The zkVot Protocol: A Distributed Computation Protocol for Censorship Resistant Anonymous Voting

Yunus Gürlek
yunus@node101.io

Kadircan Bozkurt
kadircan@node101.io

January 15, 2025

Abstract

zkVot is a client side trustless distributed computation protocol that utilizes zero knowledge proving technology. It is designed to achieve anonymous and censorship resistant voting while ensuring scalability. The protocol is created as an example of how modular and distributed computation can improve both the decentralization and the scalability of the internet.

A complete and working implementation of this paper is available here, and the demo can be tried in this link. It is important to emphasize that zkVot is one of the first complete implementations of a fully censorship resistant anonymous voting application that can scale up to a meaningful number of voters.[1]

1 Introduction

zkVot brings various distributed layers (e.g. blockchains), zero knowledge proving technology, and client side computation together to make most of the distributed value on an actual use case. By implementing this project and writing this article, our main goal is to show that the technology is ready, and it is just a matter of time and perspective to bring decentralization into the life of the actual end user.

Even though some solutions presented here may seem specific to a voting application, actually it is possible to extend concepts across different domains and solve various problems by embracing the same approach. And with this consideration, maybe it is even more appropriate to describe zkVot as a protocol more than a product: In an internet that is fully designed on top of censorship resistant layers and distributed computation, zkVot is one of the first utilizers of a transfer protocol that we want to create with the hope of a better internet.

2 Background and Motivation

The current internet is mostly built using a transfer protocol named HTTP[17], controlling all the interactions between different parties of the network. In the most common set up, two parties exist in a web application: the client and the server. However, modern web applications go much beyond this basic model to optimize the performance and the security. Instead of creating a single server with a lot of responsibilities, applications divide tasks among different layers that are optimized for some specific purpose. For instance, complex applications use at least a few different types of data storages to improve caching and filtering, or various ASPs (Application Service Providers) to securely perform different tasks. Moreover, clients of the system can also choose to use additional layers while interacting with the application, like VPNs to make their interactions partially private.

Here, it is important to emphasize that all the interactions between various layers are always controlled by a protocol like HTTP. As a matter of fact, the generic and scalable design of these protocols is the reason behind the success of the internet in today's world.

However, unfortunately almost all layers in today's internet are controlled by some central authorities, and this centralized design of the internet harms user privacy, creates a risk of censorship, and limits scalability.

zkVot is a similar transfer protocol designed for online governance using decentralized networks. As well as fulfilling the UX requirements of the current internet, elections using the zkVot Protocol

[1] The current testing estimates that 1 million votes can be counted under 6 hours along with the associated zero knowledge proof without any significant trust assumption.

provide full anonymity and censorship resistance, which is practically impossible with the current internet. Moreover, the distributed approach of zkVot allows elections to scale with every new user joining the protocol, unleashing a scalability potential that has never been possible before.

3 Problem Definition: An Ideal Voting Application

In an ideal online voting application, it is desired to achieve following properties:

1. Every user's (a.k.a. voters) identifier (e.g. public key) must be checked to be in a predefined public set, to allow the verifiability of the election's integrity by anyone.
2. No one should be able to follow a vote back to the voter during the election.
3. The anonymity of each voter must be preserved even long after the election is complete.
4. No one should be able to censor the counting of a vote based on any criteria.
5. Every vote should be counted only once (i.e., no double voting).
6. No one should have any doubt about the result of the election.
7. The election results must be preserved in a public, verifiable and persistent network.
8. The election must end at a predetermined and public time.
9. The system must be scalable at least up to a meaningful number of voters [2] without compromising any of the above properties.

The zkVot Protocol addresses these requirements to achieve both aspects of the decentralization: being trustless and censorship resistant. Moreover, by adapting the distributed computation over all participants of the network, zkVot Protocol aims for a scalability and computation power that was impossible before.

This article is written to go through the layers creating the zkVot Protocol and describe each step making this technology possible.

4 Layer Specifications

The zkVot Protocol is designed as a communication protocol between different parties that makes use of zero knowledge proving technology. Currently, there exists 5 different layers of the protocol.

This section aims to explain the protocol in its full detail by going through all layers one by one, including some unique implementation specifications, the current benchmarking and security considerations.

4.1 Voting Layer: Privacy of a Single Vote

The first layer of the zkVot Protocol is the Voting Layer, where all votes are created. The Voting Layer is responsible for providing anonymity while also ensuring scalability.

4.1.1 General Overview

To achieve a fully private design, zkVot uses client side computation: If an information never leaves the client side (e.g. browser), then there is no risk of losing privacy.[8] Thus, each vote is created in the form of a ZKProof (zero knowledge proof) in the voter's personal device.

Currently, ZKProofs are one of the best solutions out there to provide programmable privacy between two parties (named as prover and verifier): ZKProofs allow the prover to convince verifiers to the proper execution of some process[3] without revealing any additional data.

Making use of these properties, the client side (i.e. each single voter) joins the zkVot Protocol as a part of the Voting Layer. This distributed approach does not only guarantee privacy, but also improves the scalability of the system greatly, as the computational complexity of a vote becomes

[2] Here, a meaningful number of voters may be 10^6 or 10^7, as most of the communities in the world may be described as a reasonable partition of these coefficients (e.g. cities of a country).

[3] Here, the proper execution is agreed upon between the prover and the verifier beforehand.

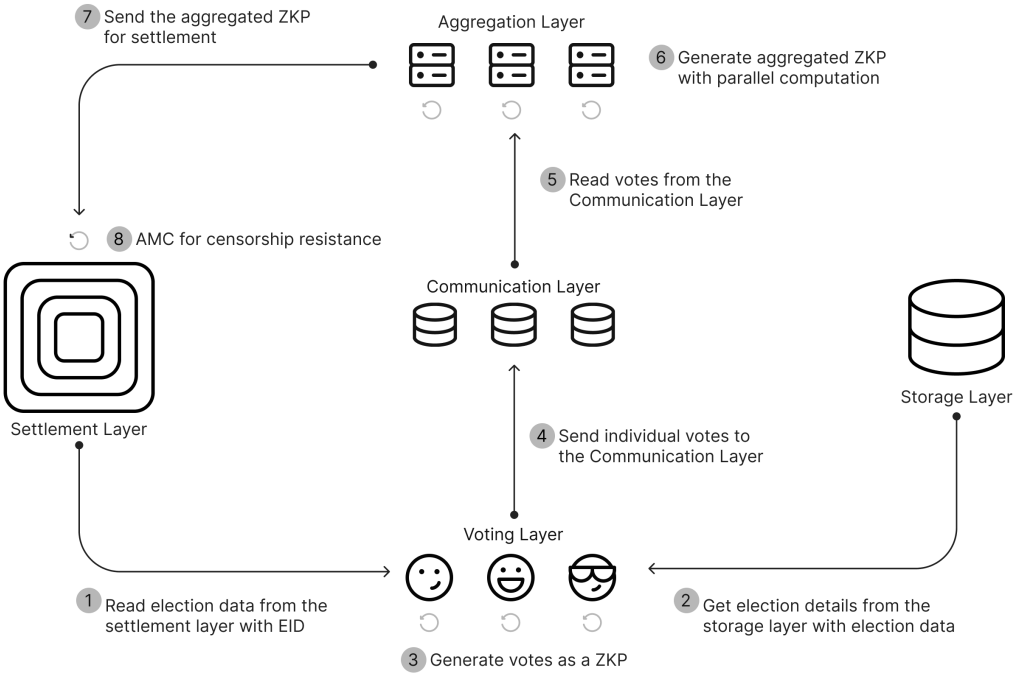

Figure 1: The general overview of the zkVot Protocol's layers and the interaction of a voter during the voting process.

constant over the number of voters: Each voter contributes with their own local computation power to the Voting Layer by generating their own respective ZKProof.

