[p_i] \leftarrow 0$                              // update the list of delegation identifiers
11:    end if
12:  end if
13:   $\log p_i, \text{lock}[p_i], L_d, L_{\text{did}}[p_i]$ 
14: end function

```

begins with the initiation of the voting process, Alg. 7. Any voter with the intention to create an election can trigger the setup phase. They are required to provide a unique election identifier and a description of the election. The contract, in response, initializes counters for each election option. In our example, we have three options: 'Yes', 'No', and 'Abstain', with their counters initialized to the encryption of zeros. Additionally, the contract establishes a vote map to track participants voting activity, preventing double voting. It also initializes a snapshot of the voting power, which is crucial for the next phase. This phase can only be initiated by the creator of the election. At this point, the contract captures a snapshot of the current root of the voting power list – R_{eid} . This snapshot is necessary because voting power can fluctuate if voters transfer tokens. By capturing a snapshot of the voting power distribution and delegates' total power at the beginning of the election, we maintain a consistent reference point for the duration of the voting period. While the real-time distribution may vary due to ongoing token transactions, the voting is based on this initial snapshot. This strategy eliminates the necessity of disallowing token distribution changes throughout the election. This measure would be unrealistic, especially when several elections are running at the same time. Our

method ensures that voters can freely transfer their tokens without affecting the tally correctness.

Algorithm 6 Election Setup

```

1: function ELECTION SETUP( $p_i, \text{eid}, \text{desc}$ )                      // called by  $p_i$ 
2:    $L_{\text{eid}}, E^{\text{eid}}, \text{vote}_{\text{eid}}, R_{\text{eid}}, c_{\text{eid}} \leftarrow \text{ON CHAIN ELECTION SETUP}(p_i, \text{eid}, \text{desc})$ 
3: end function
4: function ON CHAIN ELECTION SETUP( $v, \text{eid}, \text{desc}$ )
5:   if  $\text{eid} \notin L_{\text{eid}}$  then
6:      $L_{\text{eid}} \leftarrow L_{\text{eid}} \cup \text{eid}$                         // update the list of election identifiers
7:      $\text{desc}_{\text{eid}} \leftarrow \text{desc}$ 
8:      $c_{\text{f0}} \leftarrow \text{Enc.E}(\text{pk}_{\text{Enc}}^{\text{TA}}, 0; 0)$ 
9:      $E^{\text{eid}} \leftarrow c_{\text{f0}}$                                    // initialize the voting counters
10:     $\text{vote}_{\text{eid}} \leftarrow \emptyset$                              // initialize the election voting map
11:     $R_{\text{eid}} \leftarrow 0$                                        // initialize the voting power snapshot
12:     $c_{\text{eid}} \leftarrow p_i$                                    // save the election creator
13:     $\log L_{\text{eid}}, E^{\text{eid}}, \text{vote}_{\text{eid}}, R_{\text{eid}}, c_{\text{eid}}$ 
14:  else
15:    abort
16:  end if
17: end function

```

Algorithm 7 Election Start

```

1: function CREATOR ELECTION START( $p_i, \text{eid}$ )                    // called by  $p_i$ 
2:    $R_{\text{eid}} \leftarrow \text{ON CHAIN ELECTION START}(p_i, \text{eid})$ 
3:    $L_d^{\text{eid}} \leftarrow L_d$ 
4:   return  $L_d^{\text{eid}}$ 
5: end function
6: function ON CHAIN ELECTION START( $p_i, \text{eid}$ )
7:   if  $c_{\text{eid}} = p_i$  then                                     // check if  $p_i$  is the election creator
8:      $R_{\text{eid}} \leftarrow \text{MT.GetRoot}(L_d)$                      // the voting power snapshot
9:   end if
10:   $\log R_{\text{eid}}$ 
11: end function

```

3.5 Voting

For clarity, we assume there are three vote options – Yes, No, and Abstain. However, any number of choices can be supported. In the public voting scenario, Alg. 8, a delegate who wants to cast their vote must send it directly to the on-chain contract. Along with their vote, the delegate provides a Merkle proof for their encrypted voting power, allowing the on-chain contract to verify it against the Merkle tree root R_{eid} , generated at the start of the election. The on-chain contract then performs several checks: it verifies that the delegate is active, confirms that they haven't voted previously, and validates the Merkle proof. Only after successfully passing these checks does the contract homomorphically add the delegate's encrypted voting power to the tally for the selected option.

In the private voting scenario, Alg. 9, the process for a delegate to cast their vote is more nuanced. The delegate must submit an encrypted vote for each option, a vector E . The key difference is that for the option they select, they use the encryption of their voting power, while for all other options, they submit an encryption of zeros. To maintain the privacy of their vote, the delegate re-randomizes the encryption of their voting power, as it is public. Additionally, the delegate must provide a proof of correctness for these encrypted votes.

The on-chain contract then verifies the proof and checks if the delegate has not voted previously. After these checks are passed,

the contract homomorphically adds the delegate's encrypted votes to the respective tallies for each option.

The relation used in the proof:

$$\mathcal{R}_{\text{vote}} := \left\{ (v, r), (\text{pk}_{\text{Enc}}^{\text{TA}}, L_d^{\text{eid}}[p_i], E, p_i, R_{\text{eid}}, \pi_{p_i}) \mid \right. \\ E_v = \text{Enc.Rerand}(\text{pk}_{\text{Enc}}^{\text{TA}}, L_d^{\text{eid}}[p_i], r_v) \wedge \\ \forall j \neq v : E_j \leftarrow \text{Enc.E}(\text{pk}_{\text{Enc}}^{\text{TA}}, 0; r_j) \wedge \\ \left. \text{MT.Verify}(L_d^{\text{eid}}[p_i], p_i, \pi_{p_i}, R_{\text{eid}}) = 1 \right\} \quad (2)$$

Table 3 summarizes the witness and public statement for the relation.

Witness	
$v \in [3]$	A vote
$r \in \mathbb{Z}_q^3$	The randomness vector used in encryption
Public Statement	
$\text{pk}_{\text{Enc}}^{\text{TA}} \in \mathbb{G}$	The encryption public key of the trusted authority
$L_d^{\text{eid}}[p_i] \in \mathbb{G}$	The encrypted voting power of delegate p_i
$E \in \mathbb{G}^3$	The encrypted vector of votes
$p_i \in [m]$	Delegate's address and an index, where m is the number of registered delegates
$R_{\text{eid}} \in \mathbb{F}_p$	The root of the Merkle tree of encrypted voting powers for election eid
$\pi_{p_i} \in \mathbb{F}_p^{\log m}$	A Merkle proof for the element in the Merkle tree with index p_i

Table 3: Witness and Public Statement for $\mathcal{R}_{\text{vote}}$

Algorithm 8 Public Voting

```

1: function VOTING(eid,  $L_d^{\text{eid}}, p_i, v$ ) // called by  $p_i$ 
2:   if  $v \in \{\text{yes}, \text{no}, \text{abstain}\}$  then
3:      $\pi_{p_i} \leftarrow \text{MT.GetProof}(L_d^{\text{eid}}, p_i)$ 
4:      $\text{vote}[p_i], E_v^{\text{eid}} \leftarrow \text{ON CHAIN VOTING}(\text{eid}, d, v, \pi_{p_i}, L_d^{\text{eid}}[p_i])$ 
5:   else
6:     abort
7:   end if
8: end function
9: function ON CHAIN VOTING(eid,  $p_i, v, \pi_{p_i}, L_d^{\text{eid}}[p_i]$ )
10:  if  $R_{\text{eid}} \neq 0 \wedge \text{active}[p_i] = 1 \wedge \text{vote}[p_i] = 0 \wedge \text{MT.Verify}(L_d^{\text{eid}}[p_i], p_i, \pi_{p_i}, R_{\text{eid}}) = 1$  then
11:     $\text{vote}[p_i] = 1$ 
12:     $E_v^{\text{eid}} \leftarrow \text{Enc.Add}(\text{pk}_{\text{Enc}}^{\text{TA}}, E_v^{\text{eid}}, L_d^{\text{eid}}[p_i])$  // homomorphic addition of the vote to the tally
13:    log  $\text{vote}[p_i], E_v^{\text{eid}}$ 
14:  else
15:    abort
16:  end if
17: end function

```

3.6 Tally

In the Tally subprotocol, Alg. 10, TA first decrypts the tallies for each option. Subsequently, it computes the vote percentages for each option. The final results only reflect the proportional distribution of votes, without disclosing the absolute voting power behind each option. This is important in scenarios where a number of participating delegates is small. If only a small number of delegates vote, the final result might reveal their total voting powers in the public vote setting. To address this, we can change TA and make it reveal the winning option only.

