The zkVot Protocol

A Distributed Computation Protocol for Censorship Resistant Anonymous Voting

Yunus Gürlek
yunus@node101.io
zkvot.io

Introduction

zkVot is a client side trustless distributed computation protocol designed to achieve anonymous and censorship resistant voting while ensuring scalability. The protocol is created as an example of how modular and distributed computation may improve both the decentralization and scalability of the internet.

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[1]: 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.

[1] See here to read more about protocol versus application differences.

Table of Contents

Background: Today's Internet and the Motivation for zkVot

The current internet is mostly built using a transfer protocol named HTTP, 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. The client is the party requesting some service from an application, while the server is positioned as the provider of this service.

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 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.

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. 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 losing any of the above points.

By providing all of the above points, the zkVot protocol may 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.

[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).

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. 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. Thus, each vote is created in the form of a ZKP (zero knowledge proof) in the voter's personal device.

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

1. Completeness: An honest verifier will be convinced of the result of a valid proof.
2. Soundness: No dishonest prover will be able to convince a verifier to an untrue statement, with the exception of a very small probability.
3. Zero Knowledge: A verifier will be able to learn no additional information but the correctness of the proof during verification.

Moreover, on top of verifying the execution of a process trustlessly and hide any input used in the computation (i.e. private inputs), ZKPs can also output the result of this process. For instance, if an hashing algorithm is written in the form of a ZKP, then by only receiving the ZKP, an honest verifier would be convinced that the prover had an input corresponding to this hash without knowing the actual input.[4]

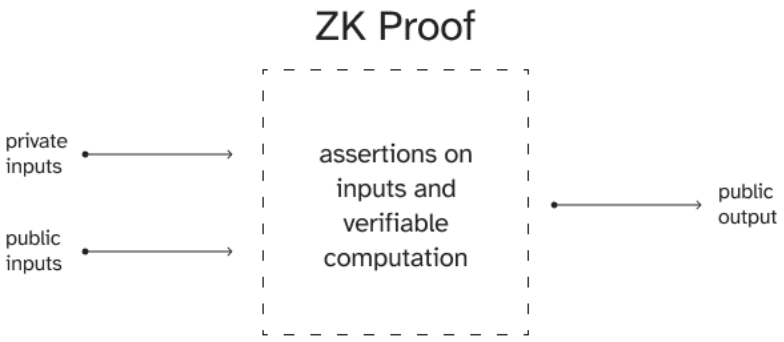

Figure 1: A simple representation of a ZKP.

[3] Here, the proper execution is agreed upon between the prover and the verifier beforehand.
[4] This example is usually referred to as "ZK Login", as it allows users to trustlessly login into their accounts without sending their passwords to a server.

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

In the zkVot Protocol, a specific type of ZKPs, ZK-SNARK proofs, are used. ZK-SNARK proofs provide all the properties of a ZKP, 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[5] 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.

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.

Before coming to what statements the ZKP vote will verify to be true (i.e. the proper execution of the process), let us discuss what output is needed from each ZKP.

First of all, the ZKP 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. Here, more advanced cryptographic techniques, like FHE (fully homomorphic encryption) may be used for hiding the candidate until the election is complete in the future version of zkVot. For the current version, zkVot Protocol does not implement this functionality.

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

Secondly, ZKPs 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 ZKP.

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 ZKP), then we are certain that using the same private key will result in the same nullifier, as hashing is a deterministic process. However, it is impossible to extract the private key from the hash.

In order to output the above information, the ZKP 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 valid public key along with a signature to verify its identity as a private input.[6]
2. The ZKP must assert the public key to be inside the predefined set of public voters.
3. The prover must provide a valid candidate along with a signature to indicate their choice as a public input.
4. The verifier must provide the initial set of allowed public keys as the public input.
5. The ZKP must generate a nullifier inside the proof to prevent double voting while preserving anonymity.
6. The prover must provide a signature of the election identifier as a private input.

The election identifier (named hereafter as the EID) is a random unique constant specific to each election. We use the hash of the EID signature as the desired nullifier. The EID signature is a deterministic value that will be the same for any vote of the same voter, but will differ unpredictably across voters. Using unique EIDs also prevents pseudo anonymity across elections.