In the zkVot Protocol, ZK-SNARK proofs[10] are used. ZK-SNARK proofs provide all the properties of a ZKProof, and they are also always verified in constant time. This means that the verification time of a ZK-SNARK proof does not depend on the input size or the proof complexity. This improves the scalability even further.

On top of the privacy and scalability advantages, client side generation also allows customization of the authentication method used for different voters of the same election. As the proof is generated on the client side, and the verification of election eligibility[4] will be done inside the proof, any private / public key pair algorithm can be used for voters. Different voters may choose to use different public key algorithms belonging to different blockchains in the same election, making the election chain agnostic in terms of voters.[5]

As explained before, thanks to ZK-SNARK proofs, using different algorithms for authentication does not change the cost of verification in any way, it may just cause the proof generation time of different voters to differ. Nevertheless, remember that since the Voting Layer performs distributed proof generation for voters, this does not affect the scalability of the protocol in any way.

4.1.2 ZKProof Design

The ZKProof generated in the Voting Layer should be designed specifically to achieve all of the properties above, verifying all necessary statements in the most scalable way. Before coming to what statements the ZKProof vote will verify to be true (i.e. the proper execution of the process), let us discuss what output is needed from each ZKProof.

First of all, the ZKProof should output the candidate that the voter has voted for. According to our problem definition, it does not matter if everyone can see the candidate chosen if the voter stays anonymous.

Secondly, ZKProofs need a way of providing a unicity check over each vote to prevent double voting without losing privacy. Thus, a nullifier is added as the second public output of the

[4]Being eligible for an election corresponds to the 1st point in the problem definition.

[5]Here, an interesting improvement in the current design can be using ZK TLS to allow people vote through their emails.

ZKProofs.

Nullifier is an anonymous way of identifying someone. A very basic example of a nullifier is a hash of a private key. If we know the hash algorithm to be correctly executed (which can be achieved through ZKProofs), then we are certain that using the same private key will result in the same nullifier, as hashing is a deterministic process. Moreover, it is impossible to extract the private key from the hash.

However, the hash of the private key results in pseudo-anonymity across different elections: As the same private key joins different elections, the nullifier remains the same, and the partial identity of the voter is revealed. In order to solve this problem, the zkVot Protocol defines a unique election identifier (refer to here after as the EID) for each election, and generates a non-interactive nullifier as the hash of the private key and the EID.

More formally, the ZKProofs must get the following inputs and prove the following statements to be true (which are named as assertions of a proof):

1. The prover must provide a private key as a private input.
2. The verifier must provide the initial set of public keys allowed as the public input.
3. The ZKProof must assert the public key corresponding to the given private key to be inside the predefined set of public voters.
4. The prover must provide the EID as a public input.
5. The ZKProof must generate a nullifier inside the proof from the given private key and the EID to prevent double voting while preserving anonymity.
6. The prover must provide a valid candidate along with a signature to indicate their choice as a public input.

4.1.3 Implementation Details

In the current version of the protocol, o1js[16] is used for the proof generation technology, which is a PLONK based proof system.[11] For the Voting Layer, there are multiple possible proof generation frameworks that can be preferred, thus the reasons behind the choice of o1js are detailed in the respective section below.

In the Voting Layer, one of the most important points to discuss is how to implement the inclusion check of the voter's public key to be inside the initial set. A naive approach is to get the list of public keys as a sorted array, and perform a binary search inside the proof. However, this solution depends linearly on the number of voters for the public input size and logarithmically for the proof generation time. A better implementation is using a merkle tree, where we can get the merkle root as the public input and the merkle witness as the private input of the proof. This does not change the proof generation time significantly, but it makes the public input size constant in complexity. In our ZKProof design, it is crucial to decrease the size of public inputs as much as possible in order to optimize the verification complexity. As we will explain more later[6], decreasing the verification costs increases the censorship resistance of the system significantly.

In order to prevent any issues regarding the merkle tree verification across different provers and verifiers, we can define the protocol to sort leaves of the merkle tree (a.k.a. hash of voter public keys) in an ascending order. This is an accurate example of how the communication rules defined inside zkVot Protocol help the Voting Layer to be distributed across all voters, without needing any bridge or similar.

Finally, the nullifier design specified above is unfortunately not possible this simply. Specifically, private key is a very sensitive data, and almost none of the client side wallets export private key to a browser. Thus, the nullifier cannot be directly created inside the protocol.

In order to solve this problem, the zkVot Protocol offers to use a wallet that supports creating nullifiers through ZKProofs. By using one of these wallets, voters can easily generate their nullifiers without revelaing their private key, and input their generated nullifiers to the ZKProof. Then, the ZKProof verifies the nullifier to be correctly generated[7] for the given EID and public key.

This solution is optimal in terms of performance and decentralization. The only problem is the fact that this set up limits the user to use some specific wallets. Unfortunately, as of today, the

[6] Please refer to the Settlement Layer section.

[7] This is a recursive verification of a SNARK proof inside another, which is described more in the Aggregation Layer section.

majority of most common wallets does not implement ZK functionality to generate nullifiers. This causes zkVot to only support some specific wallets.

A possible solution here (which is not yet implemented in the current design) is to ask users to set passphrase before joining to the election and then to vote using this passphrase (where the ZKProof generate the nullifier from this passphrase with the first process described). However, this forces voters to interact with the protocol in two different times[8], and it is also very hard to design in a scalable way[9]. In short, we believe it may be wiser to wait more wallets to implement ZK nullifier support instead of forcing this UX.

4.1.4 Security and Benchmarking

The security of the Voting Layer is maximal, as there is no private data access or authentication mechanism. The proof set up itself forces the voter to act honestly, and any invalid proof (or double vote) is rejected by other layers of the system without any enforcement in the Voting Layer (detailed more below).

As a result, the only significant property of the Voting Layer to benchmark is the proof generation time. By taking the mean performance of different environments over 1000 votes, we find that the generation time is sufficient. Specifically, each user is responsible for the generation of their own vote, so the proof generation time of a vote has very little effect on the scalability of the system.

Table 1: Vote Proof Time Analysis

Machine	**Avg Time per Vote (ms)**
Apple M2 Silicon 16 GB 8 Core	7796.85
AMD 5600x 16 GB 6 Core	50050.45

4.2 Communication Layer: Gathering of All Votes

After generating private votes on the client side, zkVot needs to define a layer that will be responsible for the transfer of these votes to verifiers. As described in the 8th point of the problem definition, there is a pre allowed time interval to submit a vote, and until this time has passed, ZKProofs should be stored in a layer and communicated to every verifier in the system. This layer is naturally named as the Communication Layer.

4.2.1 General Overview

While designing the Communication Layer, there are few important considerations:

1. Anyone should be able to communicate their vote without the risk of censorship.
2. Anyone should be able to observe the communication without any time delay.
3. There must be a high availability guarantee for communicated votes for a meaningful time.[10]
4. The communication should be fast (i.e. the Communication Layer must support a high number of transactions per second or TPS)
5. The communication costs (i.e. transaction costs) should be as low as possible.

Let us emphasize here that zkVot does not choose the Communication Layer to perform any kind of computation or verification over the sent ZKProofs. To its consideration, a TX (transaction) made to the Communication Layer may be a duplicated vote, an invalid ZKProofs, or even just some random data. Moreover, the Communication Layer does not implement any functionality to stop the communication after the election is done. This functionality is provided in another layer. And all these choices allow zkVot to use a DA (data availability) layer as the Communication Layer.