Algorithm 9 Private Voting

```

1: function VOTING(eid,  $L_d^{\text{eid}}, p_i, v$ ) // called by  $p_i$ 
2:   if  $v \in \{\text{yes}, \text{no}, \text{abstain}\}$  then
3:      $r \xleftarrow{\$} \mathbb{Z}_q^3$ 
4:      $E_v \leftarrow \text{Enc.Rerand}(\text{pk}_{\text{Enc}}^{\text{TA}}, L_d^{\text{eid}}[p_i], r_v)$ 
5:      $\forall j \neq v : E_j \leftarrow \text{Enc.E}(\text{pk}_{\text{Enc}}^{\text{TA}}, 0; r_j)$ 
6:      $\pi_{p_i} \leftarrow \text{MT.GetProof}(L_d^{\text{eid}}, p_i)$ 
7:      $\pi \leftarrow \text{zkProve}_{R_{\text{vote}}}((\text{pk}_{\text{Enc}}^{\text{TA}}, L_d^{\text{eid}}[p_i], E, p_i, R_{\text{eid}}, \pi_{p_i}), (v, r))$ 
8:      $\text{vote}[p_i], E^{\text{eid}} \leftarrow \text{ON CHAIN VOTING}(\text{eid}, p_i, E, \pi, \pi_{p_i}, L_d^{\text{eid}}[p_i])$ 
9:   else
10:    abort
11:   end if
12: end function
13: function ON CHAIN VOTING(eid)
14:   if  $R_{\text{eid}} \neq 0 \wedge \text{active}[p_i] = 1 \wedge \text{vote}[p_i] = 0 \wedge$ 
 $\text{zkVerify}_{R_{\text{vote}}}((\text{pk}_{\text{Enc}}^{\text{TA}}, L_d^{\text{eid}}[p_i], E, p_i, R_{\text{eid}}, \pi_{p_i}), \pi) = 1$  then
15:      $\text{vote}[p_i] = 1$ 
16:     for  $j \in \{\text{yes}, \text{no}, \text{abstain}\}$  do
17:        $E_j^{\text{eid}} \leftarrow \text{Enc.Add}(\text{pk}_{\text{Enc}}^{\text{TA}}, E_j^{\text{eid}}, E_v)$ 
18:     end for
19:     log  $\text{vote}[p_i], E^{\text{eid}}$ 
20:   else
21:     abort
22:   end if
23: end function

```

Algorithm 10 Tally

```

1: function AUTHORITY TALLY( $\text{sk}_{\text{Enc}}^{\text{TA}}, \text{eid}, E^{\text{eid}}$ ) // called by TA
2:    $D^{\text{eid}} \leftarrow \text{Enc.D}(\text{sk}_{\text{Enc}}^{\text{TA}}, E^{\text{eid}})$ 
3:   for  $i \in \{\text{yes}, \text{no}, \text{abstain}\}$  do
4:      $\text{res}_i^{\text{eid}} \leftarrow 100 \frac{D_i^{\text{eid}}}{\sum_{i=1}^3 D_i^{\text{eid}}}$ 
5:   end for
6:    $\text{res}_{\text{eid}} \leftarrow \text{ON CHAIN TALLY}(\text{eid}, \text{res}^{\text{eid}})$ 
7: end function
8: function ON CHAIN TALLY(eid,  $\text{res}^{\text{eid}}$ )
9:   log  $\text{res}^{\text{eid}}$ 
10: end function

```

4 SECURITY

4.1 The Universal Composability Framework

For our security analysis, we use a variant of the UC framework called SUC [12]. These frameworks require that no PPT adversary can distinguish between the execution of a real protocol π and an execution of an ideal process f .

In every execution, there is an environment \mathcal{Z} , an adversary \mathcal{A} , a set of parties p_1, \dots, p_n , and an ideal process f (sometimes called an ideal functionality). The environment \mathcal{Z} writes inputs to parties p_1, \dots, p_n , reads their outputs, and can also communicate with the adversary \mathcal{A} . The execution ends once \mathcal{Z} outputs a bit. This \mathcal{Z} represents all the external protocols that may run concurrently with our protocol. Due to space constraints, we will not describe all aspects of the communication and execution model of SUC, but we will briefly introduce the following special cases of these models.

In the *real model*, there is no ideal functionality, and honest parties adhere to the specified protocol π . In the *ideal model*, the parties are restricted to only communicate with the ideal functionality f . In the *hybrid model*, both the protocol π and ideal functionality f exist. Honest parties follow the protocol, but in addition, the protocol permits parties to send messages to f and specifies how to process messages received from f .

We say that π securely realizes f if, for every “real-world” adversary \mathcal{A} interacting with π , there exists an “ideal-world” adversary \mathcal{S} , such that no environment \mathcal{Z} can distinguish between these two scenarios. A similar statement can be defined for the hybrid model vs. ideal model.

The primary distinction of the SUC framework compared to UC is that SUC incorporates built-in authenticated channels. Additionally, it does not allow the dynamic addition of parties, thus mandating that protocols operate with sets of parties fixed ahead of time. Because of these constraints, the SUC framework cannot accommodate every type of protocol. However, it is compatible with our settings, making it a good choice for our security proof. Furthermore, [12] demonstrates a security-preserving transformation from SUC to UC. This essentially implies that our protocol, proven to be SUC-secure, can also be made UC-secure.

4.2 Security Proof

We focus on the public voting scenario, and note that our proof can be similarly adapted for the private voting scenario. In our analysis, we consider a static adversary that can corrupt voters but cannot corrupt the trusted authority TA. We operate within the local random oracle model, as we require the simulator to program the random oracle [10]. To prove the security of our protocol, we start by defining an ideal process for the voting process, ideal functionality $\mathcal{F}_{\text{vote}}$ (defined in Alg. 11). As previously mentioned, we assume the existence of a computing bulletin board, provided by a smart contract. This forces us to use a hybrid model. We define a contract functionality $\mathcal{F}_{\text{Contract}}$ (Alg. 12), that captures the computing bulletin board. Operating within the $\mathcal{F}_{\text{Contract}}$ -hybrid model, we abstract the bulletin board as a functionality that performs only the necessary computations for voting. We emphasize that $\mathcal{F}_{\text{Contract}}$ encompasses only the public on-chain computations in our protocol, which are reliably executed under the assumption of blockchain security or can be independently verified by any honest participant. Therefore, it is reasonable to model these parts as an ideal functionality.

