Kite: How to Delegate Voting Power Privately

Kamilla Nazirkhanova, Vrushank Gunjur, X. Pilli Cruz-De Jesus, and Dan Boneh

Stanford University
{nazirk, vrushank, pilli, dabo}@stanford.edu

Abstract. Ensuring ballot secrecy is crucial to maintaining 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 protocol that enables *private* delegation of voting power for DAO members, meaning that 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 evaluate our implementation, showing that the most expensive operation, delegation, takes 7–167 seconds on a consumer-grade machine depending on the requested privacy level, demonstrating Kite's practical feasibility.

Keywords: DAO, 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 might be executed by the smart contract that manages the DAO. The proliferation of DAOs has generated renewed interest in new voting mechanisms for DAOs [26], and has underscored the importance of privacy in voting. For example, the Nouns DAO has been exploring ways to improve privacy in voting [4].

The research community has been exploring private digital voting systems for a long time, starting with the work of Chaum [17] in 1981. Some protocols are based on homomorphic encryption [7, Ch.3], some are based on mix nets [7, Ch.6], some are based on blind signatures [7, Ch.2], and some are based on other mechanisms. Modern protocols stress the notion of end-to-end verifiability, where the core requirement is that every voter can verify that their vote was counted as cast. [5,6,16]. We refer to [7] for a survey of the area. More recently, some papers study private voting in the context of blockchains [23,32].

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* [24] — 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, or delegate to themselves to vote directly. 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 fact, this is not specific to DAOs and is central to representative democracy, where voters privately elect representatives who vote on their behalf publicly. In a liquid democracy [9], Alice can revoke her delegation at any time and delegate to someone else as often and as many times as she wants. Liquid democracy allows delegations to form transitive chains, with proxy voting being a special case of chains of length one. Indeed, this logic is supported by most DAO governance contracts.

In practice, a delegate can tell when a voter has delegated away from the delegate. This makes voters reluctant to re-delegate because public re-delegation can cause social friction. Concretely, in the Nouns DAO 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 September 2025, an average of 70% of votes were cast by delegates. This influence has consistently grown over time, with the average proposal in August 2025 seeing 78% of votes coming from delegates. Moreover, delegates command a larger portion of the voting power, accounting for 46.8%, in contrast to regular voters at 20.2%, while the remainder of the voting power falls under the control of the treasury [3].

These numbers suggest that delegation is wide spread. To improve the voter experience, DAOs governance systems would benefit from *private* delegation, so that re-delegation does not incur a social cost to voters. Beyond private delegation one may also ask for private voting for delegates. However, we stress that private delegate voting is not always desirable because it makes it harder for voters to hold their delegates to account, as it was noted in [33].

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. Such design reduces social pressure or friction, so Alice is free to re-delegate her voting power without concern for how David or others might react. Privacy means that delegations are hidden by default, but the protocol does not prevent a voter from later proving how they delegated. Therefore, we do not claim secrecy, which would also require receipt-freeness. Importantly, each voter can only delegate their own voting power, they cannot delegate someone else's. This design ensures there are never delegation cycles, distinguishing Kite from liquid democracy systems. 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 and cannot be erased. In practice, a secure blockchain (e.g., Ethereum) provides such a bulletin board, and its transaction mechanism ensures authentication: every transaction is signed by the submitting account, so only the rightful owner of each account can submit delegation, undelegation, or voting messages.

Kite has three types of participants: voters, delegates, and a tally committee, which is technically a set of parties but, for clarity, we refer to interchangeably as the "trusted authority." These participants interact with the voting system using the following functions: (the detailed implementation of these functions is described in Section 3, and details on realizing the trusted authority as a set of parties are provided in Remark 1).

Setup: initiated by the tally committee. The tally committee 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 [13]. In particular, we use a variant called (SUC) [14]. We first define an ideal voting functionality and 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 proof-of-concept in Solidity, extending the standard Governor Bravo smart contract [1] to support private delegation with public voting. Our user interface mimics Nouns DAO to demonstrate how our protocol addresses their quid pro quo voting challenges. This approach makes our implementation broadly 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 [20], a widely-used standard for fungible tokens on the Ethereum network. Crucially, Kite requires a mechanism to lock delegated tokens. This operation temporarily restricts their transfer and use in order to prevent the reuse of the voting power. Without such a lock, a user could delegate their tokens and then transfer them, allowing the voting power to be used twice.

Our implementation uses zero-knowledge SNARKs [8] 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 framework 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 [40,28,29,30], addressing properties such as vote privacy, delegation privacy, coercion-resistance, and robustness. Kulyk et al. [28,29,30] explored integrating proxy voting into systems like Helios [5] and boardroom voting [31], achieving delegation privacy using various tools such as anonymous channels, delegation servers, or secret-sharing. In contrast, Kite does not require additional parties or anonymous channels, nor does the delegator need to communicate with every voter, and the delegate learns nothing, not even that they were delegated to. These prior approaches are less suitable for blockchain settings, where minimizing additional infrastructure and on-chain interactions is crucial.

In another line of work [37,38,39,27], delegation privacy, alongside other security guarantees, was explored in the context of blockchain governance, with the protocols rigorously proven UC-secure. These systems support delegation by allowing a voter to indicate, when casting a ballot, that their vote is delegated to a delegate, with privacy enforced via encryption and/or mix-nets. In contrast, Kite separates voting and delegation, so that voters do not need to delegate in every election. It also allows delegates to cast votes either publicly, enabling accountability, or privately, preserving privacy, whereas prior schemes support only private voting. Moreover, Kite achieves a much more straightforward tallying process, requiring only a single decryption of the final tally. By comparison, the schemes of [37,38,39,27] must process every ballot individually to compute delegations and the final result. Since these computations in practice are distributed among the tally committee, simpler operations are desirable, especially in a blockchain setting. Formally, while Kite also follows the simulation paradigm, our analysis is conducted in the SUC framework rather than full UC. We also emphasize that our design naturally extends existing DAO governance protocols and can be easily adapted without any core changes.

2 Preliminaries

In this section we briefly list the cryptographic primitives used in our system, all of which are standard.

- A CPA-secure additively homomorphic encryption scheme ($\mathsf{Enc.Gen}, \mathsf{Enc.E}, \mathsf{Enc.D}, \mathsf{Enc.Add}, \mathsf{Enc.Rerand}$)
- An existentially unforgeable digital signature scheme ($\mathsf{Sig.Gen}, \mathsf{Sig.Sign}, \mathsf{Sig.Verify}$)
- A collision resistant hash function H
- A Merkle Tree ($\mathsf{MT.GetRoot}, \mathsf{MT.GetProof}, \mathsf{MT.Verify}$) from H

The complete syntax and details are provided in Appendix A.

Additionally, we need succinct non-interactive zero-knowledge arguments. A **Succinct Non-interactive Zero-Knowledge Argument of Knowledge** for a relation $\mathcal{R}$ consists of three algorithms ($\mathcal{G}, \mathsf{zkProve}, \mathsf{zkVerify}$) such that:

- $\mathcal{G}(1^\lambda) \to \mathsf{pk}, \mathsf{vk}$: generates a prover and verification key pair.
- $\mathsf{zkProve}(\mathsf{pk}, R, w) \to \pi$: outputs a proof for a statement R given a witness w.
- $\mathsf{zkVerify}(\mathsf{vk}, R, \pi) \to b \in \{0, 1\}$: verifies the validity of proof π.

We require it to be complete, succinct, and zero-knowledge (as defined in Appendix A.4). Importantly, we need it to be knowledge-sound, however, we require a straight-line extractor [21], as we further prove the security of our protocol in the universal composability framework.

Throughout our voting protocol, we make use of zero-knowledge proofs for various relations. In the subsequent sections, we omit the prover and verifier key inputs in the corresponding algorithms for brevity, and specify the relation as a subscript.

3 Protocol

In Kite, we identify three types of entities: a set of voters $\{p_i\}_{i \in n}$, 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 practical relaxations to the trust in the TA later in this Section and 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 $(\mathsf{Enc}(0), \mathsf{Enc}(t_2), \mathsf{Enc}(t_3))$, 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 $(\mathsf{Enc}(0), \mathsf{Enc}(0), \mathsf{Enc}(t_1))$ to the contract. The contract then updates the delegates' voting power list to $(\mathsf{Enc}(0), \mathsf{Enc}(t_2), \mathsf{Enc}(t_3 + t_1))$ 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 $\mathsf{Enc}(t_3 + t_1)$, 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 $\mathsf{Enc}(t_3 + t_1)$ 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 $(\mathsf{Rerand}(\mathsf{Enc}(t_3 + t_1)), \mathsf{Enc}(0))$.

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

Note that every voter's encryption must include a proof of correctness. Otherwise, a malicious voter could submit an ill-formed ciphertext and affect the election outcome. Kite uses two such proofs: one for correct delegation and one for correct voting, only needed in the private voting setting.