[8] There must be a reasonable amount of time between two interactions to preserve anonymity.
[9] You can read more about parallel aggregation during the arrival of votes in the Aggregation Layer section.
[10] A meaningful time in the context of an election is usually less than a few days.

4.2.2 Usage of DA Blockchains in the Communication Layer

DA layers are very efficient ways of storing data with very high liveness guarantee and with very low cost, by implementing a structure called light nodes. Light nodes[21] can read from and write to the blockchain like full nodes, meaning without requiring any intermediary but the gossip network. Yet, light nodes are not a part of the consensus, allowing them to be started without a complex synchronization process. They only store a small amount of the data, and thus they are much "lighter" machines than full nodes. A light node can be started in a personal computer in minutes without waiting for the synchronization from genesis.

By implementing light node structures, DA layers allow users to send TXs with a minimal risk of censorship. Even though most DA layers do not usually provide programmability on top, zkVot uses DA for the other advantages it has:

1. DA layers have very high liveness for short time storage.
2. TXs made to DA layers are cost efficient.
3. The block time and the TPS of a DA layer is sufficient for scalability.
4. Communication of votes becomes censorship resistant, as light nodes make the need of an RPC or similar intermediary while writing and reading data.

The final point needs a bit more clarification: Individual voters can of course choose to submit their personal votes through their own light nodes. However, this is not really required, as using an RPC while sending a vote is not a real risk of censorship. Submitting a block to the DA through RPC is a single time trust, and in most networks installing a light node also involves a single time trust. Moreover, the voter may use multiple RPCs; If all RPCs reject a vote, this means that the entire network is censoring the voter, which is against the majority assumptions of a DA. However, the same cannot be said for verifiers. Verifiers cannot trust RPCs to communicate all votes without any significant delay, as this is an ongoing trust during the election. Thus, the light node aspect of a DA is useful for verifiers, preventing any censor while reaching communicated votes.

On a different notice, it is important to emphasize once more that DA layers are not a good choice as a persistent storage. They do not have high availability guarantees for historical data, but this does not really matter for our purposes. Communication Layer is only used for the distribution of votes, and not the storage of any long term data.

Even though DA layers may seem like the perfect Communication Layer, as said before, they do not provide programmability over data. As a result, there is a need for an additional layer responsible for the tally of the election by verifying sent ZKProofs.

Nevertheless, this actually is not a problem that should be fixed about DA layers. In an ideal architecture, these functionalities must indeed be separate. By not handling the verification of sent data, DA layers maximize short term liveness, minimize costs of storing and sharing data, and prevent any censorship while reading blocks.[11]

4.2.3 Implementation Details

Even though the zkVot Protocol does not implement its own DA layer, but rather just uses an existing one, it is an important question to answer which DA layer to use. Fortunately, our answer is really satisfying: It really does not matter.

As we will discover more later, the zkVot protocol depends really little on the choice of the DA layer. Thus, different elections can choose different DAs as their Communication Layer. Moreover, an election can even use multiple DAs at the same time for the Communication Layer.

If the Communication Layer consists of multiple DAs, then verifiers need to listen to all DAs at the same time. However, this is not a problem: It is enough to install a light node to track the DA, and verifiers can have multiple light nodes in the same machine. By using multiple DAs at the same time, elections optimize their performance and decentralization:

1. As voters need to submit to only 1 DA to be included in the protocol, the TPS of the Communication Layer increases significantly with each DA used.
2. Even if one of the DAs becomes unavailable because of overuse or a technical problem, the communication continues over the rest.

[11] Note that this paragraph is a summary of the idea behind zkVot Protocol: Each layer improves some properties by giving up some other, and zkVot brings the best attributes of each to optimize the entire system.

3. The UX improves, as voters may choose the DA they prefer instead of adapting themselves to the DA chosen by the election.

In the current implementation, zkVot uses Celestia[1] and Avail[22], since they meet all the criteria above and they are frequently used in the blockchain space. In the future, more DAs may be added to the zkVot Protocol without needing any protocol level update. Specifically, elections using the zkVot Protocol may customize their choice of DA layer by just releasing a common terminology between all layers included, very similarly to the usage of various ASPs in different applications over HTTP.

4.2.4 Security and Benchmarking

As explained before, zkVot does not implement its own DA layer or creates a bridge between the Communication Layer and other layers. Thus, we can directly adapt the security considerations of the used DA layers[1][22].

Nevertheless, it is worth mentioning that it is the constant time verification property of ZK-SNARKs that allows zkVot to design the Communication Layer in this way. If it was costly to understand if a block is a valid ZKProof or not, then a bad actor may be able to censor the system by sending too many invalid ZKProofs.

By benchmarking the verification process in various machines, we show that this is indeed not the case:

Table 2: Vote Verification Time Analysis

Machine	**Avg Time per Vote (ms)**
Apple M2 Silicone 16 GB 8 Core	628.64
AMD 5600x 16 GB 6 Core	3151.90

As a result, the only way to really censor the sending of valid ZKProofs is to use all of the available network space, equivalent to censoring the entire DA layer, which is prevented by the DA architecture.[12]

4.3 Aggregation Layer: Counting Votes as a ZKProof

After having votes communicated to everyone interested, the zkVot Protocol needs to count all votes without losing privacy or decentralization.

4.3.1 General Overview

Getting the result of the election from ZKProofs is actually quite simple. Anyone can track published ZKProofs from the DA layer, verify each of them in their local machine (also while making sure there is none with a repeated nullifier to prevent double voting), and sum up public outputs of proofs to come up with the result of the election.

This process is totally trustless and scalable, since the verification of a ZKProof is very cheap even in a personal computer (as shown before). However, DA layers are not ideal as a persistent storage, so we cannot trust all ZKProofs to be available in the long run for anyone to verify the result of the election.

Moreover, allowing everyone to see all votes does not solve the problem with the current election systems. The process described above is actually very similar to the tally process in conventional elections. Using DA is a slight improvement to the conventional tally, as only 1 observer (i.e. tallyman) is enough to observe the entire process. However, observers are not accepted by legal authorities, and the final result is always decided by a single governmental authority. Thus, zkVot Protocol needs to implement similar functionality online: All results must be combined for once, while preserving trustlessness, and must be settled in a provable layer for anyone to verify at any point of time. For this chapter, we will talk only about the counting process and come back to the settlement process later.

A possible approach to allow decentralized counting may be using on-chain computation to verify and add up votes. By performing the above process in an on-chain computation environment,

[12] Sending a block to the DA has a cost, thus a full censor of the network would require too much funds[1][22].

zkVot may allow anyone to verify the result of the election by only reading the state of the on-chain smart contract. However, this approach needs every vote to be submitted as a TX to the on-chain computation environment. This is not scalable, both in performance and costs, as it depends linearly to the number of voters. Thus we need to use off-chain computation to scale without losing decentralization.

Off-chain computation is the process of using off-chain resources to perform some computation, and updating the state with the final result. When we say off-chain computation, we may refer to any computation environment but the final blockchain we store the result. However, in the zkVot case, the off-chain layer is deliberately chosen as a centralized server to maximize the scalability of the system.

Nevertheless, in order to preserve decentralization, all computations in this off-chain server must be performed in a trustless and censorship resistant manner. There are various ways of making an off-chain process trustless, and a technique called ZKProof aggregation is preferred in the zkVot Protocol.

4.3.2 Trustless Vote Counting through ZKProof Aggregation

ZKProof aggregation[2] is the process of including ZKProofs inside another in the form of a different ZKProof, also while combining their public outputs. The final aggregated ZKProof verifies all the ZKProofs included during its computation, and additionally may assert any other custom statement. Basically, it executes the same process of a trustless counting in the form of a ZKProof, so that anyone can verify only the final proof to learn the final result. Naturally, zkVot names the layer responsible for ZKProof aggregation as the Aggregation Layer, and the off-chain server that is responsible for the proof generation as the aggregator.