Next, we build a simulator, as defined in Alg. 13. The simulator exists within the ideal model. There, the trusted authority supplies $\mathcal{F}_{\text{vote}}$ with the token distribution, L_T . The functionality then subsequently broadcasts L_T to all participants. Upon receiving the token distribution from $\mathcal{F}_{\text{vote}}$, \mathcal{S} executes the trusted authority setup as specified by the protocol, acquiring secret keys for encryption and signature schemes. This enables the simulator to sign and decrypt messages on behalf of the trusted authority for its interactions with \mathcal{A} (line 2 of Alg. 13). For every message it receives from the adversary on behalf of a corrupted party, the simulator forwards a corresponding message to the ideal functionality (lines 4-11 of Alg. 13). Possessing the secret key of the authority, it can decrypt the delegation vector ct sent by the adversary and relay the delegate’s address to the functionality in clear. Conversely, for each message received from the ideal functionality, the simulator, on behalf of an honest party, sends a corresponding message to the adversary (lines 12-20 of Alg. 13). In the delegation step, \mathcal{S} does not know the delegate’s address, only the fact that the voter has delegated. Therefore, to simulate ct , it encrypts an all-zero vector and

Algorithm 11 Functionality $\mathcal{F}_{\text{vote}}(\text{aux})$

- 1: Functionality $\mathcal{F}_{\text{vote}}(\text{aux}, T)$ runs with voters $p_1, \dots, p_n \in \mathcal{P}$, trusted authority TA and adversary \mathcal{A} . For every party $p_i \in \mathcal{P}$, $|\mathcal{P}| = n$, the functionality maintains a bit $\text{reg}_i \in \{0, 1\}$, integers $d_i \in [n]$ and $t_i \in [B]$. For initialization, set $\text{reg}_i = 0$, $d_i := i$ for all $i \in [n]$.
 - 2: – Upon receiving (setup, L_T) from TA, check whether $L_T \in [B]^N$ and set $t_i = L_T[i]$ for all $i \in [n]$. Send (setup, L_T) to all participants.
 - 3: – Upon receiving (register) from p_i , set $\text{reg}_i := 1$. Send (register, p_i) to all participants.
 - 4: – Upon receiving (unregister) from p_i , set $\text{reg}_i := 0$. Send (unregister, p_i) to all participants.
 - 5: – Upon receiving (delegate, p_j) from p_i , check whether $\text{reg}_i = 0$. In this case, set $d_i := j$. Send (delegate, p_i) to all participants.
 - 6: – Upon receiving (undelegate) from p_i , set $d_i := i$. Send (undelegate, p_i) to all participants.
 - 7: – Upon receiving (election setup, desc, eid) from p_i , set a bit $\text{vote}_j^{\text{eid}} := 0$ and an integer $t_j^{\text{eid}} = 0$ for every p_j and a triple of integers $r^{\text{eid}} \in [B]^3$. Store (eid, p_i) . Send (election setup, $\text{desc}, \text{eid}, p_i$) to all participants.
 - 8: – Upon receiving (election start, eid) from p_i , check whether there is a stored value (eid, p_i) . In this case, for all $i \in [n]$:
 - 9: if $d_i = j, i \neq j$, then set $t_j^{\text{eid}} := t_j^{\text{eid}} + t_i$
 - 10: else set $t_i^{\text{eid}} := t_i^{\text{eid}} + t_i$ end if
 - 11: Send (election start, eid, p_i) to all participants.
 - 12: – Upon receiving (vote, eid, v) from p_i , check whether $\text{reg}_i = 1$ and whether $v \in \{\text{yes}, \text{no}, \text{abstain}\}$. In this case, set $\text{vote}_i^{\text{eid}} := 1$ and $r^{\text{eid}}[v] := r^{\text{eid}}[v] + t_i^{\text{eid}}$. Send (vote, eid, v, p_i) to all participants.
 - 13: – Upon receiving (tally, eid) from TA, for every v set $r_v^{\text{eid}} = \frac{r_v^{\text{eid}}}{\sum_{v=1}^3 r_v^{\text{eid}}}$ and send (tally, $\text{eid}, r^{\text{eid}}$) to all participants.
-

simulates the proof of correctness. For tallying, it takes the result it received from $\mathcal{F}_{\text{vote}}$ and simulates a proof of correct decryption.

Finally, we can state our security theorem for our voting protocol Π defined in Sec. 3.

THEOREM 1. *The voting protocol Π in Sec. 3 SUC-securely realizes $\mathcal{F}_{\text{vote}}$ with respect to $\Pi^{\mathcal{F}_{\text{Contract}}}$, assuming a collision-resistant hash function, a CPA-secure encryption scheme, and a secure non-interactive zero-knowledge argument of knowledge.*

PROOF. For our analysis, we apply the hybrid argument technique. The goal is to start with our protocol and introduce modifications, step by step, gradually transforming into the ideal functionality. Note that SUC assumes the authenticated channels, therefore, we do not need to verify trusted authority’s signature explicitly, as it is implicitly verified in the model.

Experiment 0. Experiment 0 is the same as the protocol, with the exception that we restrict the adversary \mathcal{A} to generate Merkle proofs honestly.

SUC – HYBRID $\mathcal{F}_{\text{Contract}}^{\Pi, \mathcal{A}, \mathcal{Z}} \approx \text{EXP}_0$. We argue that the protocol and Experiment 0 are indistinguishable to \mathcal{Z} . To show that, assume the

Algorithm 12 On Chain Contract Functionality $\mathcal{F}_{\text{Contract}}$

- 1: Functionality $\mathcal{F}_{\text{Contract}}$ runs with voters $p_1, \dots, p_n \in \mathcal{P}$ and trusted authority TA.
 - 2: – Upon receiving (setup, $\text{pk}_{\text{Enc}}^A, \text{pk}_{\text{Sig}}^A, L_T, \sigma_T$) from TA, execute ON CHAIN SETUP($L_v, \text{pk}_{\text{Enc}}^A, \text{pk}_{\text{Sig}}^A, L_T, \sigma_T$). Send output to all participants.
 - 3: – Upon receiving (register) from p_i , execute ON CHAIN DELEGATE REGISTRATION(p_i). Send output to all participants.
 - 4: – Upon receiving (unregister) from p_i , execute ON CHAIN DELEGATE UNREGISTRATION(p_i). Send output to all participants.
 - 5: – Upon receiving (delegate, $\text{ct}, \pi, \pi_{p_i}$) from p_i , execute ON CHAIN DELEGATION($p_i, \text{ct}, \pi, \pi_{p_i}$). Send output to all participants.
 - 6: – Upon receiving (undelegate, ct), from p_i , execute ON CHAIN UNDELEGATION(p_i, ct). Send output to all participants.
 - 7: – Upon receiving (election setup, eid, desc) from p_i , execute ON CHAIN ELECTION SETUP($p_i, \text{eid}, \text{desc}$). Send output to all participants.
 - 8: – Upon receiving (election start, eid) from p_i , execute ON CHAIN ELECTION START(p_i, eid). Send output to all participants.
 - 9: – Upon receiving (vote, $\text{eid}, p_i, v, \pi_{p_i}, L_d^{\text{eid}}[p_i]$) from p_i , execute ON CHAIN VOTING($\text{eid}, p_i, v, \pi_{p_i}, L_d^{\text{eid}}[p_i]$). Send output to all participants.
 - 10: – Upon receiving (tally, $\text{eid}, \text{res}^{\text{eid}}$) from TA, execute ON CHAIN TALLY($\text{eid}, \text{res}^{\text{eid}}$). Send output to all participants.