Remark 1 (Distributing the trusted authority). For clarity, we model the trusted authority as a single party. In practice, however, it should be implemented as a set of parties, some of which may be corrupt. The tallying procedure of Kite is simple, requiring only the decryption of a publicly available encrypted tally, and can be easily realized using an appropriate threshold decryption scheme [19,15,10]. In this setting, tally committee post their decryption shares on the chain, and once a sufficient number of shares is available, the final result can be decrypted.

Remark 2 (Ranked-choice voting). 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. While tallying would require decrypting individual ballots, in practice this may be unnecessary if the winner can be determined from first-choice tallies alone.

Remark 3 (Publishing approximate voting powers). Since delegation is fully private, Kite might face the problem of over-delegation, where one delegate accumulates sufficient voting power to cause a proposal to pass by this single delegate's vote [35]. One way to resolve this issue is to periodically post a vector of approximate voting power of all delegates. More specifically, the trusted authority members decrypt the voting power vector, add differentially private noise to the vector to hide who delegated to who, and publish the noisy vector on the blockchain. A similar issue arises in private voting systems more generally: since voters do not observe intermediate tallies, they cannot determine whether their vote is pivotal and may therefore abstain from voting. As with the delegation power vector, this can be mitigated by periodically publishing a noisy election tally but at the cost of reduced privacy.

Remark 4 (Split delegation). As pointed out in [35], it might be beneficial to allow split delegation. Although Kite supports delegation to only one delegate, it is possible to delegate to multiple delegates by splitting voting power across different accounts and delegating from each account.

Remark 5 (Monitoring the delegate). If Alice delegates her voting power to Bob and Bob subsequently becomes inactive or stops being a delegate, Alice's voting power becomes unused. To avoid this, Alice should monitor Bob's activity on the blockchain, which is publicly observable even in the private voting setting, and undelegate her voting power if necessary.

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.

Global contract parameters			
L_{eid}	List of election identifiers	t_v	Number of voting tokens owned by v
L_T	List of token counts of 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 election start	lock	Lock map
Election parameters		**Other notation**	
eid	Election identifier	$\mathsf{pk}_P^Q, \mathsf{sk}_P^Q$	Keys of party Q for primitive P
vote	Vote map	σ_{TA}	TA's signature on R_T
$desc_{eid}$	Election description		
E_i^{eid}	Encrypted votes for option i		
D_i^{eid}	Vote count for option i		

Table 1: Glossary

3.1 Setup

In this subprotocol (illustrated in Alg.1), TA begins by generating public keys for the encryption and signature schemes. The underlying plaintext space is $\mathbb{Z}_q$. To avoid modular reduction during homomorphic addition, we set the modulus q to be larger than the maximum anticipated sum of plaintext values.

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. Using a Merkle tree provides a clean abstraction for the security proofs, avoiding the need to reason directly about the Ethereum state. Additionally, it reduces gas costs by eliminating on-chain calls to verify individual token 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.

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. We denote the function for locking tokens by lock, so calling $\mathsf{lock}(p_i)$ locks the tokens held by voter p_i. The corresponding function unlock reverses this operation. 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. 2). 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, based on their active and token-locking status: unlocked and inactive (has not delegated),

Algorithm 1 Setup

function AUTHORITY SETUP(λ, L_T) // called by the trusted authority TA
  $\mathsf{pk}^{\mathsf{TA}}_{\mathsf{Enc}}, \mathsf{sk}^{\mathsf{TA}}_{\mathsf{Enc}} \leftarrow \mathsf{Enc.Gen}(1^\lambda, q)$ // where q is twice of all the voting powers
  $\mathsf{pk}^{\mathsf{TA}}_{\mathsf{Sig}}, \mathsf{sk}^{\mathsf{TA}}_{\mathsf{Sig}} \leftarrow \mathsf{Sig.Gen}(1^\lambda)$
  $R_T \leftarrow \mathsf{MT.GetRoot}(L_T)$
  $\sigma_T \leftarrow \mathsf{Sig.Sign}(\mathsf{sk}^{\mathsf{TA}}_{\mathsf{Sig}}, R_T)$
  $\mathsf{pk}_{\mathsf{zk}}, \mathsf{vk}_{\mathsf{zk}} \leftarrow \mathcal{G}(1^\lambda)$
  $L_T, R_T, L_{eid}, \boldsymbol{L_d}, L_{did}, \mathsf{lock}, \mathsf{active} \leftarrow$ ON CHAIN SETUP($\mathsf{pk}^{\mathsf{TA}}_{\mathsf{Enc}}, \mathsf{pk}^{\mathsf{TA}}_{\mathsf{Sig}}, L_T, R_T, \sigma_{\mathsf{TA}}, \mathsf{pk}_{\mathsf{zk}}, \mathsf{vk}_{\mathsf{zk}}$)
  store $\mathsf{sk}^{\mathsf{TA}}_{\mathsf{Enc}}, \mathsf{sk}^{\mathsf{TA}}_{\mathsf{Sig}}$
end function
function ON CHAIN SETUP($\mathsf{pk}^{\mathsf{TA}}_{\mathsf{Enc}}, \mathsf{pk}^{\mathsf{TA}}_{\mathsf{Sig}}, L_T\ R_T, \sigma_{\mathsf{TA}}, \mathsf{pk}_{\mathsf{zk}}, \mathsf{vk}_{\mathsf{zk}}$)
  $L_{eid} \leftarrow \emptyset$ // list of election identifiers
  $L_{did} \leftarrow \emptyset$ // list of delegation identifiers
  $\mathsf{lock} \leftarrow \mathbf{0}$ // lock map
  $\mathsf{active} \leftarrow \mathbf{0}$ // active delegate map
  $\boldsymbol{L_d} \leftarrow \mathbf{0}$ // list of delegates voting power, all-zero vector
  if $\mathsf{Sig.Verify}(\mathsf{pk}^{\mathsf{TA}}_{\mathsf{Sig}}, R_T, \sigma_T) = 1$ **then**
    $R_T \leftarrow R_T$ // save R_T in the contract state
    log $L_T, R_T, L_{eid}, \boldsymbol{L_d}, L_{did}, \mathsf{lock}, \mathsf{active}, \mathsf{pk}_{\mathsf{zk}}, \mathsf{vk}_{\mathsf{zk}}$
  else
    abort
  end if
end function
function VOTER SETUP(L_T) // called by p_i
  $t_{p_i} \leftarrow L_T[p_i]$ // store the number of tokens p_i has
  $\boldsymbol{ct} \leftarrow \mathbf{0}$ // initialize delegation vector
  store $t_{p_i}, \boldsymbol{ct}$
end function

unlocked and active (impossible), locked and inactive (delegated tokens), or locked and active (delegate).

3.3 Delegation and Undelegation

In the delegation subprotocol, Alg. 3, if a voter p_i wishes to delegate their voting power t_{p_i} to a delegate p_j, they create an encrypted vector $\boldsymbol{ct}$. It is an encryption of the all-zero vector, except for the entry corresponding to p_j. Additionally, the voter generates a proof that $\boldsymbol{ct}$ 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 $\boldsymbol{ct}$ 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 $\boldsymbol{ct}$, 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. 3, 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, $\boldsymbol{ct}$, that they used for delegation. Note that making the voter resubmit

Algorithm 2 Delegate Registration/Unregistration

function DELEGATE REGISTRATION(p_i) // called by p_i
  $p_i, \mathsf{lock}[p_i], \mathsf{active}[p_i], L_d[p_i] \leftarrow$ ON CHAIN DELEGATE REGISTRATION(p_i)
end function
function ON CHAIN DELEGATE REGISTRATION(p_i)
  if $\mathsf{active}[p_i] = 0 \wedge \mathsf{lock}[p_i] = 0$ **then**
    $\mathsf{lock}(p_i)$ // lock tokens of msg.sender in ERC20 contract
    $\mathsf{lock}[p_i] = 1$ // update lock map
    $\mathsf{active}[p_i] = 1$ // update active delegate map
    $t_{p_i} \leftarrow L_T[p_i]$
    $L_d[p_i] \leftarrow \mathsf{Enc.E}(\mathsf{pk}^{\mathsf{TA}}_{\mathsf{Enc}}, t_{p_i}; 0)$ // update p_i's voting power in the delegate list
  end if
  log $p_i, \mathsf{lock}[p_i], \mathsf{active}[p_i], L_d[p_i]$
end function
function DELEGATE UNREGISTRATION(p_i) // called by p_i
  $p_i, \mathsf{lock}[p_i], \mathsf{active}[p_i], L_d[p_i] \leftarrow$ ON CHAIN DELEGATE UNREGISTRATION(p_i)
end function
function ON CHAIN DELEGATE UNREGISTRATION(p_i)
  if $\mathsf{active}[p_i] = 1 \wedge \mathsf{lock}[p_i] = 1$ **then**
    $\mathsf{unlock}(p_i)$ // unlock tokens of msg.sender in ERC20 contract
    $\mathsf{lock}[p_i] = 0$ // update lock map
    $\mathsf{active}[p_i] = 0$ // update active delegate map
    $t_{p_i} \leftarrow L_T[p_i]$
    $e \leftarrow \mathsf{Enc.E}(\mathsf{pk}^{\mathsf{TA}}_{\mathsf{Enc}}, -t_{p_i}; 0)$
    $L_d[p_i] \leftarrow \mathsf{Enc.Add}(\mathsf{pk}^{\mathsf{TA}}_{\mathsf{Enc}}, L_d[p_i], e)$
  end if
  log $p_i, \mathsf{lock}[p_i], \mathsf{active}[p_i], L_d[p_i]$
end function

$\boldsymbol{ct}$ avoids storing the full ciphertext in the contract. 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 $\boldsymbol{ct}$. Once confirmed, the contract proceeds to homomorphically subtract $\boldsymbol{ct}$ from L_d and also removes the corresponding delegation identifier $\mathsf{H}(\boldsymbol{ct})$ from L_{did}.