A final point 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

[6] It is possible to directly use the private key of the voter as a private input of the ZKP, but since the ZKP will be generated in the browser, accessing the private key is usually prohibited because of security reasons.

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 the ZKP, it is crucial to decrease the size of public inputs as much as possible. As we will explain more later, decreasing the verification costs increases the overall decentralization of the system.

With this new additions of a merkle tree, we update the ZKP as follows:

1. The prover must provide a witness for their public key as a private input.
2. The verifier must provide the merkle root of the public key.
3. The ZKP must compute the root of a merkle tree with the given public key and witness, and assert this root to be equal to the given merkle tree.

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.

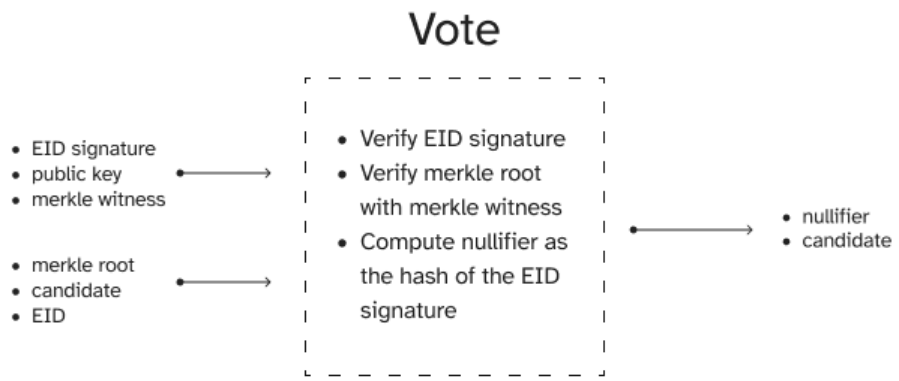

Figure 2: The final design of a vote in the Voting Layer.

With this above design of an online vote in the form of the ZKP, the zkVot protocol ensures the 1st, 2nd, 3rd, and 9th points in the problem definition. For now, we do not define the ZKP generation framework that we will use, as this depends highly on other layers that we will define, and continue with the next layer of the protocol.

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, ZKPs should be stored in a layer and communicated to every verifier in the system. This layer is naturally named as the Communication Layer. 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.[7]
4. The communication should be fast (i.e. the Communication Layer must support a high number of TPS (transactions per second))
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 ZKPs. To its consideration, a TX (transaction) made to the Communication Layer may be a duplicated vote, an invalid ZKP, 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.

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 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.

[7] A meaningful time in the context of an election is usually less than a few days.

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 light 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 involves also 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 ZKPs.

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.[8]

Here, it is worth mentioning that the constant time verification property of ZK-SNARKs allows zkVot to design the Communication Layer in this way. If it was costly to understand if a block is a valid ZKP or not, then a bad actor may be able to censor the system by sending too many invalid ZKPs. However, as this is not the case, the only way to really censor the sending of valid ZKPs is to use all of the available network space, which is equivalent to censoring the entire DA layer, which is prevented by the DA architecture.[9]

Before passing to the next layer, one final question to answer is which DA layer to use, and the 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.
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 and Avail, since they meet all the criterias 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.

[8] 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.
[9] Sending a block to the DA has a cost, thus a full censor of the network would require too much funds.

Aggregation Layer: Counting Votes as a ZKP

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

Getting the result of the election from ZKPs is actually quite simple. Anyone can track published ZKPs 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, as verifying a ZKP is very cheap and you can perform it even in a personal computer. However, as stated above, DA layers are not ideal as a persistent storage. We cannot trust all ZKPs to be available in the long run for anyone to verify the result of the election.

Moreover, allowing everyone to show 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 trustless counting may be using on-chain computation to verify and add up votes. By performing the above process in an on-chain computation environment, 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 computations, and updating the state with the final result of these computations. 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 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 ZKP aggregation is preferred in the zkVot Protocol.