Once the aggregation is done, the Aggregation Layer submits the final proof to the next layer to verify and settle the election result in a decentralized manner. In the zkVot Protocol, this layer is named as the Settlement Layer.

Along with providing trustless off-chain computation, there are several other advantages of using ZKProof aggregation in the counting process of the zkVot Protocol

1. It is enough to submit 1 TX to verify and store the final result of an aggregation.
2. Aggregation of ZKProofs is constant in verification time[13], meaning the final aggregated ZKProof is always verified in constant time complexity regardless of the proof count in the aggregation. Thus, on-chain verification costs are the same regardless of the election size.
3. Aggregation of ZKProofs is fixed in proof size[14], meaning after the first aggregation, the proof size always stays constant. Thus, the final proof size is the same regardless of the number of voters in the election.

As a result, it is enough to scale the off-chain aggregator to scale the election: Scalability of the system does not depend on the performance of the Settlement Layer. And since off-chain computation is much cheaper than a decentralized computation environment, the Aggregation Layer provides cost efficiency for zkVot.

However, it is not easy to design the aggregation ZKProof. There are a few important problems to solve to not lose the scalability.

4.3.3 Aggregation ZKProof Design

First of all, during the counting of votes, the aggregation ZKP should include a unicity check to make sure none of the votes are double counted by the aggregator. Each step of the aggregation must also output some structure showing which nullifiers are used during previous iterations, so that the next step of the aggregation may check the exclusion of the new nullifier inside this structure. We can use another merkle tree to express all used nullifiers in an efficient and verifiable way, but it is really hard to perform an exclusion check over a merkle tree: We need to compare all leaves of the merkle tree with the new nullifier to make sure none of them is the same. A much more optimized and scalable way to perform a unicity check is to use a merkle map.

Merkle map (a.k.a. sparse merkle tree or indexed merkle tree) is a giant merkle tree that has as many leaves as the prime order of the hash function used. For instance, if the hash algorithm

[13] As stated before, ZK-SNARKs are verified always in constant time, and the property is preserved during aggregation.

[14] Note that this property is true for the proof system used, but not necessarily for all proving technologies.

that is used has the prime order P (i.e. outputs of the hash are all in the range $[0, P-1]$), then the merkle map also stores P leaves. Initially, every leaf of the merkle map is set to 0, representing that nothing is included in the tree. Every new insertion to the merkle map turns the corresponding leaf from 0 to 1, instead of adding a new leaf as in the case of the merkle tree. A leaf being 1 corresponds to having the hash in the merkle map, while 0 indicates the absence of this hash.

The merkle map's architecture may sound very unoptimized, as the hash domain is usually huge and we need a very big merkle map to store all possible outputs of a hash[15]. However, as the verification of a point in the tree is logarithmic in complexity, merkle maps are efficient enough to be used inside ZKPs. And as merkle maps can also generate a proof showing the absence of an hash value in the tree, they allow efficient exclusion checks.

To summarize, the aggregation ZKProof gets the following inputs and assert the following statements to be true:

1. The proof gets the previous aggregated proof. For the base case, this proof is empty.
2. The proof gets the vote proof to be counted in this step (this vote proof has the specifications described in the Voting Layer chapter)
3. The proof verifies the vote proof to be correctly generated.
4. The proof gets the merkle map witness of the vote proof's nullifier.
5. The proof asserts that the merkle map witness of the vote proof's nullifier points to an empty leaf (value of 0) in the merkle map root of the previous aggregated proof.
6. The proof computes the new merkle map root from the merkle map witness with a filled leaf (value of 1).
7. The proof takes the previous number of votes over candidates from the previous aggregated proof and adds 1 vote to the candidate given in vote proof's public output.
8. The proof takes the previous amount of aggregated votes from the previous aggregated proof and adds 1 to compute the new aggregated vote count.
9. The proof outputs:
   (a) The new candidate list representing number of votes over each candidate.
   (b) The new merkle map root including the new nullifier.
   (c) The new aggregated vote count.

With this above design, it is impossible for the aggregator to lie about the result of a counting process. However, off-chain aggregation does not guarantee censorship resistance. Notably, the aggregator may choose not to include a ZKP from the Communication Layer in the final result without breaking the integrity of the aggregated ZKProof. In short, trustlessness does not equal decentralization, and the Aggregation Layer needs a way to provide censorship resistance.

4.3.4 Censorship Resistant Aggregation

There is a straightforward approach to the censorship problem, which is to connect the Communication Layer and the Settlement Layer[16]. By doing so, the Settlement Layer may verify the aggregated ZKProof to include as many valid ZKProofs as on the DA layer. However, this approach is problematic in various ways.

The first possible usage of a connection between the Settlement and the Communication Layer is the comparison of available votes versus the aggregated result. By asserting the two numbers to be the same, the Settlement Layer may prevent the Aggregation Layer from censoring any vote. However, deducing the number of available votes in the Communication Layer does not make sense. If the Settlement Layer verifies all blocks in the Communication Layer to deduce the number of currently available votes, then it may as well count the votes to finalize the election, which is equivalent to the on-chain computation approach described above.

Another way is to force the Aggregation Layer to send proofs not only for valid ZKProofs, but also for invalid blocks proving that they cannot be included in the final result. Then, the Settlement Layer may assert the number of all proofs in the aggregation to be the same as all blocks

[15] For SHA256 hashing, for instance, the merkle map needs to have approximately $2^{256} \approx 10^{77}$ leaves.

[16] For instance, through signature verification of the Communication Layer consensus in the Settlement Layer.

in the Communication Layer. But unfortunately, this is not really feasible in practice. Proving something to not be a valid ZKProof is very difficult to optimize and implement. Moreover, this setup forces aggregation nodes to aggregate much more information then before, creating a possible attack vector on the system as the Communication Layer is permissionless. Note here that the permissionless aspect of the Communication Layer was not a problem before, as the verification time of a ZKProof was constant. However, this is not the case for proving something is not a valid vote, thus invalid blocks put a huge burden over the Aggregation Layer.

As a result of these limitations, in order to solve the censorship issue issue, zkVot Protocol defines a solution that does not connect the Communication and Settlement Layer: Instead of making the aggregator permissioned, zkVot Protocol allows anyone to join the Aggregation Layer as an aggregator. Remember that we previously defined the aggregator as a single centralized server that was permissioned by the Settlement Layer. Now we update this set up to allow anyone to aggregate ZKProofs and settle results: Communication Layer is already permissionless, and anyone may join the DA by installing a light node to track all votes. This means that anyone can actually run some zkVot aggregation software[17] to reach the election result trustlessly.

A permissionless Aggregation Layer prevents censorship since it needs only 1 honest aggregator to submit a settlement TX. If **only 1 aggregator** is acting honest and settling all TXs, then the real results get reflected on the chain. Needing only 1 honest aggregator is a very light trust assumption, especially considering most consensus algorithms need at least the 51% of the system to be honest.

In order to incentivize the honest aggregator, we need an incentivization mechanism in the protocol. In zkVot, the incentivization is defined in the Settlement Layer. The first honest aggregator receives the rewards available in the Settlement Layer at the end of the election. This setup is secure and scales with the number of voters in the election: A bigger election means a bigger incentive to be censorship resistant, and thus the election provider has more funds to put on the Settlement Layer to motivate the honest aggregator.

However, there is a problem with making the Aggregation Layer permissionless: The verification process needs a way to understand which aggregator is honest and which ones are censoring some votes. zkVot solves this problem by putting a condition over the settlement, named as the Accepting the Maximum Condition.