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

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

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 $\boldsymbol{ct}$ only for that set. We discuss this in more detail in Section 5 and Appendix C.3.

Remark 6 (Topic-based delegation). While we focus on a single global delegation relation, the protocol can be extended to support topic-based delegation. In this variant, each topic is associated with a separate delegation vector. For example, Alice could delegate her voting power to Bob for proposals related to the US and to Charlie for proposals related to Europe by submitting two separate encrypted delegation vectors, one for each topic-delegate pair.

Witness and Public Statement for $\mathcal{R}_{\mathbf{del}}$			
Witness		**Public Statement**	
$p_j \in [m]$	Delegate's address and index	$\mathsf{pk}_{\mathsf{Enc}}^{\mathsf{TA}} \in \mathbb{G}$	Encryption public key of TA
$\boldsymbol{r} \in \mathbb{Z}_q^m$	Randomness vector for encryption	$\boldsymbol{ct} \in \mathbb{G}^m$	Delegation vector of encryptions
		$t_{p_i} \in \mathbb{Z}_q$	Voting power of voter p_i
		$R_T \in \mathbb{F}_p$	Root of Merkle tree of voting powers
		$\pi_{p_i} \in \mathbb{F}_p^{\log n}$	Merkle proof for element at index p_i
		$p_i \in [n]$	Voter's address and index

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

3.4 Election Setup and Election Start

Our election subprotocol is structured into two phases. The first phase begins with the election setup, initiated when the election details are made available on-chain. The second phase begins with the initiation of the voting process. Both phases are illustrated in Alg. 4. 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 the start of each election, 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 at the beginning of each election, we maintain a consistent reference point for the duration of that election. This design is particularly important when multiple elections run concurrently: each election relies on its own snapshot, preventing interference between overlapping elections with different start times. As a result, voters may freely transfer tokens without affecting the correctness of any ongoing tally, and the protocol avoids the unrealistic requirement of freezing token transfers during elections.

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. 5), 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.

While our protocol supports both private and public voting, delegation is kept private in all settings. The detailed algorithm for private voting is omitted

Algorithm 3 Delegation/Undelegation

function $\textsc{Delegation}(p_i, p_j)$ // called by p_i
  $\boldsymbol{r} \xleftarrow{\$} \mathbb{Z}_q^m$ // m is the number of registered delegates
  for $i \in len(L_d) \wedge i \neq p_j$ **do**
    $ct[i] \leftarrow \mathsf{Enc.E}(\mathsf{pk}_{\mathsf{Enc}}^{\mathsf{TA}}, 0; r_i)$
  end for
  $ct[p_j] \leftarrow \mathsf{Enc.E}(\mathsf{pk}_{\mathsf{Enc}}^{\mathsf{TA}}, t_{p_i}; r_j)$
  $\pi_{p_i} \leftarrow \mathsf{MT.GetProof}(L_T, p_i)$ // get a Merkle proof for a leaf with index p_i
  $\pi \leftarrow \mathsf{zkProve}_{\mathcal{R}_{\text{del}}}\left((\mathsf{pk}_{\mathsf{Enc.E}}^{\mathsf{TA}}, \boldsymbol{ct}, t_{p_i}, R_T, \pi_{p_i}, p_i), (p_j, \boldsymbol{r})\right)$
  $p_i, \mathsf{lock}[p_i], L_d, L_{did}[p_i] \leftarrow \textsc{On Chain Delegation}(p_i, \boldsymbol{ct}, \pi, \pi_{p_i})$
end function
function $\textsc{On Chain Delegation}(p_i, \boldsymbol{ct}, \pi, \pi_{p_i})$
  if $\mathsf{lock}[p_i] = 0$ **then** // verify if p_i's tokens are not locked
    if $\mathsf{zkVerify}_{\mathcal{R}_{\text{del}}}\left((\mathsf{pk}_{\mathsf{Enc.E}}^{\mathsf{TA}}, \boldsymbol{ct}, t_{p_i}, R_T, \pi_{p_i}, v), \pi\right) = 1$ **then**
      $\mathsf{lock}(p_i)$ // lock tokens of msg.sender in ERC20 contract
      $\mathsf{lock}[p_i] = 1$ // update lock map
      $L_d \leftarrow \mathsf{Enc.Add}(\mathsf{pk}_{\mathsf{Enc}}^{\mathsf{TA}}, L_d, \boldsymbol{ct})$ // homomorphic addition of encrypted vectors
      $L_{did}[p_i] \leftarrow \mathsf{H}(\boldsymbol{ct})$ // update the list of delegation identifiers
    end if
  end if
  log $p_i, \mathsf{lock}[p_i], L_d, L_{did}[p_i]$
end function
function $\textsc{Undelegation}(p_i, \boldsymbol{ct})$ // called by p_i
  $p_i, \mathsf{lock}[p_i], L_d, L_{did}[p_i] \leftarrow \textsc{On Chain Undelegation}(p_i, \boldsymbol{ct})$
end function
function $\textsc{On Chain Undelegation}(p_i, \boldsymbol{ct})$
  if $\mathsf{lock}[p_i] = 1$ **then**
    if $L_{did}[p_i] = \mathsf{H}(\boldsymbol{ct})$ **then**
      $\mathsf{unlock}(p_i)$ // unlock tokens of msg.sender in ERC20 contract
      $\mathsf{lock}[p_i] = 0$ // update lock map
      $L_d \leftarrow \mathsf{Enc.Add}(\mathsf{pk}_{\mathsf{Enc}}^{\mathsf{TA}}, L_d, -\boldsymbol{ct})$ // homomorphic subtraction of encrypted vectors
      $L_{did}[p_i] \leftarrow 0$ // update the list of delegation identifiers
    end if
  end if
  log $p_i, \mathsf{lock}[p_i], L_d, L_{did}[p_i]$
end function

here for brevity. Appendix B contains the full description. The remainder of this section focuses on public voting, which is the setting for our security analysis.

Remark 7 (Extension: Changing votes). We also note that Kite can be extended to support vote changes. In this case, whenever a voter submits a new vote, the contract subtracts the voter's previous contribution from the tally before adding the new one either directly in cleartext for public voting, or by homomorphically subtracting the previously submitted encrypted voting vector in the private voting setting.

3.6 Tally

In the Tally subprotocol, Alg. 6, 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 4 Election Setup and Start

function Election Setup(p_i, eid, $desc$) // called by p_i
  L_{eid}, $\boldsymbol{E}^{eid}$, vote_{eid}, R_{eid}, c_{eid} ← On Chain Election Setup(p_i, eid, $desc$)
end function
function On Chain Election Setup(v, eid, $desc$)
  if $eid \notin L_{eid}$ **then**
    $L_{eid} \leftarrow L_{eid} \cup eid$ // update the list of election identifiers
    $desc_{eid} \leftarrow desc$
    $\boldsymbol{ct_0} \leftarrow \mathsf{Enc.E}(\mathsf{pk}^{\mathsf{TA}}_{\mathsf{Enc}}, \mathbf{0}; \mathbf{0})$
    $\boldsymbol{E}^{eid} \leftarrow \boldsymbol{ct_0}$ // initialize the voting counters
    $\mathsf{vote}_{eid} \leftarrow \emptyset$ // initialize the election voting map
    $R_{eid} \leftarrow 0$ // initialize the voting power snapshot
    $c_{eid} \leftarrow p_i$ // save the election creator
    log L_{eid}, $\boldsymbol{E}^{eid}$, vote_{eid}, R_{eid}, c_{eid}
  else
    abort
  end if
end function
function Creator Election Start(p_i, eid) // called by p_i
  R_{eid} ← On Chain Election Start(p_i, eid)
  $L^{eid}_d \leftarrow L_d$
  return L^{eid}_d
end function
function On Chain Election Start(p_i, eid)
  if $c_{eid} = p_i$ **then** // check if p_i is the election creator
    $R_{eid} \leftarrow \mathsf{MT.GetRoot}(L_d)$ // the voting power snapshot
  end if
  log R_{eid}
end function

4 Security

4.1 The Universal Composability Framework

We analyze Kite using a variant of the Universal Composability (UC) framework called SUC [14], which simplifies UC by assuming a fixed set of parties and built-in authenticated channels. Security is defined by comparing executions in the real, ideal, and hybrid models: informally, a protocol is secure if no efficient environment can distinguish between interacting with the real protocol and with an ideal functionality. Subroutines are executed locally rather than as separate interactive Turing machines, which simplifies composition. These changes make SUC easier to apply while still supporting security proofs that can be translated to standard UC. In the interest of space, and since UC is a well-established model in cryptography, we defer a more detailed description and intuition around it to Appendix C.