-

opposite, so the protocol and Experiment 0 are not indistinguishable to \mathcal{Z} . The only difference is the ability of \mathcal{A} to generate valid Merkle proofs for invalid leaf values in SUC – HYBRID $_{\Pi, \mathcal{A}, \mathcal{Z}}^{\mathcal{F}_{\text{Contract}}}$. Therefore, we can build a new adversary \mathcal{B} , that uses \mathcal{Z} to find p_i, p'_i such that $p_i \neq p'_i$ but $H(p_i) = H(p'_i)$. However, due to the collision-resistance property of the underlying hash function, this can happen with a negligible probability only.

Experiment 1. Experiment 1 is the same as Experiment 0, except that we restrict the adversary \mathcal{A} further and require it to send the correct ct in the undelegation step.

$\text{EXP}_0 \approx \text{EXP}_1$. We argue that the Experiment 0 and Experiment 1 are indistinguishable to \mathcal{Z} . To show that, assume the opposite, so the Experiment 0 and Experiment 1 are not indistinguishable to \mathcal{Z} . The only difference is the ability of \mathcal{A} to find a ct such that $\text{ct} \neq \text{ct}_i$ but $H(\text{ct}_i) = H(\text{ct})$. Again, we can build a new adversary \mathcal{B} , that uses \mathcal{Z} to find ct, ct_i such that $\text{ct} \neq \text{ct}_i$ but $H(\text{ct}_i) = H(\text{ct})$. Due to the collision-resistance property of the hash function, this can happen with a negligible probability only.

Experiment 2. Experiment 2 is the same as Experiment 1, except the following change in DELEGATION – the proof π is generated by \mathcal{S} .

$\text{EXP}_2 \approx \text{EXP}_1$. Experiment 2 and Experiment 1 are indistinguishable to \mathcal{Z} . To show that, assume the opposite, so the Experiment 2 and Experiment 1 are not indistinguishable to \mathcal{Z} . Therefore, we can build an adversary \mathcal{B} , that uses \mathcal{Z} to distinguish between π simulated by \mathcal{S} and π generated by p_i . Due to the zero-knowledge property of the proof system, this can happen with a negligible probability only.

Algorithm 13 Simulator \mathcal{S}

- 1: \mathcal{S} controls the random oracle \mathcal{O} , so may assign responses to queries and, therefore, can simulate proofs. For all $i \in |\mathcal{P}| = n$, where \mathcal{P} is a set of all voters, \mathcal{S} assigns $\text{ct}_i := 0$. Let \mathcal{A} corrupt a subset $I \in \mathcal{P}$.
 - 2: – Upon receiving (setup, L_T) from $\mathcal{F}_{\text{vote}}$, \mathcal{S} sets $t_i = L_T[i]$ for all $i \in [n]$ and runs AUTHORITY SETUP to obtain $\text{pk}_{\text{Enc}}^{\text{TA}}, \text{pk}_{\text{Sig}}^{\text{TA}}, L_T, \sigma_T$ and sends the output to all participants. Additionally, \mathcal{S} knows the secret keys $\text{sk}_{\text{Enc}}^{\text{TA}}$ and $\text{sk}_{\text{Sig}}^{\text{TA}}$. \mathcal{S} runs ON CHAIN SETUP($\text{pk}_{\text{Enc}}^{\text{TA}}, \text{pk}_{\text{Sig}}^{\text{TA}}, L_T, \sigma_T$) and sends the output to all participants.
 - 3: – \mathcal{S} invokes \mathcal{A} and simulates $\mathcal{F}_{\text{Contract}}$ on valid calls, ignores all invalid calls.
 - 4: – **For every** $p_i \in I$:
 - 5: – Upon receiving (register) from p_i , \mathcal{S} sends (register) on behalf of p_i to $\mathcal{F}_{\text{vote}}$
 - 6: – Upon receiving (unregister) from p_i , \mathcal{S} sends (unregister) on behalf of p_i to $\mathcal{F}_{\text{vote}}$
 - 7: – Upon receiving (delegate, $\text{ct}, \pi, \pi_{p_i}$) from p_i , \mathcal{S} checks whether the proofs π, π_{p_i} are valid. If π_{p_i} is a valid proof but the leaf opening is not t_i , \mathcal{S} aborts. Otherwise, it decrypts ct . \mathcal{S} sets $\text{ct}_i = \text{ct}$ and sends (delegate, p_j) on behalf of p_i to $\mathcal{F}_{\text{vote}}$, where p_j is such that $\text{ct}[p_j] \neq 0$.
 - 8: – Upon receiving (undelegate, ct) from p_i . If $\text{ct}_i \neq \text{ct}$ but $H(\text{ct}_i) = H(\text{ct})$, then \mathcal{S} aborts. If $\text{ct}_i = \text{ct}$, it sends (undelegate) on behalf of p_i to $\mathcal{F}_{\text{vote}}$.
 - 9: – Upon receiving (election setup, eid, desc) from p_i , \mathcal{S} sends (election setup, desc, eid) on behalf of p_i to $\mathcal{F}_{\text{vote}}$
 - 10: – Upon receiving (election start, eid) from p_i , \mathcal{S} sends (election start, eid) on behalf of p_i to $\mathcal{F}_{\text{vote}}$
 - 11: – Upon receiving (vote, $\text{eid}, p_i, v, \pi_{p_i}, L_d^{\text{eid}}[p_i]$) from p_i , \mathcal{S} checks whether the proofs π and π_{p_i} are valid. If π_{p_i} is a valid proof but the leaf opening is not $L_d^{\text{eid}}[p_i]$, \mathcal{S} aborts. Otherwise, \mathcal{S} sends (vote, eid, v) on behalf of p_i to $\mathcal{F}_{\text{vote}}$.
 - 12: – **For every** $p_i \in \mathcal{P} \setminus I$:
 - 13: – Upon receiving (register, p_i) from $\mathcal{F}_{\text{vote}}$, \mathcal{S} executes ON CHAIN DELEGATE REGISTRATION(p_i) and sends the output to \mathcal{A} .
 - 14: – Upon receiving (unregister, p_i) from $\mathcal{F}_{\text{vote}}$, \mathcal{S} executes ON CHAIN DELEGATE UNREGISTRATION(p_i) and sends the output to \mathcal{A} .
 - 15: – Upon receiving (delegate, p_i) from $\mathcal{F}_{\text{vote}}$, \mathcal{S} produces ct as an encryption of all-zero vector, generates a Merkle proof π_{p_i} and simulates a zk-proof π . Then, \mathcal{S} executes ON CHAIN DELEGATION($p_i, \text{ct}, \pi, \pi_{p_i}$) and sends the output to \mathcal{A} .
 - 16: – Upon receiving (undelegate, p_i) from $\mathcal{F}_{\text{vote}}$, \mathcal{S} executes ON CHAIN UNDELEGATION(p_i, ct) and sends the output to \mathcal{A} .
 - 17: – Upon receiving (election setup, $\text{desc}, \text{eid}, p_i$) from $\mathcal{F}_{\text{vote}}$, \mathcal{S} executes ON CHAIN ELECTION SETUP($p_i, \text{eid}, \text{desc}$) and sends the output to \mathcal{A} .
 - 18: – Upon receiving (election start, eid, p_i) from $\mathcal{F}_{\text{vote}}$, \mathcal{S} executes ON CHAIN ELECTION START(p_i, eid) and sends the output to \mathcal{A} .