4.3.5 Accepting the Maximum Condition

Accepting the Maximum Condition (referred to as the AMC hereafter) stores the last maximum number of settled votes in the Settlement Layer. When an aggregator sends a settlement TX, it is accepted only if it is bigger than the last stored number in the stored state. As a result, if there is only 1 honest aggregator in the Aggregation Layer, all voters are always included in the final result. All dishonest aggregators trying to censor some votes are blocked by the AMC in the verification contract.

Moreover, even in a set up where none of the aggregators is honest, but they are in a perfect competition to get the incentive on the Settlement Layer, the AMC may ensure that censorship resistance is achieved: Instead of rewarding all aggregators that was able to settle (i.e. all aggregators who submit an aggregation proof with a count of votes greater than the previous recorded maximum), the settlement contract waits until the election is finished to reward the honest aggregator. Thus, the first aggregator who reaches the maximum aggregation count (i.e. the real result of the election) is rewarded at the end of the settlement period. This causes dishonest aggregators to compete to reach the maximum result quickest.

To describe more formally, we may consider the settlement TXs as a sequence of integers, where each integer represents number of votes included in an aggregation proof. Then, we see that AMC forces this sequence to be strictly increasing. As the sequence is naturally bounded by the number of voters in the system[18], the model becomes a strictly increasing bounded sequence, so it is forced to converge.

Nevertheless, in order to increase the liveness of the AMC to its fullest, we need to decrease the aggregation complexity as much as possible. A more optimized aggregation process makes the job of the honest aggregator much easier, and thus the incentive becomes much higher. It may be preferable to use a proof system optimized to perform very efficient aggregation, but this does not change the aggregation complexity in terms of number of voters. Formally, designing the

[17] Aggregators can also choose to implement their own aggregation logic as long as it produces a valid ZKProof with the same verification key in the Settlement Layer.

[18] A valid aggregation ZKProof may include maximum N proofs in an election with N voters.

aggregation as a single threaded process where each aggregator may count a single vote at a time limits the scalability.

A single threaded linear approach does not scale in the long run because of hardware limitations. A bigger incentive coming with a bigger election may motivate the honest aggregator to support running better hardware. Yet, hardware may scale up to a point, and even a very performant hardware would not be enough for aggregation of millions of votes in a reasonable time. In order to improve this limitation, the Aggregation Layer must allow multi-threading through parallel aggregation with ZK-SNARKs.

4.3.6 The Parallel Aggregation Tree

Parallel aggregation is the process of aggregating different ZKPs without waiting for others, and then combining all these proofs together again in the form of a binary tree, referred to here as the aggregation tree. This allows the honest aggregator to work on the aggregation of multiple proofs at the same time or in different machines. This improves the complexity of the aggregation from linear to logarithmic dependence.

Unfortunately, merkle maps cannot be used efficiently with parallel aggregation, as each update to the merkle maps affects the root, and it is impossible to compare different roots to understand if they include the same updates or not. In order to solve this limitation, the zkVot protocol performs parallel aggregation through a segment tree like approach.

In this approach, proofs in the aggregation tree do not output a merkle map root. Instead, they output the smallest interval that the nullifiers they have used are contained in, named as the *smallest unicity interval*. To clarify, as stated before, nullifiers are a basic hash, and thus limited to be contained in a finite domain: Each nullifier is an integer in the range $[0, P-1]$, where P is the prime order of the hash function used. Thus, for every subset of nullifiers of an election, we can define the smallest unicity interval. Specifically:

$$I = [\min_{\forall j \in J} n_j, \max_{\forall j \in J} n_j]$$

where I is the smallest unicity interval of a proof and $n_j, j \in J$ is the set of nullifiers that creates the interval. For the leaves, we define the smallest unicity interval as the nullifier itself:

$$I = [n_j, n_j], \forall j \in J$$

Then, while computing the parent of two proofs in the aggregation tree, in order to make sure none of the nullifiers are duplicated between two children, we simply assert:

$$\max_{\forall i \in I_1} i < \min_{\forall i \in I_2} i$$

where I_1 and I_2 are the smallest unicity intervals of the left and right child, respectively.

Finally, we compute the smallest unicity interval of the parent, I, as follows:

$$I = [\min_{\forall i \in I_1} n_i, \max_{\forall i \in I_2} n_i]$$

As each time children are checked for exclusion, there is no risk of double voting with this setup. Moreover, this system is much more optimized than a merkle map implementation.

The only problem with this structure is the fact that, in this model we need to sort all nullifiers before starting aggregation. This is not a problem if all votes are already put on the Communication Layer before the counting process. However, this is usually not the case, and in order to allow aggregation during the arrival of votes, we should be able to add any nullifier to this tree in logarithmic time complexity. This can be achieved by storing all intermediary proofs in the cache of the aggregator and using them to add a new leaf to the tree. In the worst-case scenario, this compromises the balance of the binary tree. When multiple updates target the same range, the property of constant updates within that range is disrupted.

To solve this issue, the zkVot Protocol defines a new tree structure for **parallel SNARK aggregation with unicity check**.

While designing this tree, we first define K, number of parallel threads we want to run for the aggregation process. We can easily see that K is well defined for all elections as the number of voters in an election is known at start. Then, we divide the total range of nullifiers, $[0, P-1]$, into K equal intervals, so that the k^{th} interval is defined by:

$$I_k = [(k-1)\frac{P}{K}, k\frac{P}{K} - 1], \forall k \in [\![1, K]\!]$$

Here, we define two different types of aggregation proofs, one representing each I_k (named as the interval proof) and one responsible for the aggregation of these intervals (named as the aggregation proof). For now, we assume that the interval proofs assert all nullifiers to be in the interval it represents and also allow logarithmic additions of nullifiers to the interval. With this assumption, the aggregation proof is created with the same process as before, but since the interval proofs are always updated in the same time, the risk of becoming a chain becomes avoided.

Lastly, in order to allow the interval proofs to have logarithmic inclusion of a new nullifier, we use the same merkle map structure as before. Here, we observe that each interval proof is already assigned to a thread (since we defined K to be number of threads). Thus, the responsible thread of an interval does not need to perform parallel aggregation for the interval proof generation anymore, so the interval proof can be created in the single threaded way of a merkle map. Being single threaded, using merkle map does not create any risk of increasing the update complexity.

As a result, the time complexity $T(U)$ of an update U becomes:

$$T(U) = T(I_k) + T(A)$$

where $T(I_k)$ is the time complexity of the update of an interval proof and $T(A)$ is the time complexity of the new aggregation proof. As already explained, $T(A) = O(log(K))$, since there are K intervals and we store all intermediary proofs.

Finally, as the merkle maps included in each interval proof has a range of P/K, $T(U) = O(log(\frac{P}{K}))$. As a result:

$$T(U) = T(I_k) + T(A) = O(log(\frac{P}{K})) + O(log(K)) = O(log(P))$$

which is equivalent to the time complexity of using a merkle map proof in a single threaded aggregation.

Notably, the time complexity of a single update does not depend on the choice of K. Nevertheless, the choice of K affects the total proof generation time as we can work on K proofs at the same time in total (here the aggregation proof generation can be optimized even more for simultaneous updates of different interval proofs). Specifically, assuming the aggregation of an interval proof takes S seconds, the total time needed for the aggregation to be completed becomes $O(S\frac{N}{K})$ for N voters.

On the other hand, a bigger K means that we need to have more threads, which increases costs of aggregation. In this article, an ideal choice of K is not discussed, but a trivial nearly optimal choice of K is $K = \sqrt{N}$ for N voters. With this consideration, the total time complexity of aggregation equals $O(S\sqrt{N})$.