4.2 Security Proof

We prove that Kite securely realizes an ideal functionality, $\mathcal{F}_{\mathsf{vote}}$. In contrast to the UC functionalities of prior work [37,38,39,27], our $\mathcal{F}_{\mathsf{vote}}$ is an SUC functionality that separates delegation from voting, enabling delegations to be long-lived across multiple elections. The proof is conducted in the $\mathcal{F}_{\mathsf{Contract}}$-hybrid model, where $\mathcal{F}_{\mathsf{Contract}}$ abstracts the bulletin board functionality provided by a smart contract. We assume a static adversary that may corrupt voters but not the trusted authority TA, and we work in the local random oracle model, which al-

Algorithm 5 Public Voting

function $\textsc{Voting}(eid, L^{eid}, p_i, v)$ // called by p_i
  if $v \in \{\text{yes}, \text{no}, \text{abstain}\}$ **then**
    $\pi_{p_i} \leftarrow \mathsf{MT.GetProof}(L_d^{eid}, p_i)$
    $\mathsf{vote}[p_i], E_v^{eid} \leftarrow \textsc{On Chain Voting}(eid, d, v, \pi_{p_i}, L_d^{eid}[p_i])$
  else
    abort
  end if
end function
function $\textsc{On Chain Voting}(eid, p_i, v, \pi_{p_i}, L_d^{eid}[p_i])$
  if $R_{eid} \neq 0 \wedge \mathsf{active}[p_i] = 1 \wedge \mathsf{vote}[p_i] = 0 \wedge \mathsf{MT.Verify}(L_d^{eid}[p_i], p_i, \pi_{p_i}, R_{eid}) = 1$ **then**
    $\mathsf{vote}[p_i] = 1$
    $E_v^{eid} \leftarrow \mathsf{Enc.Add}(\mathsf{pk}_{\mathsf{Enc}}^{\mathsf{TA}}, E_v^{eid}, L_d^{eid}[p_i])$ // homomorphic addition of the vote to the tally
    log $\mathsf{vote}[p_i], E_v^{eid}$
  else
    abort
  end if
end function

Algorithm 6 Tally

function $\textsc{Authority Tally}(\mathsf{sk}_{\mathsf{Enc}}^{\mathsf{TA}}, eid, \boldsymbol{E}^{eid})$ // called by TA
  $\boldsymbol{D}^{eid} \leftarrow \mathsf{Enc.D}(\mathsf{sk}_{\mathsf{Enc}}^{\mathsf{TA}}, \boldsymbol{E}^{eid})$
  for $i \in \{\text{yes}, \text{no}, \text{abstain}\}$ **do**
    $res_i^{eid} \leftarrow 100 \frac{D_i^{eid}}{\Sigma_{i=1}^{3} D_i^{eid}}$
  end for
  $res_{eid} \leftarrow \textsc{On Chain Tally}(eid, res^{eid})$
end function
function $\textsc{On Chain Tally}(eid, res^{eid})$
  log res^{eid}
end function

lows the simulator to program the random oracle [12]. Note that we can also use a global random oracle but with domain separation: each party and protocol instance queries the random oracle with a unique prefix. We note that our proof can be similarly adapted for the private voting scenario.

Theorem 1. Kite *(Sec. 3) SUC-securely realizes* $\mathcal{F}_{\mathsf{vote}}$ *with respect to* $\Pi^{\mathcal{F}_{\mathsf{Contract}}}$, *assuming a collision-resistant hash function, a CPA-secure encryption scheme, and a secure non-interactive zero-knowledge argument of knowledge.*

The full proof, including the formal functionalities $\mathcal{F}_{\mathsf{vote}}$ and $\mathcal{F}_{\mathsf{Contract}}$, as well as the simulator code and the simulation analysis, is provided in Appendix C.

5 Implementation

We developed a proof-of-concept implementation of Kite, supporting private delegation and public voting [1]. The implementation consists of a React frontend, two Rust servers for cryptographic operations and trusted authority functions, Noir circuits for zero-knowledge proofs, and on-chain Ethereum smart contracts extending OpenZeppelin ERC20 and Compound Governor Bravo. The Rust backend handles proof generation, ElGamal encryption, and homomorphic operations

[1] `https://anonymous.4open.science/r/GovernorPrivate-BCD5/README.md`

over ElGamal ciphertexts, while the trusted authority server decrypts final tallies and posts zk-proofs of correct decryption.

To improve efficiency and preserve privacy, Kite uses an *anonymity set* during delegation. When a voter delegates, the backend constructs a set of T delegate accounts that includes the intended delegate and $T-1$ randomly sampled other delegates. The voter posts encrypted voting power for all accounts in this set, along with a zero-knowledge proof of correctness. This ensures that observers know the voter delegated to someone in the set, but cannot identify the specific delegate. The size of the anonymity set balances privacy and efficiency: larger sets provide stronger privacy but increase proof size and on-chain costs, while smaller sets reduce costs at the expense of some privacy. Choosing $T = 1$ reduces to public delegation. This "hide within a set" approach has been used in prior voting protocols. For instance, in Selections[18], a voter's real password is hidden among a large set of panic passwords. In contrast to that setting, in Kite the size of the set primarily affects the adversary's success probability in guessing which delegate was chosen, rather than enabling a concrete attack. Even if the set is largely composed of corrupt delegates, the adversary cannot learn the specific recipient. It is also important to note that delegations are expected to be long-lived: a voter can keep an existing delegation to the same delegate across many elections, eliminating the need to resubmit fresh anonymity sets and thereby mitigating set-intersection linkability attacks.

All zero-knowledge relations are verified on-chain using smart contracts auto-generated from Noir, with custom optimizations to reduce proving and gas costs. For more details such as circuit design, smart contract logic, and ZK relation specifications, we refer the reader to Appendix C.3.

6 Evaluation

We evaluated the performance of our proof-of-concept implementation of Kite, focusing on zero-knowledge proof generation and verification. Proving times increase with the size of the anonymity set used for delegation: smaller sets provide faster proofs but less privacy, while larger sets increase proof time and memory. For typical anonymity sets of size 5, 10, or 20, proving times remain practical, ranging from a few seconds to just over a minute.

Other ZK relations, such as vote proofs and implementation-specific operations, are lightweight in comparison. On-chain verification costs are moderate, and combining multiple relations into a single proof reduces gas usage. Using a master verifier contract significantly lowers deployment costs compared to deploying individual verifier contracts. Overall, the system demonstrates reasonable performance on consumer-grade hardware. We also note that delegation, though more costly to prove, is infrequent, so typical user interactions remain low-latency.

The end-to-end implementation was deployed to a local Anvil testnet. We measured the gas cost of each operation required for end-to-end voting and delegation in our proof-of-concept implementation(Fig. 1). The gas cost associated with PLONK proof verification on chain is independent of the relation being verified, the average associated gas cost is represented by the red horizontal line.

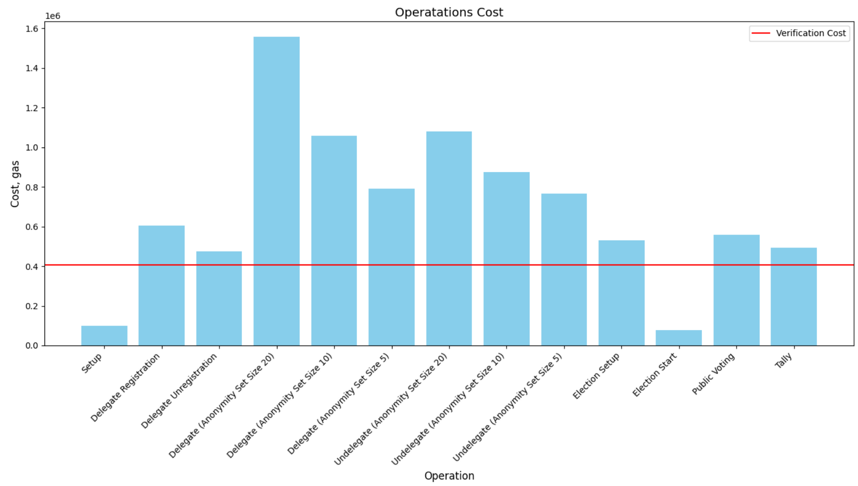


Fig. 1: Gas cost of all protocol operations.

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. For a small number of options (here, three), private voting would not incur significant additional gas overhead: its cost is comparable to delegation with an anonymity set of size three, and thus remains on the same order as public voting.

More results, including detailed proving times, circuit sizes, gas costs, and additional graphs are provided in Appendix C.6.