 - 19: – Upon receiving (vote, $\text{eid}, p_i, v, \pi_{p_i}$) from $\mathcal{F}_{\text{vote}}$, \mathcal{S} generates a Merkle proof π_{p_i} , executes ON CHAIN VOTING($\text{eid}, p_i, v, \pi_{p_i}, L_d^{\text{eid}}[p_i]$) and sends the output to \mathcal{A} .
 - 20: – Upon receiving (tally, $\text{eid}, \text{res}^{\text{eid}}$) from $\mathcal{F}_{\text{vote}}$, \mathcal{S} simulates a zk-proof π . Then, \mathcal{S} executes ON CHAIN TALLY($\text{eid}, \text{res}^{\text{eid}}, \pi$) and sends the output to \mathcal{A} .
-

Experiment 3. Is the same as Experiment 2, except the following change in DELEGATION – instead of generating ct as described in Alg. 4, set $\text{ct} := \text{Enc.E}(\text{pk}_{\text{Enc}}^{\text{TA}}, 0; \mathbf{r})$, where \mathbf{r} is drawn randomly.

$\text{EXP}_3 \approx \text{EXP}_2$. We argue that the Experiment 3 and Experiment 2 are indistinguishable to \mathcal{Z} . To show that, assume the opposite, so

the Experiment 3 and Experiment 2 are not indistinguishable to \mathcal{Z} . Therefore, we can build an adversary \mathcal{B} , that uses \mathcal{Z} to distinguish between $\text{Enc.E}(\text{pk}_{\text{Enc}}^{\text{TA}}, 0; r)$ and $\text{Enc.E}(\text{pk}_{\text{Enc}}^{\text{TA}}, t; r')$. Due to the CPA security of the encryption scheme, this can happen with a negligible probability only.

$\text{EXP}_3 \approx \text{SUC} - \text{IDEAL}_{\mathcal{F}_{\text{vote}}, \mathcal{S}, \mathcal{Z}}$. The Experiment 3 is the simulated interaction of \mathcal{Z} and \mathcal{S} . Note that all zero-knowledge proofs in the Experiment 3 are simulated, and the delegated encryption is generated as if encrypting an all-zero vector, exactly like \mathcal{S} does. Importantly, \mathcal{S} never aborts, as we restricted \mathcal{A} in the Experiments 0 and 1. \square

5 IMPLEMENTATION

5.1 System Implementation Details

We developed a proof-of-concept implementation of Kite ² which supports private delegation and public voting. Our implementation totalled 10,238 lines of code and includes a React frontend (2,450 lines), two servers written in Rust (3,312 lines), zero-knowledge circuits written in Noir (911 lines), and on-chain Ethereum smart contracts developed in Solidity (4,476 lines). The frontend was designed to mimic the Nouns DAO tally UI as a concrete use case. The two Rust servers manage all cryptographic operations and communications with the chain. The first Rust server functions as the backend tasked with constructing the zero knowledge proofs as well as deploying and calling most on-chain contracts with the relevant data required for verification. The second Rust server represents the trusted authority TA, responsible for setting up the on-chain private governance contract and decrypting the final tallies. In practice, the trusted authority would be securely distributed among multiple parties using threshold decryption. However, for our proof-of-concept, the trusted authority is implemented as a single entity.

Our on-chain Solidity implementation builds on the Open Zepelin ERC20 and Compound Governor Bravo contracts. We extended the ERC20 contract to include a locking mechanism, to restrict the movement of tokens when needed. We utilize a mapping from addresses to booleans, indicating the lock status for each user. The standard ERC20 functions – transfer, transferFrom, approve, and spendAllowance – have been modified to revert if the user’s tokens are locked, while retaining their original functionality otherwise.

We use COMP tokens as the ERC20 governance tokens. To vote on proposals, every voter must delegate their voting power to themselves or to some other voter (the delegate). When a proposal is first posted, the Comp contract (which implements the COMP token) takes a snapshot of the current voting powers and delegation status of all voters. When a voter casts their vote, the Comp contract sends the snapshot of the user’s voting power, as recorded when the proposal was first posted, to the Governor Bravo contract.

The voting protocol logic is implemented as a significant extension to the Governor Bravo contract. This includes the logic to verify all the ZK proofs generated by the Rust servers on behalf of the participants, as well as the logic to act homomorphically on encrypted data. The original Governor Bravo contract permitted

users to delegate their voting power while simultaneously receiving delegations from others. Our delegate registration implementation eliminates this potential loophole. Additionally, when a user who previously delegated their voting power wishes to redelegate to a new account, there was no explicit undelegation step; instead, users directly delegated to another individual. Our implementation rectifies this by incorporating an undelegation step, enabling users to subtract their token balance from their original delegate.

To reduce the time to generate the ZK proofs, our implementation introduces the notion of an *anonymity set*. When a voter intends to delegate, the backend server constructs an anonymity set by randomly sampling $T - 1$ delegate accounts without replacement, and adding the delegate’s account to obtain an anonymity set of size T . The voter’s posted vector of encrypted powers is now of size T , which is smaller than the total number of delegates in the system. This both reduces the amount of data to post on chain, and reduces the time to generate the relevant ZK proofs. The cost is that an observer learns that the voter delegated to someone in the anonymity set, whereas in the full protocol of Section 3, an observer only learns that a delegation to some delegate took place.

On election start, the off-chain, trusted authority TA generates the election snapshot and pushes it onto the blockchain. The snapshot is stored on-chain and written to the transaction log to facilitate off-chain access to the snapshotted voting powers.

In the tally decryption step, the trusted authority TA decrypts the tally for a specific election and computes the percentages of votes for, against, and abstained. These percentages, along with a zkproof of correct decryption are submitted on-chain. Posting percentages instead of raw or rounded vote weights minimizes the disclosure of information regarding a delegate’s voting power, while still ensuring transparency in the tally decryption process.

5.2 Implementing the ZK Relations

We implemented each of the ZK relations used in Section 3, including $\mathcal{R}_{\text{vote}}$ (used for private voting), though we only support public voting in our proof-of-concept. While in Section 3 we assumed a fully trusted authority TA to simplify the proof of security, our implementation relaxes this assumption somewhat. In particular, we require the authority to provide a ZK proof of correct decryption of the final tally results. We do so using the following \mathcal{R}_{dec} relation.

$$\mathcal{R}_{\text{dec}} := \left\{ \left(\text{sk}_{\text{Enc}}^{\text{TA}}, (\text{pk}_{\text{Enc}}^{\text{TA}}, E^{\text{eid}}, \text{res}^{\text{eid}}) \right) \mid \text{for } i = 1, 2, 3: \right. \\ \left. D_i^{\text{eid}} \leftarrow \text{Enc.D}(\text{sk}_{\text{Enc}}^{\text{TA}}, E_i^{\text{eid}}) \wedge \text{res}_i^{\text{eid}} = 100 \frac{D_i^{\text{eid}}}{\sum_{i=1}^3 D_i^{\text{eid}}} \right\} \quad (3)$$

where $\text{sk}_{\text{Enc}}^{\text{TA}} \in \mathbb{Z}_q$ and $\text{pk}_{\text{Enc}}^{\text{TA}} \in \mathbb{G}$ are the secret and public keys of TA, $E^{\text{eid}} \in \mathbb{G}^3$ is a vector of the encrypted numbers of votes for each option, and $\text{res}^{\text{eid}} \in [0, 100]$ is a vector of the percentage of votes for each option.