ZKP aggregation is the process of including ZKPs inside another in the form of a different ZKP, also while combining their public outputs. The final aggregated ZKP verifies all the ZKPs 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 ZKP, so that anyone can verify only the final proof to learn the final result. Naturally, zkVot names the layer responsible for ZKP 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 ZKP 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 ZKPs is constant in verification time[10], meaning the final aggregated ZKP is always verified in constant time complexity regardless of the aggregation count. Thus, on-chain verification costs are the same regardless of the election size.
3. Aggregation of ZKPs is fixed in proof size[11], 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.

[10] As stated before, ZK-SNARKs are verified always in constant time, and the property is preserved during aggregation.
[11] Note that this property is true for ZK-SNARKs, but not necessarily for all proving technologies.

However, it is not easy to design an off-chain Aggregation Layer. There are a few important problems to solve to not lose the scalability and decentralization.

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 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[12]. 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.

Even though merkle map provides a huge complexity improvement with the proof generation, ZK-SNARK aggregation is still costly. As a matter of fact, client side aggregation was considered impossible until very recently. Luckily, proving technologies have now improved enough to support efficient aggregation: The zkVot Protocol choses to use o1js, a ZKP generation technology with various advantages:

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

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. As a matter of fact, it is possible to perform even client side aggregation with o1js.

After handling the exclusion check and the proof aggregation performance, there is one final problem to tackle in the Aggregation Layer, which is the censorship resistance of the zkVot Protocol.

As we have described above, it is impossible for the aggregator to lie about the properties of a ZKP included in the aggregation, as the entire combination is secured again in the form of ZKP: With the proper ZKP design, ZKP aggregation may provide trustlessness.

However, off-chain aggregation does not guarantee liveness or 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 ZKP.

There is a straightforward approach to this problem, which is to connect the Communication Layer and the Settlement Layer[13]. By doing so, the Settlement Layer may verify the aggregated ZKP count to be the same as the valid ZKPs on the DA layer. However, this approach is problematic in various ways.

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

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 ZKPs, 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 in the Communication Layer. But unfortunately, this is not really feasible in practice. Proving something to not be a valid ZKP 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.[14]

In order to solve the liveness 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 ZKPs 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[15] 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 actual 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.

---

[14] Note here that the permissionless aspect of the Communication Layer was not a problem before, as the verification time of a ZK-SNARK was constant. However, this is not the case for verifying something is not a valid vote, thus invalid blocks put a huge burden over the Aggregation Layer.

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

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.[16]

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.

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.

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. As explained before, o1js already improves aggregations costs significantly, but it does not change the aggregation complexity in terms of number of voters. Aggregation is a single threaded process, and thus the complexity depends linearly on the number of voters.

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.

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,

[16] Please see the Social Consensus Layer chapter for more details on the incentivization mechanism.

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 dependance.