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 that delegate. Another important open direction is to achieve a stronger property of ballot secrecy which would require the integration of a coercion-resistant voting mechanism in Kite. Existing techniques, such as fake credentials, rely on hiding voters' true public keys and therefore do not directly apply to DAOs, where voting power is tied to public keys.

Independently, there has been growing interest in bringing our system to production, with various teams, including the Open Zeppelin team, reportedly working on a production-ready implementation.

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. Nounsdao private voting grant. `https://discourse.nouns.wtf/t/small-grants-zk-private-voting-for-nouns-dao/3405`.
5. 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.
6. 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.
7. 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.
8. 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.
9. C. Blum and C. I. Zuber. Liquid democracy: Potentials, problems, and perspectives. *Journal of political philosophy*, 24(2):162–182, 2016.
10. D. Boneh, X. Boyen, and S. Halevi. Chosen ciphertext secure public key threshold encryption without random oracles. In *CT-RSA*, volume 3860 of *Lecture Notes in Computer Science*, pages 226–243. Springer, 2006.
11. D. Boneh and V. Shoup. *A graduate course in applied cryptography (version 0.6)*. 2023. `cryptobook.us`.
12. 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.
13. R. Canetti. Universally composable security. *J. ACM*, 67(5):28:1–28:94, 2020.
14. 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.
15. R. Canetti and S. Goldwasser. An efficient *Threshold* public key cryptosystem secure against adaptive chosen ciphertext attack. In *EUROCRYPT*, volume 1592 of *Lecture Notes in Computer Science*, pages 90–106. Springer, 1999.
16. R. Carback, D. Chaum, J. Clark, J. Conway, A. Essex, P. S. Herrnson, 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 with ballot privacy. In *USENIX Security 2010*, pages 291–306, Washington, DC, USA, Aug. 11–13, 2010. USENIX Association.
17. D. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Commun. of the ACM*, 24(2):84–90, 1981.
18. J. Clark and U. Hengartner. Selections: Internet voting with over-the-shoulder coercion-resistance. In *Financial Cryptography*, volume 7035 of *Lecture Notes in Computer Science*, pages 47–61. Springer, 2011.
19. Y. Desmedt and Y. Frankel. Threshold cryptosystems. In *CRYPTO*, volume 435 of *Lecture Notes in Computer Science*, pages 307–315. Springer, 1989.
20. Open Zeppelin ERC20. `https://docs.openzeppelin.com/contracts/4.x/erc20`.

21. 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.
22. 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.
23. 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.
24. Open Zeppelin Governance. `https://docs.openzeppelin.com/contracts/4.x/api/governance`.
25. 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.
26. A. Hall. Governance FAQs, 2023.
27. L. Kovalchuk, B. Zhang, A. Nastenko, Z. Yin, R. Oliynykov, and M. Rodinko. Universally composable on-chain quadratic voting for liquid democracy. *IACR Cryptol. ePrint Arch.*, page 803, 2025.
28. O. Kulyk, K. Marky, S. Neumann, and M. Volkamer. Introducing proxy voting to helios. In *ARES*, pages 98–106. IEEE Computer Society, 2016.
29. O. Kulyk, S. Neumann, K. Marky, J. Budurushi, and M. Volkamer. Coercion-resistant proxy voting. *Comput. Secur.*, 71:88–99, 2017.
30. 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.
31. 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.
32. 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.
33. M. Nejadgholi, N. Yang, and J. Clark. Short paper: Ballot secrecy for liquid democracy. In *Financial Cryptography Workshops*, volume 12676 of *Lecture Notes in Computer Science*, pages 306–314. Springer, 2021.
34. Noir. `https://noir-lang.org`.
35. J. Strnad. Delegation and participation in decentralized governance: An epistemic view. 2025.
36. B. WhiteHat, J. Baylina, and M. Belles. Baby jubjub elliptic curve.
37. B. Zhang, R. Oliynykov, and H. Balogun. A treasury system for cryptocurrencies: Enabling better collaborative intelligence. In *NDSS*. The Internet Society, 2019.
38. B. Zhang and H. Zhou. Statement voting. In *Financial Cryptography*, volume 11598 of *Lecture Notes in Computer Science*, pages 667–685. Springer, 2019.
39. J. Zhang, B. Zhang, A. Nastenko, H. Balogun, and R. Oliynykov. Privacy-preserving decision-making over blockchain. *IEEE Trans. Dependable Secur. Comput.*, 20(6):4648–4663, 2023.
40. B. Zwattendorfer, C. Hillebold, and P. Teufl. Secure and privacy-preserving proxy voting system. In *ICEBE*, pages 472–477. IEEE Computer Society, 2013.

A Cryptographic Primitives

A.1 Additively Homomorphic Encryption

For our purposes, we require a additively homomorhpic **asymmetric encryption scheme** Enc comprised of five algorithms ($\mathsf{Enc.Gen}$, $\mathsf{Enc.E}$, $\mathsf{Enc.D}$, $\mathsf{Enc.Add}$, $\mathsf{Enc.Rerand}$) such that:

- $\mathsf{Enc.Gen}(1^\lambda, q) \to (\mathsf{pk}, \mathsf{sk})$: generates a public and private key pair.
- $\mathsf{Enc.E}(\mathsf{pk}, m; r) \to ct$: encrypts a message $m \in \mathbb{Z}_q$ with public key pk with randomness r.
- $\mathsf{Enc.D}(\mathsf{sk}, ct) \to m \in \mathbb{Z}_q$: decrypts ciphertext ct using secret key sk.
- $\mathsf{Enc.Add}(\mathsf{pk}, ct_1, ct_2) \to ct_+$: homomorphically adds ciphertexts ct_1 and ct_2 such that $\mathsf{Enc.D}(\mathsf{sk}, ct_1) + \mathsf{Enc.D}(\mathsf{sk}, ct_2) = \mathsf{Enc.D}(\mathsf{sk}, ct_+)$.
- $\mathsf{Enc.Rerand}(\mathsf{pk}, ct, r') \to ct'$: re-randomizes ct with new randomness r'.

We only require that the scheme be semantically secure against chosen plaintext attack, also known as CPA-secure [11, Ch. 5].

A.2 Digital Signature Scheme

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

- $\mathsf{Sig.Gen}(1^\lambda) \to (\mathsf{pk}, \mathsf{sk})$: generates a public and private key pair.
- $\mathsf{Sig.Sign}(\mathsf{sk}, m) \to \sigma$: signs message m with secret key sk.
- $\mathsf{Sig.Verify}(\mathsf{pk}, m, \sigma) \to b \in \{0, 1\}$: verifies signature σ.

We require that the scheme be existentially unforgeable under a chosen message attack [11, Ch. 13].

A.3 Hash Functions and Merkle Trees

A **hash function** H over $(\mathcal{K}, \mathcal{M}, \mathcal{T})$ is a pair of algorithms ($\mathsf{H.Gen}$, H) such that:

- $\mathsf{H.Gen}(1^\lambda)$: generates key $s \in \mathcal{K}$; that selects a hash function from the hash family.
- $\mathsf{H}(s, x)$: outputs a string $x' \in \mathcal{T}$, which is the evaluation of $\mathsf{H}(s, \cdot)$ at $x \in \mathcal{M}$.

We require that the hash function is collision resistant, meaning it is infeasible to find two different inputs that produce the same hash value [11, Ch. 8]. Additionally, we will use hash functions to construct hash trees, also known as Merkle trees [11, Ch. 8].

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

- $\mathsf{MT.GetRoot}(T) \to R$: computes the root R of the tree T
- $\mathsf{MT.GetProof}(T, i) \to \pi_i$: computes the proof π_i for the leaf with index i
- $\mathsf{MT.Verify}(T[i], i, \pi_i, R) \to b \in \{0, 1\}$: verifies the proof π_i for the leaf $T[i]$ with index i against the root R

A.4 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 $(x, w) \in \mathcal{R}$:

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

- **Knowledge Soundess**: 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 $\mathcal{A}_0, \mathcal{A}_1$:

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

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

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

$$\mathsf{Adv}^{\mathrm{zk}}_{\mathcal{A},\Pi}(\lambda) := \Big| \Pr\big[\mathcal{A}^H\big(\mathsf{pp}, x, P^H(\mathsf{pp}, x, w)\big) = 1\big] - \Pr\big[\mathcal{A}^{H[h]}\big(\mathsf{pp}, x, \pi\big) = 1\big] \Big|$$

  where $\mathsf{pp} \xleftarrow{\$} \mathsf{setup}(1^\lambda)$ and $(\pi, h) \xleftarrow{\$} \Pi.\mathcal{S}(\mathsf{pp}, x)$. Here h is a partial function $h : \mathcal{X} \to \mathcal{Y}$ output by $\Pi.\mathcal{S}$, and $H[h]$ refers to the oracle $H : \mathcal{X} \to \mathcal{Y}$ modified by entries in h. That is, we allow $\Pi.\mathcal{S}$ 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 $|x|$.