We implemented all zero-knowledge relations using the Noir language [27]. Noir is a domain-specific language with support for a modular backend meant to work with any ACIR (Abstract Circuit Intermediate Representation)-compatible proving system. We use Aztec Labs’ Barretenberg backend for our proving system, which runs on PLONK [17]. Noir is a powerful tool for rapid code

²<https://github.com/PilliCode/GovernorPrivate>

iteration in circuits that deal with complex operations, while still providing reasonable performance. This makes Noir well-suited for our needs. One limitation of Noir is its lack of support for elliptic curve operations on many familiar curves such as secp256k1. Thus our implementation is centered around the use of the BabyJubJub curve [29]. The restriction to arithmetic on this curve necessitates the use of a few extra proofs in the implementation in order to manage gas costs, since these curve operations are not currently implemented in Solidity efficiently and robustly. Specifically, we make use of the following relations to offload on-chain work to our backend rust server:

$$\begin{aligned} \mathcal{R}_{\text{addmt}} &:= \{(null), (pk_{\text{Enc}}^{\text{TA}}, ct_1, ct_2, ct_+, \pi_{p_i}, R) \mid \\ \text{MT.Verify}(ct_2, p_i, \pi_{p_i}, R) = 1 \wedge ct_+ &= \text{Enc.Add}(pk_{\text{Enc}}^{\text{TA}}, ct_1, ct_2)\} \end{aligned} \quad (4)$$

$$\begin{aligned} \mathcal{R}_{\text{vebsub}} &:= \{(null), (pk_{\text{Enc}}^{\text{TA}}, ct_1, ct_2, ct_-) \mid \\ ct_- &= \text{Enc.Add}(pk_{\text{Enc}}^{\text{TA}}, ct_1, -ct_2)\} \end{aligned} \quad (5)$$

$$\begin{aligned} \mathcal{R}_{\text{vecadd}} &:= \{(null), (pk_{\text{Enc}}^{\text{TA}}, ct_1, ct_2, ct_+) \mid \\ ct_+ &= \text{Enc.Add}(pk_{\text{Enc}}^{\text{TA}}, ct_1, ct_2)\} \end{aligned} \quad (6)$$

$$\begin{aligned} \mathcal{R}_{\text{encsub}} &:= \{(null), (pk_{\text{Enc}}^{\text{TA}}, t, ct, ct_-) \mid \\ e &= \text{Enc.E}(pk_{\text{Enc}}^{\text{TA}}, -t; 0) \wedge ct_- = \text{Enc.Add}(pk_{\text{Enc}}^{\text{TA}}, ct, e)\} \end{aligned} \quad (7)$$

$$\mathcal{R}_{\text{enc}} = \{(null), (pk_{\text{Enc}}^{\text{TA}}, ct, t, r) \mid ct = \text{Enc.E}(pk_{\text{Enc}}^{\text{TA}}, t; r)\} \quad (8)$$

In particular, $\mathcal{R}_{\text{addmt}}$ is used by Alg. 8, $\mathcal{R}_{\text{vebsub}}$ by Alg. 5, $\mathcal{R}_{\text{encsub}}$ by Alg. 3, and \mathcal{R}_{enc} by Alg. 2. As a special case, Alg. 4 requires the homomorphic addition of ciphertext vectors (for which we would verify a proof of $\mathcal{R}_{\text{vecadd}}$) immediately after verifying the relation \mathcal{R}_{del} . In our implementation, we actually concatenate \mathcal{R}_{del} and $\mathcal{R}_{\text{vecadd}}$ into one circuit to avoid the consecutive execution of two expensive proof verifications on-chain. However, since \mathcal{R}_{del} is inherent to the protocol and $\mathcal{R}_{\text{vecadd}}$ is implementation-specific, we report their metrics separately in Section 6.1.

These proofs allow us to carry out expensive operations lacking robust implementations in Solidity within our Rust backend. The results of these operations, as well as a proof of their correctness, are then provided to the on-chain contract.

All of the relations we use must be verified on-chain. Noir’s tooling allows for the generation of smart contracts with functionality to load a relation-specific verification key and verify a set of public inputs against a proof, with each contract corresponding to a single relation. We extend these auto-generated contracts by rolling them into a single contract (3,291 lines of code, most of which are generated by Noir) that has the ability to load one of many verification keys and verify any of the relations we use in our implementation. This reduces the number of helper contracts referenced by our governance contract, which in turn reduces gas costs and eliminates duplicated code across contracts. As part of the delegation process, a user submits a public list of delegate addresses. An

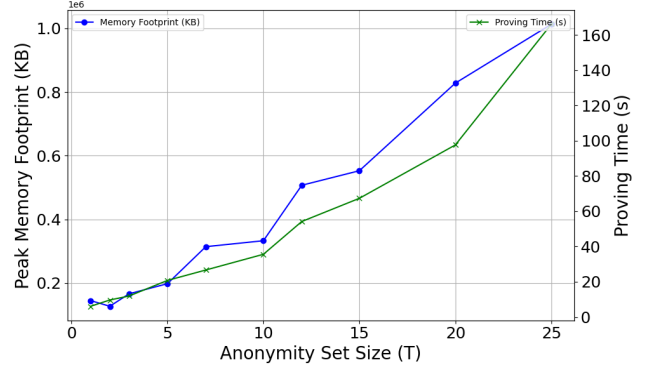


Figure 1: Proving time of \mathcal{R}_{del} and peak memory footprint of proof generation vs anonymity set size.

observer knows that the user delegated their voting power to one of these addresses, but not which one in particular. Thus, in Kite, the relation \mathcal{R}_{del} depends on the size of the anonymity set, defined in Section 5.1. For the sake of simplicity in our proof-of-concept implementation, we provide users the option to use an anonymity set of size 5, 10, or 20. However, adding support for further set sizes is trivial. Smaller anonymity sets significantly reduce proving time and gas costs but offer less privacy. Conversely, larger sets provide greater privacy at increased computational cost, leaving users to balance these trade-offs based on their privacy needs.

The time cost of the implementation is important to the usability of the protocol. Since the verification time is constant, we took steps to reduce the proving time of the relations. We use the zk-friendly Poseidon hash function [20] in all necessary Merkle proofs, and implement functions like ElGamal encryption within Noir circuits ourselves to take advantage of speedups afforded by details of our protocol. For example, many scalar-point multiplies can be foregone when the randomness is deterministic, as is the case in Alg. 2,3. We also implement a custom "small scalar"-point multiplication using the Double-and-Add algorithm. This removes unnecessary loop iterations, as the Noir interface for this operation only supports 254 bit scalars while we often only need 32 bit.

6 EVALUATION

6.1 Proving System Performance

All proving times are computed as an average of ten runs using the Nargo CLI provided by Aztec Labs. For \mathcal{R}_{del} , we report the metrics as a function of the size of the anonymity set, and collected results for sets of size up to 25. As mentioned previously, in the full implementation we only make use of the circuits for sizes 5, 10, and 20. We see that the proving time and associated memory cost for \mathcal{R}_{del} increases with the anonymity set, as expected. The largest anonymity set size we tested took just over 2 minutes and 46 seconds to prove. However, sets of intermediate sizes 15 and 12 were much faster, with proving times slightly over and below 1 minute, respectively (Fig. 1). We observed that the inclusion of a small-scalar multiplication implementation reduces the proving time of \mathcal{R}_{del} by an average of 42.61% across all anonymity

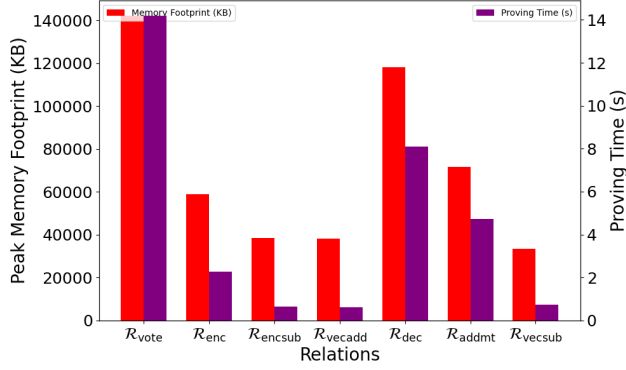


Figure 2: Proving times and peak memory footprints of proof generation of all relations, except R_{del} .