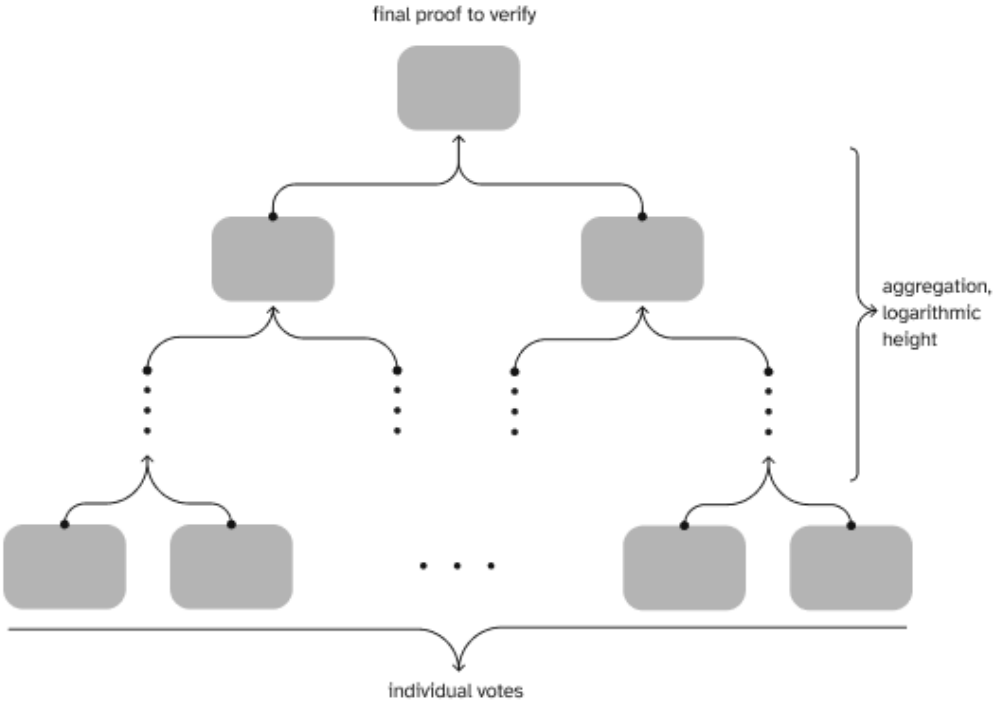

Figure 3: Representation of parallel aggregation with ZK-SNARKs. Leaves of the array are individual ZKPs in the Communication Layer. While computing a node from its children, the aggregation logic makes sure none of the nullifiers in the subtrees are the same.

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 update 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:

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

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:

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

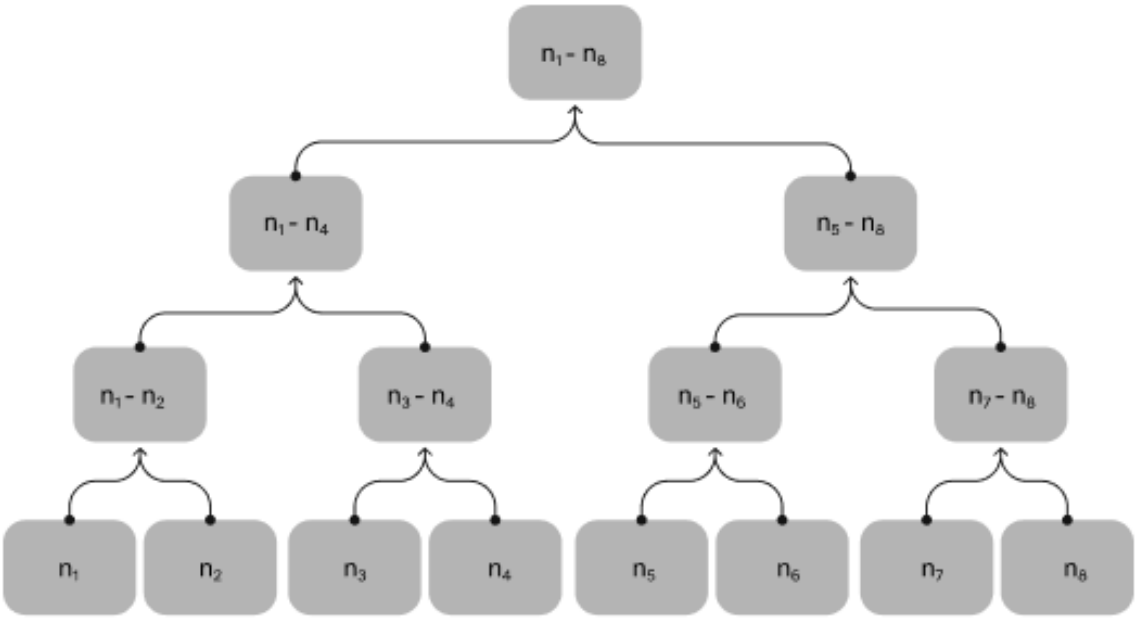

Figure 4: Visualization of the aggregation tree for an election with 8 voters.

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 is that, in this model we need to sort all nullifiers in ascending order before starting aggregation. This is not a problem if all votes are already put on the Communication Layer. However, in order to allow aggregation during the arrival of votes, we should be able to add any nullifier to this tree logarithmically. 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. This hurts the balanceness of the binary tree, but as the nullifiers arrive in random order,[17] there is no risk of the tree getting longer in one branch and turning into a chain. In the current version of the zkVot Protocol, this functionality is not yet implemented.