Finally, it is important to observe that if all nullifiers are in the same interval I_k, then this method becomes equivalent to a single threaded aggregation. However, nullifiers are deterministic and resistant to a preimage attack, so a possible attacker cannot attack the system by altering some nullifiers to always be in the same range. Thus, it is a safe assumption that randomly created nullifiers are always equally spread among K parallel threads.

4.3.7 Implementation Details

The most important detail to clarify about the Aggregation Layer is which proof generation framework to use. The zkVot Protocol choses to use o1js, a ZKP generation technology with various advantages:

1. o1js is a ZK-SNARK generation framework, and it provides full privacy through zero knowledge property.

2. o1js is an NPM library in TypeScript that can natively be run in Wasm and similar NPM libraries. This increases adaptability of the protocol to various JavaScript frontend frameworks: As all clients are a part of the Voting Layer, being able to run o1js in different frontend frameworks is important.

3. As it is in Typescript, o1js is designed to improve the proof generation performance in limited environments like browser. Again, this improves the Voting Layer performance significantly.

4. o1js includes a lot of pre-built features and libraries, decreasing the implementation time and costs of creating the protocol. For instance, the Merkle Tree and Merkle Map implementations are already available in o1js.

5. And finally, o1js includes a layer called Pickles to improve ZK-SNARK aggregation by recursion. As described above, the aggregation performance is important to maximize the censorship resistance of the AMC.

Nevertheless, o1js is not the only proof generation framework with this capabilities, and the most important reason to choose o1js is given in the Settlement Layer chapter.

4.3.8 Security and Benchmarking

Similar to the Voting Layer, the permissionless aspect of the Aggregation Layer allows it to be in the maximal security possible: These is no authentication or similar to become an aggregator. If a dishonest aggregators tries to alter the proof aggregation process, then the result proof is simply rejected by the Settlement Layer, which is secured by the security considerations of the Settlement Layer.

In terms of scalability, the free choice of K allows the system to scale with the number of voters. As there are more voters, there is a bigger incentive to run more threads (or collaborators) to reach to maximum voter count the earliest possible. As a result, the only factor affecting the scalability of the system is the aggregation time of two interval proofs. Below, you may find the average time of the aggregation of 1000 interval proofs:

Table 3: Interval Proof Aggregation Time Analysis

Machine	**Avg Time (ms)**
Apple M2 Silicone 16 GB 8 Core	18444.15
AMD 5600x 16 GB 6 Core	98894.93

For accepting the average time of an interval proof generation around $18s$, in an election of 1 million voters, by having $\sqrt{10^6} = 1000$ threads, we can count 1 million votes in around $18000s = 5$ hours. Please note that $18s$ is a very unoptimal estimate, and it is used mainly to give a lower bound to the scalability of the system.

4.4 Settlement Layer: Optimizing Verifiability and Performance

After the aggregation is complete, zkVot Protocol needs to settle the aggregated ZKP and respective results into a verifiable layer, named The Settlement Layer.

4.4.1 General Overview

Maybe after defining all other layers as above, it may seem like there is not too much that is needed from the Settlement Layer. However, this is not the case, and the zkVot protocol requires the Settlement Layer to have the following properties:

1. fast and cheap on-chain verification,
2. efficient and private off-chain computation,
3. verifiable, censorship resistant, and cheap on-chain data storage.

This 3 points may seem like a lot to ask from a single layer, but they can actually be provided all together as zkVot does not require some very basic properties of a blockchain from the Settlement Layer, which are:

1. custom on-chain computation,
2. low block time and high TPS performance,
3. unlimited on-chain storage.

The first point, not having any custom on-chain computation, is really important for scalability. zkVot adapts the philosophy of Bitcoin[18], which is to position a blockchain as a global verification machine rather than a complete execution environment. This ultimately allows the system to scale in proportion to the number of voters, as each voter joins the system with their computation power.

The second point is more an observation than an optimization. After creating the Aggregation Layer to aggregate all results to one final TX, the need for a fast consensus disappears. zkVot can

of course work with much lower block times, but there is no real connection between the TPS and the decentralization or performance of the protocol.

Finally, being able to work with a very limited on-chain storage allows the zkVot protocol to ensure required properties in the Settlement Layer. It is basically a trade off: It is of course much preferable to store everything in the Settlement Layer, as this is much easier to implement. Yet, on-chain storage usually comes with a cost, and this consideration frees zkVot design from the limitations of having unlimited storage capacity.

4.4.2 Implementation Details

After making these observations, there is one L1[19] that particularly comes forward for our needs: Mina Protocol. Mina Protocol[4] is a ZKProof verification layer with some very unique properties. In zkVot, the following points highlight the most from Mina's architecture:

1. Aggregated Block History for Constant Time and Size Verification

   Mina is basically a huge aggregated ZKProof. The entire block history is represented as a constant sized aggregated ZKProof: The entire Mina state history is 22 KB in size. This allows anyone to verify Mina state in constant time and memory. In an ideal voting protocol, we want the final results to be verifiable by anyone with minimal trust, and this set up allows us to achieve this.

2. Low Network Fees

   All TXs are described as a ZKProof, and Mina validators are only responsible for the verification of these proofs. And since the verification time of a ZK-SNARK is independent from the proof size, TX fees in Mina do not depend on the computational complexity. This ensures deterministic and low network fees[20]. As a result, the settlement costs of the zkVot protocol stays constant over the number of voters.

3. Full Node Accessibility

   Thanks to the constant sized block history and the usage of on-chain verification with constant time complexity, Mina requires very low hardware to join the gossip network as a full node. Mina does not implement light nodes yet, but browser compatibility[21] makes Mina full nodes to be almost as accessible as light nodes. Using a light node for settling is not important[22], but this increases long term censorship resistance of reading the election results.

By combining all these properties, Mina becomes a very strong choice as a Settlement Layer for the zkVot Protocol. However, providing these properties without losing decentralization comes with some cost.

First of all, Mina only support its own native proof generation framework, o1js (previously Snarky), to be able to verify proofs on the blockchain. This enforces the Aggregation Layer to use the o1js as the proof generation framework. Even though o1js is a performant technology, this situation causes the flexibility around the protocol design to disappear for the proof generation framework.

Moreover, Mina has very low TPS and high block time. Mina's current block time is ∼3 minutes, which is higher than ecosystem standards. Moreover, Mina's consensus is probabilistic and allows forks like Bitcoin, so there is a hard finality duration for blocks (∼2 hours). Nevertheless, as explained before, the zkVot protocol does not actually require fast finality, as there is a single TX accepted for every election at the end[23].

Finally, Mina has the most limited on-chain storage available, probably after Bitcoin. A single Mina smart contract can store up to 8 Field elements (32 bytes). Thus, the zkVot protocol requires some clever approaches to the on-chain storage usage in order to use Mina as the Settlement Layer.

[19] L1 (a.k.a. layer 1) refers to a base blockchain like Bitcoin or Ethereum, that can particularly work without depending on any other blockchain.

[20] The average TX fee in Mina is currently about 0.02 MINA.

[21] There is a browser full node implementation for Mina.

[22] This is an equivalent trust observation that is made in the Communication Layer about using RPCs while submitting votes.

[23] Please note that this situation is unique to applications like voting, and cannot be said in general. Yet, Mina's block time is expected to improve drastically in the near future.

4.4.3 Minimizing the Amount of Data Stored in the Settlement Layer

Before starting this section, it is important to mention that even though we are going to talk about our implementation specific to the Mina Protocol, approaches that zkVot adopts here may actually be beneficial to other blockchains as well, as the storage is usually costly in settlement layers.