set sizes. Out of the other relations, R_{vote} is the most costly, with the additional implementation-specific and protocol-independent relations R_{addmt} , R_{vecsub} , R_{vecadd} , R_{encsub} , and R_{enc} claiming only marginal resources in comparison (Fig. 2). The proving times of the vector-valued R_{vecsub} and R_{vecadd} would depend on the size of the anonymity set, but in our implementation we use the circuits for a vector of length 20 and pad to fill the vector for smaller set sizes since these relations are quite lightweight.

We also examine the circuit size in number of gates, and find that the order of relations by proving time is the same as their circuit size. The largest circuit we implemented was R_{del} with an anonymity set size of 25, with a reported 566,597 gates (767,297 without small-scalar multiplication); the smallest being the implementation-specific 20-element R_{vecadd} with 1,325 gates.

The proofs posted on-chain are a constant 4,288 bytes in size. Gas costs to verify the different relations averaged to 406,646 with a median of 396,323 on Forge’s Anvil local testnet. Gas cost optimization was not a primary focus of this work, leaving room for reduction in future work. In our implementation, multiple relations are concatenated into a single larger relation, allowing one proof to verify multiple operations, such as R_{del} and R_{vecsub} , reducing gas costs. Since we also used additional proofs to off-load the elliptic curve operations to the Rust backend, implementing operations on the chosen curve efficiently in Solidity to avoid the use of additional proofs can result in future gas savings. However, the use of a single master verifier contract did significantly reduce the gas costs associated with the deployment of verifier contracts in our implementation by eliminating duplicated information on the blockchain. If the verifier contracts for each of the eight circuits we used were deployed separately as they are generated by Noir’s tooling, it would cost an estimated 18,567,088 gas (according to estimates from Forge gas reports [2]). This is compared with our master verifier’s deployment cost of 5,981,566 gas, saving 12,585,522 gas during contract deployment.

Our proof-of-concept achieves reasonable performance on a consumer-grade machine with minimal optimization. Most proofs are generated within 15 seconds, except delegation, which takes 7–167 seconds based on the desired privacy levels. However, delegation is rather infrequent, as its main purpose is to reduce user

interaction with the voting system, leaving end-users to primarily engage with lower-latency operations.

6.2 Proof-of-Concept Performance

The end-to-end implementation was deployed to a local Anvil testnet. We measured the gas cost of each operation (Fig. 3) required for end-to-end voting and delegation in our proof-of-concept implementation. The gas cost associated with verifying a proof on chain is independent of the relation being verified, the average associated gas cost is represented by the red horizontal line. As expected, delegation and undelegation are the most resource-intensive operations due to the need for verifying homomorphic vector operations. In turn, delegate unregistration has the lowest gas cost due to its minimal inputs and on-chain operations.

7 CONCLUSION AND FUTURE WORK

We presented Kite, a voting system for DAOs, that enables *private* delegation of voting power. The system is implemented as a direct extension to a popular DAO voting smart contract. Currently, Kite provides either complete transparency for delegate votes (so that delegates can be held accountable for their voting record) or total privacy for delegate votes, where only the delegate knows their own votes. One direction for future research is something in between, namely a system that maintains confidentiality of the delegate’s vote from the general public, but reveals how they voted to voters who have delegated their voting tokens to the delegate.

Acknowledgments. This work was funded by IOG, NSF, DARPA, the Simons Foundation, UBRI, and NTT Research. Opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA.

REFERENCES

- [1] Compound Governor Bravo. <https://docs.compound.finance/v2/governance/#governor-bravo>.
- [2] Foundry book: Gas reports. <https://book.getfoundry.sh/forge/gas-reports>.
- [3] Nouns dashboard. <https://dune.com/thepass/nouns>.
- [4] 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.
- [5] J. Benaloh, M. Naehrig, O. Pereira, and D. S. Wallach. ElectionGuard: a cryptographic toolkit to enable verifiable elections. In *USENIX Security 2024*. USENIX Association, Aug. 2024.
- [6] D. Bernhard and B. Warinschi. Cryptographic voting - A gentle introduction. In *FOSAD*, volume 8604 of *Lecture Notes in Computer Science*, pages 167–211. Springer, 2013.
- [7] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *ITCS*, pages 326–349. ACM, 2012.
- [8] C. Blum and C. I. Zuber. Liquid democracy: Potentials, problems, and perspectives. *Journal of political philosophy*, 24(2):162–182, 2016.
- [9] D. Boneh and V. Shoup. *A graduate course in applied cryptography (version 0.6)*. 2023. cryptobook.us.
- [10] J. Camenisch, M. Drijvers, T. Gagliardoni, A. Lehmann, and G. Neven. The wonderful world of global random oracles. In *EUROCRYPT (1)*, volume 10820 of *Lecture Notes in Computer Science*, pages 280–312. Springer, 2018.
- [11] R. Canetti. Universally composable security. *J. ACM*, 67(5):28:1–28:94, 2020.
- [12] R. Canetti, A. Cohen, and Y. Lindell. A simpler variant of universally composable security for standard multiparty computation. In R. Gennaro and M. J. B. Robshaw, editors, *CRYPTO 2015, Part II*, volume 9216 of *LNCS*, pages 3–22, Santa Barbara, CA, USA, Aug. 16–20, 2015.
- [13] R. Carback, D. Chaum, J. Clark, J. Conway, A. Essex, P. S. Herrnsen, T. Mayberry, S. Popoveniuc, R. L. Rivest, E. Shen, A. T. Sherman, and P. L. Vora. Scantegrity II municipal election at takoma park: The first E2E binding governmental election

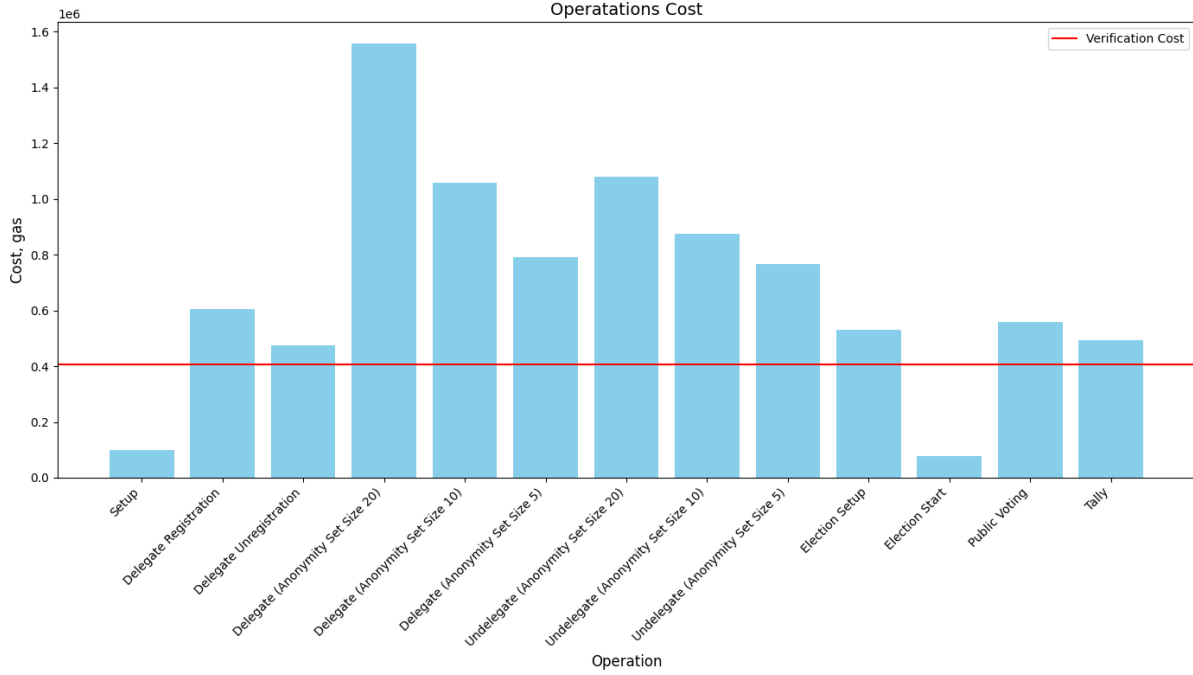


Figure 3: Gas cost of all protocol operations.