A final point to emphasize is the fact that parallel aggregation can also be used to improve the censorship resistance of the Aggregation Layer. Through parallel aggregation, votes can be partitioned among aggregators to scale the system in architecture.

In order to motivate different honest aggregators work together, the incentivization mechanism may be designed to not only incentive the aggregator sending the settlement TX, but also to incentive all different aggregators worked together in the aggregation in accordance with the ratio of proofs they have aggregated over total number of aggregated proofs.

Currently, the communication of aggregators is not yet implemented, mainly because an election rarely needs a distributed Aggregation Layer: Even the current setup of the zkVot may be scaled up to millions of voters. However, in more loaded applications like exchange platforms, a similar protocol would need to implement a distributed Aggregation Layer.

With this final notice, we can conclude on the basis underlying the Aggregation Layer and pass to the next section, where we will discuss how to design the Settlement Layer to combine all the layers described above.

[17] Nullifiers are hashes of fixed signatures, so voters cannot attack the system by altering their nullifiers to always be in the same range.

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. And 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, 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.

After making these observations, there is one L1[18] that particularly comes forward for our needs: Mina Protocol.

Mina Protocol is a ZKP verification layer with some very unique properties. In zkVot, the following points highlight the most from Mina's architecture:

1. Off-Chain Computation & On-Chain Verification Approach

   In Mina, there is no on-chain computation environment. Smart contracts on Mina only perform verifications of ZKPs, so in a sense all applications on Mina are already designed to be used with an external aggregation layer. This aligns very well with the zkVot's perspective.

2. o1js Proof Generation Framework

   The choice of proof generation framework that the Aggregation Layer made for efficiency, o1js, is the proof generation framework of Mina Protocol. Thus, Mina is designed specifically for optimizing verification of o1js proofs.

3. Low Network Fees

   All TXs are described as a ZKP, 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[19]. As a result, the settlement costs of the zkVot protocol stays constant over the number of voters.

4. Aggregated Block History for Constant Time and Size Verification

   As well as allowing off-chain proof aggregation, Mina itself is also a huge aggregated ZKP. The entire block history is represented as a constant sized aggregated ZKP: 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.

[18] 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.
[19] The average TX fee in Mina is currently about 0.02 MINA.

5. 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[20] makes Mina full nodes to be almost as accessible as light nodes. Using a light node for settling is not important[21], but this increases long term censorship resistance of reading the election results.

6. Snarket Place for ZKP Aggregation Scalability

   As stated before, ZKP aggregation is costly, and aggregating an entire blockchain is usually not feasible. Mina solves the problem by implementing an open marketplace for SNARK generation, named as the Snarket Place. Instead of aggregating the proofs themselves, Mina full nodes buy proof primitives from SNARK providers in the Snarket Place. As the network load increases, there is a higher need and incentive to generate a SNARK, and the open market approach of the Snarket Place allows reflective scaling. The aggregation speed over the number of TXs ratio stays constant in Mina. This is important for zkVot, as the protocol values the long term scalability of all the layers used, and using a Settlement Layer with reflective scalability to demand is ideal.

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:

1. Low TPS and High Block Time

   Unfortunately, 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). It is important to note here that this is mainly due to the current implementation then the underlying

[20] There is a browser full node implementation for Mina.
[21] This is an equivalent trust observation that is made in the Communication Layer about using RPCs while submitting votes.

architecture[22], and Mina's block time is expected to improve drastically in the near future. Nevertheless, this limitation exists for the current version of zkVot Protocol.

2. Concurrency Issue over State Changes

   In Mina, there is no on-chain computation, and ZKPs are all created outside of the chain. In order to get and use a state inside a TX, o1js requires to put an equality assertion on the used state. Then, the state is provided as a public input to the Mina TX during the verification of the ZKP inside validators. If the state changes before the verification of the ZKP, then the verification fails. As a result, if a TX is using and setting a state variable, then it is possible to accept only one of these types of TXs in the same block. This problem is referred to as the concurrency issue, and it is usually solved by having a single thread on-chain computation to aggregate state updates inside the blockchain[23]. As we have explained above, the parallel nature of zkVot is the real reason behind the powerful scalability, and losing parallel aggregation is not an option. Thus, it is more appropriate to define concurrency issue as a hard property than a problem to solve, and zkVot must provide an architectural approach to work with it without causing scalability concerns.