B Private Voting

In the private voting scenario, Alg. 7, the process for a delegate to cast their vote is more nuanced compared to the public vote. The delegate must submit an encrypted vote for each option, a vector $\boldsymbol{E}$. The key difference is that for the option they select, they use the encryption of their voting power, while for all

Witness and Public Statement for $\mathcal{R}_{\text{vote}}$			
Witness		**Public Statement**	
$v \in [3]$	A vote	$\mathsf{pk}^{\mathsf{TA}}_{\mathsf{Enc}} \in \mathbb{G}$	Encryption public key of TA
$\boldsymbol{r} \in \mathbb{Z}_q^3$	Randomness vector for encryption	$L_d^{eid}[p_i] \in \mathbb{G}$	Encrypted voting power of delegate p_i
		$\boldsymbol{E} \in \mathbb{G}^3$	Encrypted vector of votes
		$p_i \in [m]$	Delegate's address and index
		$R_{eid} \in \mathbb{F}_p$	Root of Merkle tree for election *eid*
		$\pi_{p_i} \in \mathbb{F}_p^{\log m}$	Merkle proof for element at index p_i

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

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:

$$\mathcal{R}_{\text{vote}} := \Big\{(v, \boldsymbol{r}), (\mathsf{pk}^{\mathsf{TA}}_{\mathsf{Enc}}, L_d^{eid}[p_i], \boldsymbol{E}, p_i, R_{eid}, \pi_{p_i}) \mid E_v = \mathsf{Enc.Rerand}(\mathsf{pk}^{\mathsf{TA}}_{\mathsf{Enc}}, L_d^{eid}[p_i], r_v) \wedge$$

$$\forall j \neq v : E_j \leftarrow \mathsf{Enc.E}(\mathsf{pk}^{\mathsf{TA}}_{\mathsf{Enc}}, 0; r_j) \wedge \mathsf{MT.Verify}(L_d^{eid}[p_i], p_i, \pi_{p_i}, R_{eid}) = 1\Big\} \quad (2)$$

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

C Security Proof

C.1 Universal Composability Framework

For our security analysis, we use a variant of the UC framework called SUC [14]. 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 $p_1, \ldots, p_n$, and an ideal process f (sometimes called an ideal functionality) . The environment $\mathcal{Z}$ writes inputs to parties $p_1, \ldots, p_n$, 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.

Algorithm 7 Private Voting

function VOTING(eid, L^{eid}, p_i, v) // called by p_i
  if $v \in \{\text{yes}, \text{no}, \text{abstain}\}$ **then**
    $\boldsymbol{r} \xleftarrow{\$} \mathbb{Z}_q^3$
    $E_v \leftarrow \mathsf{Enc.Rerand}(\mathsf{pk}^{\mathsf{TA}}_{\mathsf{Enc}}, L^{eid}_d[p_i], r_v)$
    $\forall j \neq v : E_j \leftarrow \mathsf{Enc.E}(\mathsf{pk}^{\mathsf{TA}}_{\mathsf{Enc}}, 0; r_j)$
    $\pi_{p_i} \leftarrow \mathsf{MT.GetProof}(L^{eid}_d, p_i)$
    $\pi \leftarrow \mathsf{zkProve}_{\mathcal{R}_{\text{vote}}}\left((\mathsf{pk}^{\mathsf{TA}}_{\mathsf{Enc}}, L^{eid}_d[p_i], \boldsymbol{E}, p_i, R_{eid}, \pi_{p_i}), (v, r)\right)$
    $\mathsf{vote}[p_i], \boldsymbol{E}^{eid} \leftarrow$ ON CHAIN VOTING($eid, p_i, \boldsymbol{E}, \pi, \pi_{p_i}, L^{eid}_d[p_i]$)
  else
    abort
  end if
end function
function ON CHAIN VOTING(eid)
  if $R_{eid} \neq 0 \wedge \mathsf{active}[p_i] = 1 \wedge \mathsf{vote}[p_i] = 0 \wedge \mathsf{zkVerify}_{\mathcal{R}_{\text{vote}}}\left((\mathsf{pk}^{\mathsf{TA}}_{\mathsf{Enc}}, L^{eid}_d[p_i], \boldsymbol{E}, p_i, R_{eid}, \pi_{p_i}), \pi\right) = 1$ **then**
    $\mathsf{vote}[p_i] = 1$
    for $j \in \{\text{yes}, \text{no}, \text{abstain}\}$ **do**
      $E^{eid}_j \leftarrow \mathsf{Enc.Add}(\mathsf{pk}^{\mathsf{TA}}_{\mathsf{Enc}}, E^{eid}_v, E_v)$
    end for
    log $\mathsf{vote}[p_i], \boldsymbol{E}^{eid}$
  else
    abort
  end if
end function

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, [14] 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.

C.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 [12]. To prove the security of our protocol, we start by defining an ideal process for the voting process, ideal functionality $\mathcal{F}_{\mathsf{vote}}$ (defined in Alg. 8). 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 $\mathcal{F}_{\mathsf{Contract}}$ (Alg. 9), that captures the computing bulletin board. Operating within the $\mathcal{F}_{\mathsf{Contract}}$-hybrid model, we abstract the bulletin board as a functionality that performs only the necessary computations for voting. We emphasize that $\mathcal{F}_{\mathsf{Contract}}$ 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.

Algorithm 8 Functionality $\mathcal{F}_{\mathsf{vote}}(\text{aux})$

1: Functionality $\mathcal{F}_{\mathsf{vote}}(\text{aux}, T)$ runs with voters $p_1, \ldots, p_n \in \mathcal{P}$, trusted authority TA and adversary $\mathcal{A}$. For every party $p_i \in \mathcal{P}, |\mathcal{P}| = n$, the functionality maintains a bit $\mathsf{reg}_i \in \{0,1\}$, integers $d_i \in [n]$ and $t_i \in [B]$. For initialization, set $\mathsf{reg}_i = 0$, $d_i := i$ 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 $(\mathsf{register})$ from p_i, set $\mathsf{reg}_i := 1$. Send $(\mathsf{register}, p_i)$ to all participants.

4: – Upon receiving $(\mathsf{unregister})$ from p_i, set $\mathsf{reg}_i := 0$. Send $(\mathsf{unregister}, p_i)$ to all participants.

5: – Upon receiving $(\mathsf{delegate}, p_j)$ from p_i, check whether $\mathsf{reg}_i = 0$. In this case, set $d_i := j$. Send $(\mathsf{delegate}, p_i)$ to all participants.

6: – Upon receiving $(\mathsf{undelegate})$ from p_i, set $d_i := i$. Send $(\mathsf{undelegate}, p_i)$ to all participants.

7: – Upon receiving $(\mathsf{election\ setup}, desc, eid)$ from p_i, set a bit $\mathsf{vote}_j^{eid} := 0$ and an integer $t_j^{eid} = 0$ for every p_j and a triple of integers $r^{eid} = (0,0,0)$. Store (eid, p_i). Send $(\mathsf{election\ setup}, desc, eid, p_i)$ to all participants.

8: – Upon receiving $(\mathsf{election\ start}, eid)$ 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 $t_j^{eid} := t_j^{eid} + t_i$

10: **else** set $t_i^{eid} := t_i^{eid} + t_i$ **end if**

11: Send $(\mathsf{election\ start}, eid, p_i)$ to all participants.

12: – Upon receiving (vote, eid, v) from p_i, check whether $\mathsf{reg}_i = 1$, $\mathsf{vote}_i^{eid} = 0$, and $v \in \{\text{yes}, \text{no}, \text{abstain}\}$. In this case, set $\mathsf{vote}_i^{eid} := 1$ and $r^{eid}[v] := r^{eid}[v] + t_i^{eid}$. Send $(\mathsf{vote}, eid, v, p_i)$ to all participants.

13: – Upon receiving (tally, eid) from TA, for every v set $r_v^{eid} = 100\frac{r_v^{eid}}{\sum_{v=1}^{3} r_v^{eid}}$ and send $(\mathsf{tally}, eid, r^{eid})$ to all participants.