- with ballot privacy. In *USENIX Security 2010*, pages 291–306, Washington, DC, USA, Aug. 11–13, 2010. USENIX Association.
- [14] D. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Commun. of the ACM*, 24(2):84–90, 1981.
- [15] Open Zeppelin ERC20. <https://docs.openzeppelin.com/contracts/4.x/erc20>.
- [16] M. Fischlin. Communication-efficient non-interactive proofs of knowledge with online extractors. In V. Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*, pages 152–168, Santa Barbara, CA, USA, Aug. 14–18, 2005.
- [17] A. Gabizon, Z. J. Williamson, and O. Ciobotaru. PLONK: permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. *IACR Cryptol. ePrint Arch.*, page 953, 2019.
- [18] N. Glaeser, I. A. Seres, M. Zhu, and J. Bonneau. Cicada: A framework for private non-interactive on-chain auctions and voting. *IACR Cryptol. ePrint Arch.*, page 1473, 2023.
- [19] Open Zeppelin Governance. <https://docs.openzeppelin.com/contracts/4.x/api/governance>.
- [20] L. Grassi, D. Khovratovich, and M. Schofnegger. Poseidon2: A faster version of the poseidon hash function. In N. El Mrabet, L. De Feo, and S. Duquesne, editors, *AFRICACRYPT 23*, volume 14064 of *LNCS*, pages 177–203, Sousse, Tunisia, July 19–21, 2023.
- [21] A. Hall. Governance FAQs, 2023.
- [22] O. Kulyk, K. Marky, S. Neumann, and M. Volkamer. Introducing proxy voting to helios. In *ARES*, pages 98–106. IEEE Computer Society, 2016.
- [23] O. Kulyk, S. Neumann, K. Marky, J. Budurushi, and M. Volkamer. Coercion-resistant proxy voting. *Comput. Secur.*, 71:88–99, 2017.
- [24] O. Kulyk, S. Neumann, K. Marky, and M. Volkamer. Enabling vote delegation for boardroom voting. In *Financial Cryptography Workshops*, volume 10323 of *Lecture Notes in Computer Science*, pages 419–433. Springer, 2017.
- [25] O. Kulyk, S. Neumann, M. Volkamer, C. Feier, and T. Koster. Electronic voting with fully distributed trust and maximized flexibility regarding ballot design. In *EVOTE*, pages 1–10. IEEE, 2014.
- [26] T. Lloyd, D. O’Broin, and M. Harrigan. The on-chain and off-chain mechanisms of dao-to-dao voting. In *IEEE International Conference on Blockchain, Blockchain 2024, Copenhagen, Denmark, August 19–22, 2024*, pages 649–655. IEEE, 2024.
- [27] Noir. <https://noir-lang.org>.
- [28] Nouns DAO. <https://nouns.wtf>.
- [29] B. WhiteHat, J. Baylina, and M. Belles. Baby jubjub elliptic curve.
- [30] B. Zwartendorfer, C. Hillebold, and P. Teufl. Secure and privacy-preserving proxy voting system. In *ICEBE*, pages 472–477. IEEE Computer Society, 2013.

A ZK-SNARKS SECURITY DEFINITIONS

We define completeness, knowledge soundness, zero-knowledge, non-interactivity, and succinctness for a zk-SNARK below.

- **Completeness:** if $(x, w) \in \mathcal{R}$, then verification should pass. That is, for all $\lambda \in \mathcal{N}$ and all $(x, w) \in \mathcal{R}$:

$$\Pr \left[V(pp, x, \pi) = 1 \quad : \quad \begin{array}{l} pp \xleftarrow{\$} \text{setup}(1^\lambda) \\ \pi \leftarrow P(pp, x, w) \end{array} \right] = 1$$

- **Knowledge Soundness:** if an adversary can produce a valid proof for some x , then there should be a polytime extractor that can compute a witness w such that $(x, w) \in \mathcal{R}$. That is, Π has knowledge error ϵ if there exists a PPT extractor \mathcal{E} such that for all PPT $\mathcal{A}_0, \mathcal{A}_1$:

$$\Pr \left[\begin{array}{l} (x, w) \in \mathcal{R} \quad : \quad \begin{array}{l} pp \xleftarrow{\$} \text{setup}(1^\lambda) \\ (x, st) \xleftarrow{\$} \mathcal{A}_0(pp) \\ w \xleftarrow{\$} \mathcal{E}_{\mathcal{A}_1(pp, st)}(pp) \end{array} \end{array} \right] \geq$$

$$\Pr \left[\begin{array}{l} V(pp, x, \pi) = 1 \quad : \quad \begin{array}{l} pp \xleftarrow{\$} \text{setup}(1^\lambda) \\ (x, st) \xleftarrow{\$} \mathcal{A}_0(pp) \\ \pi \xleftarrow{\$} \mathcal{A}_1(pp, st) \end{array} \end{array} \right] - \epsilon$$

- **Zero-Knowledge:** We state the definition in the random oracle model where all the algorithms are oracle machine that can query an oracle $H : \mathcal{X} \rightarrow \mathcal{Y}$ for some finite sets \mathcal{X} and \mathcal{Y} . The zk-SNARK is zero knowledge if there is a PPT simulator $\Pi.S$ such that for all $(x, w) \in \mathcal{R}$ and all PPT

adversaries \mathcal{A} , the following function is negligible

$$\text{Adv}_{\mathcal{A}, \Pi}^{\text{zk}}(\lambda) := \left| \frac{\Pr[\mathcal{A}^H(\text{pp}, x, P^H(\text{pp}, x, w)) = 1] - \Pr[\mathcal{A}^{H[h]}(\text{pp}, x, \pi) = 1]}{2} \right|$$

where $\text{pp} \xleftarrow{\$} \text{setup}(1^\lambda)$ and $(\pi, h) \xleftarrow{\$} \Pi.S(\text{pp}, x)$. Here h is a partial function $h : \mathcal{X} \rightarrow \mathcal{Y}$ output by $\Pi.S$, and $H[h]$

refers to the oracle $H : \mathcal{X} \rightarrow \mathcal{Y}$ modified by entries in h . That is, we allow $\Pi.S$ to program the oracle H .

- **Non-interactive:** the proof is non-interactive, and a proof created by the prover can be checked by any verifier.
- **Succinct:** the proof size and verifier runtime are $o(|w|)$. The verifier can run in linear time in $|x|$.