3. Unique Architectural Design

   In software, being unique is a problem. Mina's choices for the proving system and cryptographic primates are not preferred by the majority of the ecosystem. For instance, while most blockchains use SHA256 or Keccak for hashing, Mina uses Poseidon. Poseidon is ZK friendly and optimizes proof generation time, but it is not very compatible with the SHA family. Thus, it is a huge challenge to use other proof systems or verify different types of signatures in o1js.

4. Limited On-Chain Storage

   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.

[22] Specifically the gossip network's performance in Mina is the bottleneck: Snarket Place and full nodes cannot communicate as efficiently as it should be in the current implementation.
[23] See here for a better explanation of how conventional blockchains use the single threaded approach to solve the concurrency issue.

It is important here to emphasize once more that these 4 points are not external to Mina's architecture, and Mina's unique advantages are actually possible thanks to these hard limitations. Nevertheless, in order to take advantage of all strong properties of Mina, the zkVot Protocol must be designed to work with them in a way that will not cause any decentralization or scalability concerns.

First of all, as explained before, zkVot protocol does not actually require fast finality, as there is a single TX accepted for every election at the end. However, combined with the concurrency issue, it may be a possible attack vector to censor the election result by sending too many invalid TXs to the Mina smart contract. Luckily, in the zkVot design, we do not put any assertions on the previous settled election result. The Aggregation Layer makes sure that each result sent is valid for a subset of votes. However, in order to apply the AMC, the settlement contract needs to store and assert a state variable representing the number of voters in the last settled election result. As a result there is a concurrency issue over this state variable that makes it impossible to accept multiple settlement TXs in a block.

Luckily, o1js has a solution to this, which is to put pre assertions to Mina TXs. Pre assertions are custom assertions that we make on state variables without reading them from the state: The smart contract can assert some condition[24] over a state variable without accessing its value directly. By implementing the AMC as a pre assertion that checks for being greater than, zkVot solves the concurrency issue.

To explain further, in the zkVot Protocol, the settlement contract requires only the TX's election result to include more voters than before. There is no usage of the previous number of voters in the last accepted settlement TX. Thus, it is enough to express this condition as a pre assertion. By doing so, the settlement contract guarantees to accept the TX with the highest number of voters in each block. Note here that the AMC is actually not a voting specific solution, and various types of decentralized aggregation protocols can use the same approach to allow parallel threading without running into the concurrency issue.

The third problem of Mina, having a unique architectural design, is not a protocol problem, but just an engineering challenge. This is another reason why zkVot adapts the AMC: zkVot Protocol makes sure that all available votes are settled and there is no censorship without connecting the Communication and the Settlement Layer in any way.

[24] Currently o1js supports some basic pre assertions like comparison checks, but this is enough for the zkVot Protocol.

Nevertheless, it is important to mention that there exists some bridge implementations to some DA layers from Mina that can efficiently be used. However, it is still much more optimal to not use any bridges; as it minimizes trust assumptions, optimizes proof generation computation, and allows any DA to be adapted to the Settlement Layer. The AMC allows the Communication Layer to use multiple DAs at the same time without any additional implementation. As a matter of fact, the concurrency of multiple DAs is ensured by AMC.

Finally, zkVot needs to find a solution around the very limited on-chain storage capacity of Mina protocol, and there are a lot of optimizations to make the protocol feasible to implement. 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 from the Aggregation Layer.

As a result, only the following informations 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.

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[25]. When there is a constant in a proof, it changes the verification key,[26] and thus causes the verification to fail on Mina.

However, since the last AMC number and the election result are dynamic until the end of the election. We can of course choose another layer to store them, and only have a verification of the results (like the hash) on Mina. However, as stated before, it is hard to implement cross chain updates with Mina, and it is much more optimized to use the 256 bytes we have to store this 2 information.