Luckily, all along the article we have actually being prepared to this point, and most of the problems are already solved with the considerations zkVot makes on all other layers:

1. The Voting Layer uses a merkle tree to verify the inclusion of voters inside the initial set, which is constant in size as a public input. It is enough to store the merkle root on-chain to perform the inclusion verification.
2. The Communication Layer frees the settlement layer from knowing anything about individual votes.
3. The AMC during the settlement requires storing only one state to store the number of votes in the last accepted TX, and no other information is needed from the Aggregation Layer.

As a result, only the following information should be stored about every election in an available way:

1. The voter's merkle tree root,
2. The Communication Layer identifier, to notify aggregators where to track in the DA,
3. Election question and details.
4. Election's start & end data,
5. The valid choices of the election and what each choice represents as a candidate,
6. The list of voters in the election, to allow anyone to generate the merkle tree,
7. The election's result,
8. The last AMC number (i.e. number of votes in the last accepted settlement TX).

Clearly, all this information is independent for each election. Thus, it makes more sense to design each election to be a separate *smart contract* in Mina. Instead of putting every election in the same smart contract, zkVot Protocol deploys a separate smart contract for each election. This increases memory limitations significantly.

Nevertheless, it is impossible to store all of this information in any way in Mina. A single Mina smart contract can store about 32 bytes of data, thus we should only keep what is absolutely necessary in the Mina smart contract:

1. The EID,
2. The voter's merkle tree root,
3. Election's start & end data,
4. The last AMC number,
5. Election result.[24]

All of this information is crucial to have directly on Mina: The verification of the settlement TX requires the EID, the merkle root, election start & end date, and the last AMC number as a public input. However, there is a clever design to perform trustless verification without storing not all of these in the state: We can have the EID, the merkle tree root and election start & end date as a constant in the aggregated ZKP. When there is a constant in a proof, it changes the verification key, and thus causes the verification to fail on Mina.

Nevertheless, there is a lot of additional data we need to store about the election as stated above, like the election question and details. Thus, we introduce the next layer of zkVot: The Storage Layer.

[24] As we will explain more in detail in the Storage Layer section, the data that we are looking to export from Mina should be static to minimize trust assumptions and maximize the scalability. Thus, we choose to store election results on Mina as well.

4.4.4 Security and Benchmarking

As we have already discussed the security of the settlement process in the Aggregation Layer's security section, the only thing remaining here is to evaluate Mina's economical security as a settlement layer. Mainly, our security and availability guarantees over the settlement layer are much higher, since we expect the election to be verifiable for all times.

In fact, Mina does not provide an economical security[9] in the same level with Ethereum or Bitcoin. Nevertheless, it is important to observe here that the lower economical security of the Mina blockchain does not create any problem for the zkVot Protocol, as each vote in Mina is essentially an aggregated ZKProof along with some identifier information (which may all be designed as a part of the proof). Specifically, by providing the settlement aggregation proof and the associated election information, anyone can prove the election result at any time. If chosen, this information can be written to a permanent storage blockchain to maximize the availability in the future.

As a result, we believe that the Settlement Layer's economical security does not put any limitation over the zkVot Protocol. If the election result is considered too important to not be trusted by the chosen settlement blockchain, then all parties involved in the election process have the mean to store the election result as a proof for any future verification.

In terms of the benchmarking of the settlement process, as there is a need for **only 1 settlement TX per election**, the settlement layer's TPS or scalability becomes again irrelevant to the zkVot Protocol.

4.5 Storage Layer: Verifiable Elections for All Times

Storage Layer is the final layer of the zkVot Protocol, which is responsible for the storage of all the election related data. Before passing to our considerations for the Storage Layer, let us discuss how we can connect it to the Settlement Layer.[25]

4.5.1 Connecting the Storage and the Settlement Layer

To start, there is one important observation to make: All of the data in the Storage Layer is static, i.e. it does not change after the start of the election. This is deliberately chosen to be like this, as it allows us to connect Settlement and Storage Layers without a bridge.

Instead of creating a bridge between two layers, the zkVot Protocol only stores the identifying information of the Storage Layer in the settlement contract. For instance, if the information can be reached with an ID, then this ID is stored in the Mina smart contract. Note that we do not store the hash or some verification data, but we directly store the encoded ID on the smart contract. The Storage Layer has no information related to the Settlement Layer or the rest of the election.

This is possible because of the static nature of the data: If the election data is not properly uploaded to the Storage Layer, or an untrue ID is put on the Settlement Layer, then it is impossible for anyone to join the election. This is the important property that the system must hold: It is always possible to create a meaningless election, but no one should be able to censor or change a proper election once it is created.

Then, why do we need to store the Storage Layer identifier in the Settlement Layer? It may seem like a reasonable assumption that the election provider will be providing the identifier to all voters to allow them vote in the election. However, this actually creates a trust assumption.

The real power of the zkVot comes from the fact that anyone can verify the entire process by only looking at the Settlement Layer. Mina is actually very like a DA; even though it has very limited data storage capacity, it actually provides all DA guarantees on the limited amount that it stores. Moreover, it actually verifies the entire blockchain history in each step by using ZKProof aggregation. Thus, storing the Storage Layer identifier on Mina actually makes sure that anyone who knows about the election will be able to verify the result and participate if they are an eligible voter.

This of course requires the Storage Layer to be static and censorship resistant as well, meaning:

1. Anyone should be able to upload data in the Storage Layer.
2. Anyone should be able to read data from the Storage Layer.
3. It should be impossible to change or delete data from the Storage Layer.

[25] The reason that we present this section in this way is the fact that this part is the most important idea that zkVot adapts for the Storage Layer: The light connection of this layer to the rest of the protocol is one of the core components of zkVot's scalability

The final point is especially crucial, as it is the criteria that makes sure that the election stays verifiable in the future. To put it in simpler terms, if someone creates a valid election by uploading all the required data to the Storage Layer, then anyone will always be able to verify and join in the election by only knowing the Mina smart contract address.

Moreover, in order to make sure that there was no error during the election creation, zkVot Protocol also stores a hash of the entire election data in the Mina smart contract state. This makes the verification of valid elections very simple. In order to verify an election, it is enough to:

1. Go to the Mina smart contract with the given smart contract address.
2. Read the storage ID from the Mina state.
3. Go to the Storage Layer and retrieve the data with the specific ID.
4. Check its hash against the hash stored in the Mina smart contract.

As a result the entire verification of the election becomes possible only through a single layer, Mina.

Here, it is important to emphasize that an election does not become valid just because it follows the zkVot Protocol definition. The election creator can ask questions in a biased way, or select the initial list of eligible voters dishonestly. This is not something that the protocol can be secured against, but it is prohibited by the social consensus.

Social consensus is the underlying consensus of most blockchains, giving them long term decentralization guarantee. It can simply be described as the transparency of the system. Any technology makes the assumption that if a process is fully transparent, then the community (i.e. social consensus) will realize any potential fraud and act against it.[7]

In conventional blockchains, social consensus usually secures the chain through a hard fork. For instance, the Ethereum chain has been forked several times in order to prevent some major attacks or to have core level updates. In the zkVot's case, a hard fork is just someone else going there and creating a valid election: As the election creator has no privilege on the election[26], anyone can create a valid election as long as the majority of the community accepts it.

One final important advantage of designing the Storage Layer independent from the Settlement Layer is the flexibility that it provides around choosing different blockchains as the storage. Unlike the Communication Layer, there is no point of having multiple blockchains in the Storage Layer, but it is a nice UX improvement to allow the election creator to choose the Storage Layer based on the community habits.

4.5.2 Implementation Details and Security Considerations

After creating this virtual connection between the Storage Layer and the rest of the protocol, there is not much left to say. As stated above, there is no limitation on the choice of blockchain for the Storage Layer. As a result, the security of the data stored, the scalability of the data write process and the cost of data storage depend on the blockchain chosen.