Next, we build a simulator, as defined in Alg. 10. The simulator exists within the ideal model. There, the trusted authority supplies $\mathcal{F}_{\mathsf{vote}}$ with the token distribution, L_T. The functionality then subsequently broadcasts L_T to all participants. Upon receiving the token distribution from $\mathcal{F}_{\mathsf{vote}}$, $\mathcal{S}$ 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. 10). 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. 10). Possessing the secret key of the authority, it can decrypt the delegation vector $\boldsymbol{ct}$ sent by the adversary and relay the delegate's address to the functionality in clear. Conversely, for each message received from the ideal

Algorithm 9 On Chain Contract Functionality $\mathcal{F}_{\mathsf{Contract}}$

Functionality $\mathcal{F}_{\mathsf{Contract}}$ runs with voters $p_1, \ldots, p_n \in \mathcal{P}$ and trusted authority TA.
– Upon receiving $(\mathsf{setup}, \mathsf{pk}^A_{\mathsf{Enc}}, \mathsf{pk}^A_{\mathsf{Sig}}, L_T, \sigma_T)$ from TA, execute ON CHAIN SETUP$(L_v, \mathsf{pk}^A_{\mathsf{Enc}}, \mathsf{pk}^A_{\mathsf{Sig}}, L_T, \sigma_T)$. Send output to all participants.
– Upon receiving $(\mathsf{register})$ from p_i, execute ON CHAIN DELEGATE REGISTRATION(p_i). Send output to all participants.
– Upon receiving $(\mathsf{unregister})$ from p_i, execute ON CHAIN DELEGATE UNREGISTRATION(p_i). Send output to all participants.
– Upon receiving $(\mathsf{delegate}, \boldsymbol{ct}, \pi, \pi_{p_i})$ from p_i, execute ON CHAIN DELEGATION$(p_i, \boldsymbol{ct}, \pi, \pi_{p_i})$. Send output to all participants.
– Upon receiving $(\mathsf{undelegate}, \boldsymbol{ct})$, from p_i, execute ON CHAIN UNDELEGATION$(p_i, \boldsymbol{ct})$. Send output to all participants.
– Upon receiving $(\mathsf{election\ setup}, eid, desc)$ from p_i, execute ON CHAIN ELECTION SETUP$(p_i, eid, desc)$. Send output to all participants.
– Upon receiving $(\mathsf{election\ start}, eid)$ from p_i, execute ON CHAIN ELECTION START(p_i, eid). Send output to all participants.
– Upon receiving $(\mathsf{vote}, eid, p_i, v, \pi_{p_i}, L_d^{eid}[p_i])$ from p_i, execute ON CHAIN VOTING$(eid, p_i, v, \pi_{p_i}, L_d^{eid}[p_i])$. Send output to all participants.
– Upon receiving $(\mathsf{tally}, eid, res^{eid})$ from TA, execute ON CHAIN TALLY(eid, res^{eid}). Send output to all participants.

functionality, the simulator, on behalf of an honest party, sends a corresponding message to the adversary (lines 12-20 of Alg. 10). In the delegation step, $\mathcal{S}$ does not know the delegate's address, only the fact that the voter has delegated. Therefore, to simulate $\boldsymbol{ct}$, it encrypts an all-zero vector and simulates the proof of correctness. For tallying, it takes the result it received from $\mathcal{F}_{\mathsf{vote}}$ and simulates a proof of correct decryption.

Finally, we are ready to prove our main security theorem.

Theorem 2. Kite *(Sec. 3) SUC-securely realizes* $\mathcal{F}_{\mathsf{vote}}$ *with respect to* $\Pi^{\mathcal{F}_{\mathsf{Contract}}}$, *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.

$\mathsf{SUC}-\mathsf{HYBRID}^{\mathcal{F}_{\mathsf{Contract}}}_{\Pi,\mathcal{A},\mathcal{Z}} \approx \mathsf{EXP}_0$. We argue that the protocol and Experiment 0 are indistinguishable to $\mathcal{Z}$. To show that, assume the 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 $\mathsf{SUC}-\mathsf{HYBRID}^{\mathcal{F}_{\mathsf{Contract}}}_{\Pi,\mathcal{A},\mathcal{Z}}$. 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 $\mathsf{H}(p_i) = \mathsf{H}(p'_i)$. 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 $\boldsymbol{ct}$ in the undelegation step.

$\mathsf{EXP}_0 \approx \mathsf{EXP}_1$. 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 $\boldsymbol{ct}$ such that $\boldsymbol{ct} \neq \boldsymbol{ct}_i$ but $\mathsf{H}(\boldsymbol{ct}_i) = \mathsf{H}(\boldsymbol{ct})$. Again, we can build a new adversary $\mathcal{B}$, that uses $\mathcal{Z}$ to find $\boldsymbol{ct}, \boldsymbol{ct}_i$ such that $\boldsymbol{ct} \neq \boldsymbol{ct}_i$ but $\mathsf{H}(\boldsymbol{ct}_i) = \mathsf{H}(\boldsymbol{ct})$. 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 $\mathcal{S}$.

$\mathsf{EXP}_2 \approx \mathsf{EXP}_1$. 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.

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

$\mathsf{EXP}_3 \approx \mathsf{EXP}_2$. 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 $\mathsf{Enc.E}(\mathsf{pk}^{\mathsf{TA}}_{\mathsf{Enc}}, \mathbf{0}; \boldsymbol{r})$ and $\mathsf{Enc.E}(\mathsf{pk}^{\mathsf{TA}}_{\mathsf{Enc}}, \boldsymbol{t}; \boldsymbol{r'})$. Due to the CPA security of the encryption scheme, this can happen with a negligible probability only.

$\mathsf{EXP}_3 \approx \mathsf{SUC} - \mathsf{IDEAL}_{\mathcal{F}_{\mathsf{vote}}, \mathcal{S}, \mathcal{Z}}$. 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.

C.3 Implementation and Evaluation

C.4 System Implementation Details

We developed a proof-of-concept implementation of Kite, 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 Zeppelin 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 thier 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, where T is chosen by the voter 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. Note that if the voter chooses $T = 1$, then it essentially become public delegation.

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.

C.5 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 $\mathcal{R}_{\text{dec}}$ relation.

$$\mathcal{R}_{\text{dec}} := \left\{ \left(\mathsf{sk}_{\mathsf{Enc}}^{\mathsf{TA}},\ (\mathsf{pk}_{\mathsf{Enc}}^{\mathsf{TA}}, \boldsymbol{E}^{eid}, res^{eid})\right) \quad \middle| \quad \text{for } i \in [3]\text{: } D_i^{eid} \leftarrow \mathsf{Enc.D}(\mathsf{sk}_{\mathsf{Enc}}^{\mathsf{TA}}, E_i^{eid}) \wedge \\ res_i^{eid} = 100 \frac{D_i^{eid}}{\sum_{i=1}^{3} D_i^{eid}}, \right\}$$

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

We implemented all zero-knowledge relations using the Noir language [34]. 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 [22]. Noir is a powerful tool for rapid code 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 [36]. 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:

$$\mathcal{R}_{\text{addmt}} := \{(\emptyset), (\mathsf{pk}_{\mathsf{Enc}}^{\mathsf{TA}}, ct_1, ct_2, ct_+, \pi_{p_i}, R) \mid \mathsf{MT.Verify}(ct_2, p_i, \pi_{p_i}, R) = 1 \wedge ct_+ = \mathsf{Enc.Add}(\mathsf{pk}_{\mathsf{Enc}}^{\mathsf{TA}}, ct_1, ct_2)\}$$
$$\mathcal{R}_{\text{vecsub}} := \{(\emptyset), (\mathsf{pk}_{\mathsf{Enc}}^{\mathsf{TA}}, \boldsymbol{ct_1}, \boldsymbol{ct_2}, \boldsymbol{ct_-}) \mid \boldsymbol{ct_-} = \mathsf{Enc.Add}(\mathsf{pk}_{\mathsf{Enc}}^{\mathsf{TA}}, \boldsymbol{ct_1}, -\boldsymbol{ct_2}))\}$$
$$\mathcal{R}_{\text{vecadd}} := \{(\emptyset), (\mathsf{pk}_{\mathsf{Enc}}^{\mathsf{TA}}, \boldsymbol{ct_1}, \boldsymbol{ct_2}, \boldsymbol{ct_+}) \mid \boldsymbol{ct_-} = \mathsf{Enc.Add}(\mathsf{pk}_{\mathsf{Enc}}^{\mathsf{TA}}, \boldsymbol{ct_1}, \boldsymbol{ct_2}))\}$$
$$\mathcal{R}_{\text{encsub}} := \{(\emptyset), (\mathsf{pk}_{\mathsf{Enc}}^{\mathsf{TA}}, t, ct, ct_-) \mid e = \mathsf{Enc.E}(\mathsf{pk}_{\mathsf{Enc}}^{\mathsf{TA}}, -t; 0) \wedge ct_- = \mathsf{Enc.Add}(\mathsf{pk}_{\mathsf{Enc}}^{\mathsf{TA}}, ct, e)\}$$
$$\mathcal{R}_{\text{enc}} = \{(\emptyset), (\mathsf{pk}_{\mathsf{Enc}}^{\mathsf{TA}}, ct, t, r) \mid ct = \mathsf{Enc.E}(\mathsf{pk}_{\mathsf{Enc}}^{\mathsf{TA}}, t; r)\}$$

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

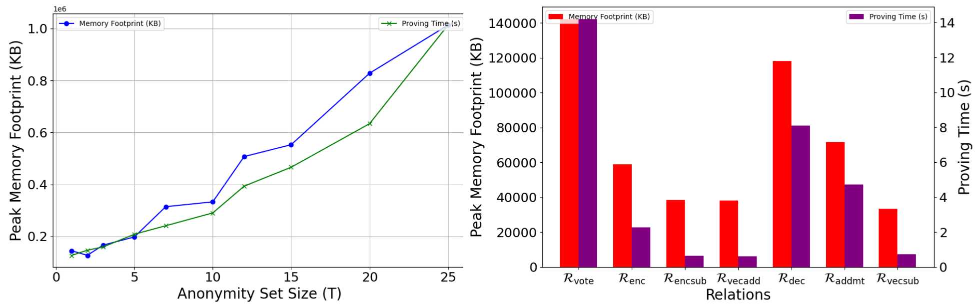
All of the relations we use must be verified on-chain. Noir's tooling allows for the generation of smart contracts with functionality to load a relation-specific verification key and verify a set of public inputs against a proof, with each contract corresponding to a single relation. We extend these 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 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 $\mathcal{R}_{\mathrm{del}}$ depends on the size of the anonymity set, defined in Section 5. 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 [25] 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. 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.