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

[25] These values are constant once the election contract is deployed on the Settlement Layer.
[26] Each ZKP has a unique constant sized verification key that does not depend on the inputs, but is affected by all assertions and computations happening in the proof. Constants are also a part of the verification key.

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.

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 ZKP 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 can read data from the Storage Layer.
3. It should be impossible to change or delete data from the Storage Layer.

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.

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 an creating a valid election: As the election creator has no privilege on the election,[27] 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.

Currently, zkVot Protocol provides the Arweave, Filecoin, and Pinata blockchains as Storage Layer options. However, it is possible to use any other blockchain without even changing any line of code.

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

Final Design and 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.

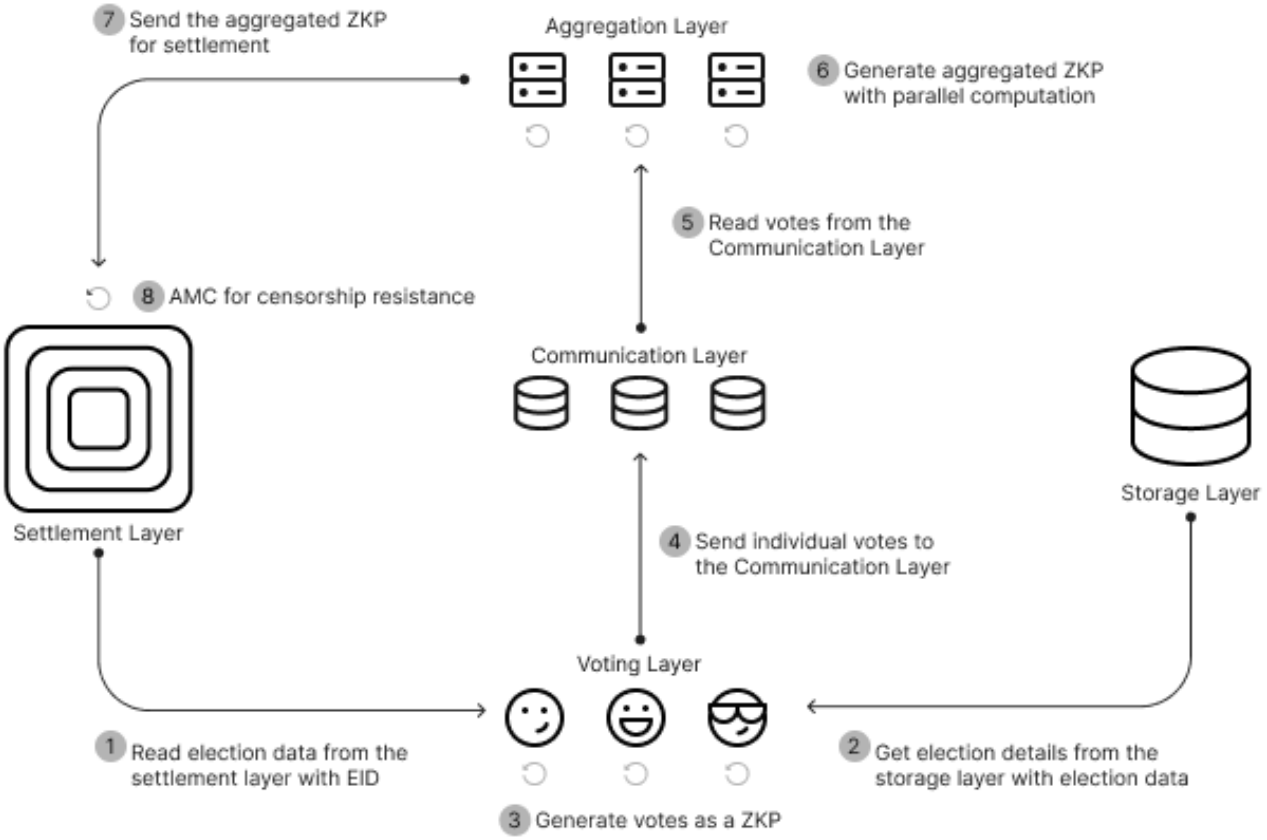

Figure 5: Voting procedure diagram of the zkVot Protocol.

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 ZKP. 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.

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. 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.

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.