Currently, zkVot Protocol provides the Arweave and Filecoin blockchains as Storage Layer options, mainly because of their advantages over the storage of long term data with low costs and high availability guarantees. However, it is possible to use any other blockchain without even changing any line of code.

4.6 Some Additional Notes on Full Anonymity

Finally, we have defined the entire election process while respecting all the points we have described in the problem definition. However, there is one final point that we have skipped while describing the protocol, which is the fact that anonymity may be breached during the sending of votes to the Communication Layer. Even though the identifier of the voter is kept anonymous by using ZK, the wallet signing the TX and paying the DA costs is revealed during the submission of the ZKProof. Thus, we need to anonymize TXs coming to the DA to provide full anonymity during the election process.

To simplify the problem, we can anonymize fund transfer happening to a custom DA wallet instead of anonymizing the TX sending. Sending anonymous TXs to the DA is equivalent to creating a single time usage wallet and transferring some anonymous funds there to pay the TX costs. Moreover, it is much simpler to anonymize funds than to anonymize custom data.

[26] As the Aggregation Layer is permissionless, it does not matter who has created the election as long as it is accepted by the community.

There are already a few working implementations of TX anonymization networks over blockchains, and by using bridges the transfer can easily be reflected on the DA layer. For now, the zkVot Protocol does not implement TX anonymity over decentralized networks for the client side application, and users should go and handle the fee anonymization outside of the network to maximize decentralization.

Nevertheless, zkVot has a working centralized server that accepts votes through a public API port and pays for the funds on behalf of users. This may seem like causing a huge censorship problem, but in real elections it is reasonable to assume that each candidate joining the election can easily provide this service to their voters. As this layer is only responsible for anonymizing the DA TXs, voters can choose to use the service of the candidate they support to not face censorship.

Here, it is important to notice that the Internet itself can provide a lot of anonymity over transfer of data. Even though VPNs are not perfect sources of anonymity, IP addresses reveal only some partial information that can easily be hidden. The real problem of an online anonymous election with the current internet is the computation, i.e. counting of votes. By transferring this issue to the Communication and Aggregation Layer, zkVot can use the private communication that the internet ecosystem provides.

A more interesting idea can be using FHE (fully homomorphic encryption) to not reveal candidates from votes.[12] Specifically, every ZKP would be outputting the encrypted candidate, and the counting would be done over the encrypted results using the homomorphic property. Then, the result will be revealed only when the election is complete and a chosen majority of the voters agreed to reveal the final result.

However, some information may be revealed with an FHE solution, especially when the number of voters is very low. Moreover, this is actually a different kind of election, where the result is hidden until the election is complete. zkVot Protocol adapts a public election model where votes are counted instantly as they arrive to the system. Thus, it is more appropriate to consider FHE as a possible (and very useful) improvement over zkVot than an anonymity solution.

5 Conclusion

Finally, we conclude. In this detailed article, we have tried to give a somewhat accurate idea of how a distributed and anonymous internet would look like over an example use case, voting, which we consider very relevant to a lot of real life use cases.

We realize that this article is long. This is partially because it describes one of the first distributed and private computation protocols in the world with a real life use case and implementation details; partially because we have deliberately chosen it to be so. Sometimes some solutions seem trivial to writers and researchers, but it may be hard for others to deduce them from a few lines of explanation. And it is easy to skip a part if you find it useless, but hard to rewrite if you want to have it. Thus, we chose to explain everything to its final detail: to raise all the questions we had while designing the system, and to answer them all. We thank you to all readers for their time.

The internet was a breakthrough. It changed the way people perceive the world around them. Yet, there is still a long way to go, and zkVot is just the first step that we take into the world of limitless computation and full privacy.

References

[1] Mustafa Al-Bassam. "Lazyledger: A distributed data availability ledger with client-side smart contracts". In: *arXiv preprint arXiv:1905.09274* (2019).

[2] Nir Bitansky et al. "Recursive composition and bootstrapping for SNARKS and proof-carrying data". In: *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*. 2013, pp. 111–120.

[3] Joseph Bonneau et al. "Coda: Decentralized cryptocurrency at scale". In: *Cryptology ePrint Archive* (2020).

[4] Joseph Bonneau et al. "Mina: Decentralized cryptocurrency at scale". In: *New York Univ. O (1) Labs, New York, NY, USA, Whitepaper* (2020), pp. 1–47.

[5] Sean Bowe, Ariel Gabizon, and Ian Miers. "Scalable multi-party computation for zk-SNARK parameters in the random beacon model". In: *Cryptology ePrint Archive* (2017).

[6] Vitalik Buterin. *Ethereum Whitepaper*. Accessed: 2025-01-14. 2014. URL: https://ethereum.org/en/whitepaper/.

[7] Vitalik Buterin. *Hard Fork Completed*. Accessed: 2025-01-14. July 2016. URL: https://blog.ethereum.org/2016/07/20/hard-fork-completed.

[8] Vitalik Buterin. "Some ways to use ZK-SNARKs for privacy". In: *Vitalik's Blog* (June 2022). URL: https://vitalik.eth.limo/general/2022/06/15/using_snarks.html.

[9] Brad Cohn, Evan Shapiro, and Emre Tekişalp. *Mina: Economics and Monetary Policy*. 2020.

[10] Alfredo De Santis, Silvio Micali, and Giuseppe Persiano. "Non-interactive zero-knowledge proof systems". In: *Advances in Cryptology—CRYPTO'87: Proceedings 7*. Springer. 1988, pp. 52–72.

[11] Ariel Gabizon, Zachary J Williamson, and Oana Ciobotaru. "Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge". In: *Cryptology ePrint Archive* (2019).

[12] Craig Gentry. "Fully homomorphic encryption using ideal lattices". In: *Proceedings of the forty-first annual ACM symposium on Theory of computing*. 2009, pp. 169–178.

[13] Jens Groth. "Short pairing-based non-interactive zero-knowledge arguments". In: *Advances in Cryptology-ASIACRYPT 2010: 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings 16*. 2010.

[14] Aayush Gupta and Kobi Gurkan. "PLUME: An ECDSA Nullifier Scheme for Unique Pseudonymity within Zero Knowledge Proofs". In: *Cryptology ePrint Archive* (2022).

[15] Florian Kluge. *The Many Saints of Privacy: Nullifiers in O1JS*. Accessed: 2025-01-13. Jan. 2025. URL: https://www.o1labs.org/blog/the-many-saints-of-privacy-nullifiers-in-o1js.

[16] O(1) Labs. *o1js*. 2022. URL: https://github.com/o1-labs/o1js.

[17] Ed Akamai Mike Bishop. "HTTP/3". In: *RFC 9114* (2022), pp. 1–64. URL: https://datatracker.ietf.org/doc/html/rfc9114.

[18] Satoshi Nakamoto. "Bitcoin: A peer-to-peer electronic cash system". In: *Satoshi Nakamoto* (2008).

[19] Alexey Pertsev, Roman Semenov, and Roman Storm. "Tornado cash privacy solution version 1.4". In: *Tornado cash privacy solution version* 1 (2019), p. 7.

[20] Eli Ben Sasson et al. "Zerocash: Decentralized anonymous payments from bitcoin". In: *2014 IEEE symposium on security and privacy*. IEEE. 2014, pp. 459–474.

[21] Ertem Nusret Tas et al. "Light clients for lazy blockchains". In: *arXiv preprint arXiv:2203.15968* (2022).

[22] Avail Team. *Avail: A Unifiying Blockchain Network*. 2024. URL: https://github.com/availproject.