C.6 Evaluation Details

Our performance figures were collected on a consumer-grade device (MacBook Air) running an 8-core Apple M2 chip with 8GB of RAM to emulate the conditions end-users of the protocol may observe. We provide proving time and peak memory footprint data for each of the relations in our proof-of-concept implementation (Figures 2a and 2b), as well as gas cost data for each step of the protocol in section 5.

All proving times are computed as an average of ten runs using the Nargo CLI provided by Aztec Labs. For $\mathcal{R}_{\mathrm{del}}$, we report the metrics as a function of



(a) Proving time of $\mathcal{R}_{\text{del}}$ and peak memory footprint of proof generation vs anonymity set size.

(b) Proving times and peak memory footprints of proof generation of all relations, except $\mathcal{R}_{\text{del}}$.

Fig. 2: Proving time and peak memory footprint of proof generation.

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 $\mathcal{R}_{\text{del}}$ 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. 2a). We observed that the inclusion of a small-scalar multiplication implementation reduces the proving time of $\mathcal{R}_{\text{del}}$ by an average of 42.61% across all anonymity set sizes. The results for $\mathcal{R}_{\text{del}}$ also provide insight into the cost of private voting, as casting a vote with a small number of options incurs roughly similar or lower proving costs than generating a delegation vector of that size. Out of the other relations, $\mathcal{R}_{\text{vote}}$ is the most costly, with the additional implementation-specific and protocol-independent relations $\mathcal{R}_{\text{addmt}}, \mathcal{R}_{\text{vecsub}}, \mathcal{R}_{\text{vecadd}}, \mathcal{R}_{\text{encsub}}$, and $\mathcal{R}_{\text{enc}}$ claiming only marginal resources in comparison (Fig. 2b). The proving times of the vector-valued $\mathcal{R}_{\text{vecsub}}$ and $\mathcal{R}_{\text{vecadd}}$ 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, and find that the order of relations by proving time is the same as their circuit size. The largest circuit we implemented was $\mathcal{R}_{\text{del}}$ 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 $\mathcal{R}_{\text{vecadd}}$ 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 $\mathcal{R}_{\text{del}}$ and $\mathcal{R}_{\text{vecsub}}$, 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 seperately 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.

Algorithm 10 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 $\boldsymbol{ct}_i := 0$. Let $\mathcal{A}$ corrupt a subset $I \in \mathcal{P}$.

2: – Upon receiving (setup, L_T) from $\mathcal{F}_{\mathsf{vote}}$, $\mathcal{S}$ sets $t_i = L_T[i]$ for all $i \in [n]$ and runs Authority Setup to obtain $\mathsf{pk}^{\mathsf{TA}}_{\mathsf{Enc}}$, $\mathsf{pk}^{\mathsf{TA}}_{\mathsf{Sig}}$, L_T, σ_T and sends the output to all participants. Additionally, $\mathcal{S}$ knows the secret keys $\mathsf{sk}^{\mathsf{TA}}_{\mathsf{Enc}}$ and $\mathsf{sk}^{\mathsf{TA}}_{\mathsf{Sig}}$. $\mathcal{S}$ runs On Chain Setup$(\mathsf{pk}^{\mathsf{TA}}_{\mathsf{Enc}}, \mathsf{pk}^{\mathsf{TA}}_{\mathsf{Sig}}, L_T, \sigma_T)$ and sends the output to all participants.

3: – $\mathcal{S}$ invokes $\mathcal{A}$ and simulates $\mathcal{F}_{\mathsf{Contract}}$ on valid calls, ignores all invalid calls.

4: – **For every** $p_i \in I$**:**

5: – Upon receiving $(\mathsf{register})$ from p_i, $\mathcal{S}$ sends $(\mathsf{register})$ on behalf of p_i to $\mathcal{F}_{\mathsf{vote}}$

6: – Upon receiving $(\mathsf{unregister})$ from p_i, $\mathcal{S}$ sends $(\mathsf{unregister})$ on behalf of p_i to $\mathcal{F}_{\mathsf{vote}}$

7: – Upon receiving $(\mathsf{delegate}, \boldsymbol{ct}, \pi, \pi_{p_i})$ 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 $\boldsymbol{ct}$. $\mathcal{S}$ sets $\boldsymbol{ct}_i = \boldsymbol{ct}$ and sends $(\mathsf{delegate}, p_j)$ on behalf of p_i to $\mathcal{F}_{\mathsf{vote}}$, where p_j is such that $\boldsymbol{ct}[p_j] \neq 0$.

8: – Upon receiving $(\mathsf{undelegate}, \boldsymbol{ct})$ from p_i. If $\boldsymbol{ct}_i \neq \boldsymbol{ct}$ but $\mathsf{H}(\boldsymbol{ct}_i) = \mathsf{H}(\boldsymbol{ct})$, then $\mathcal{S}$ aborts. If $\boldsymbol{ct}_i = \boldsymbol{ct}$, it sends $(\mathsf{undelegate})$ on behalf of p_i to $\mathcal{F}_{\mathsf{vote}}$.

9: – Upon receiving $(\mathsf{election\ setup}, eid, desc)$ from p_i, $\mathcal{S}$ sends $(\mathsf{election\ setup}, desc, eid)$ on behalf of p_i to $\mathcal{F}_{\mathsf{vote}}$

10: – Upon receiving $(\mathsf{election\ start}, eid)$ from p_i, $\mathcal{S}$ sends $(\mathsf{election\ start}, eid)$ on behalf of p_i to $\mathcal{F}_{\mathsf{vote}}$

11: – Upon receiving $(\mathsf{vote}, eid, p_i, v, \pi_{p_i}, L_d^{eid}[p_i])$ 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 $L_d^{eid}[p_i]$, $\mathcal{S}$ aborts. Otherwise, $\mathcal{S}$ sends (vote, eid, v) on behalf of p_i to $\mathcal{F}_{\mathsf{vote}}$.

12: – **For every** $p_i \in \mathcal{P} \setminus I$**:**

13: – Upon receiving $(\mathsf{register}, p_i)$ from $\mathcal{F}_{\mathsf{vote}}$, $\mathcal{S}$ executes On Chain Delegate Registration(p_i) and sends the output to $\mathcal{A}$.

14: – Upon receiving $(\mathsf{unregister}, p_i)$ from $\mathcal{F}_{\mathsf{vote}}$, $\mathcal{S}$ executes On Chain Delegate Unregistration(p_i) and sends the output to $\mathcal{A}$.

15: – Upon receiving $(\mathsf{delegate}, p_i)$ from $\mathcal{F}_{\mathsf{vote}}$, $\mathcal{S}$ produces $\boldsymbol{ct}$ 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$(p_i, \boldsymbol{ct}, \pi, \pi_{p_i})$ and sends the output to $\mathcal{A}$.

16: – Upon receiving $(\mathsf{undelegate}, p_i)$ from $\mathcal{F}_{\mathsf{vote}}$, $\mathcal{S}$ executes On Chain Undelegation$(p_i, \boldsymbol{ct})$ and sends the output to $\mathcal{A}$.

17: – Upon receiving $(\mathsf{election\ setup}, desc, eid, p_i)$ from $\mathcal{F}_{\mathsf{vote}}$, $\mathcal{S}$ executes On Chain Election Setup$(p_i, eid, desc)$ and sends the output to $\mathcal{A}$.

18: – Upon receiving $(\mathsf{election\ start}, eid, p_i)$ from $\mathcal{F}_{\mathsf{vote}}$, $\mathcal{S}$ executes On Chain Election Start(p_i, eid) and sends the output to $\mathcal{A}$.

19: – Upon receiving $(\mathsf{vote}, eid, p_i, v, p_i)$ from $\mathcal{F}_{\mathsf{vote}}$, $\mathcal{S}$ generates a Merkle proof π_{p_i}, executes On Chain Voting$(eid, p_i, v, \pi_{p_i}, L_d^{eid}[p_i])$ and sends the output to $\mathcal{A}$.

20: – Upon receiving $(\mathsf{tally}, eid, r^{eid})$ from $\mathcal{F}_{\mathsf{vote}}$, $\mathcal{S}$ simulates a zk-proof π. Then, S executes On Chain Tally(eid, res^{eid}, π) and sends the output to $\mathcal{A}$.