Blind-OVOTE:
OVOTE with Blind Signatures

arnaucube, Alex Kampa, Roger Baig

Aragon ZK Research

https://research.aragon.org

Aragon ZK Research is part of the Aragon ecosystem
and is being funded by Aragon Association

November 2022

Abstract

Blind-OVOTE is a Layer 2 voting solution which combines the validity rollup ideas with blind signatures over elliptic curves inside zkSNARK, to provide gasless anonymous voting with on-chain binding execution on Ethereum. The census is based on blind signatures over voters public keys, and votes are aggregated off-chain and proved on-chain through a zkSNARK proof. In this way, voters remain anonymous and gas costs are constant independently of how many thousands of votes are being aggregated. Blind-OVOTE is an evolution of ideas already presented in our former project OVOTE, and also deliberately does not use proof recursion. We provide an implementation of the scheme using the arkworks libraries.

Target audience: web3 developers, advanced DAO practitioners.

Contents

1 Introduction

Blind-OVOTE is a L2 validity rollup combined with blind signatures over elliptic curves inside zkSNARK. The main objective is to provide off-chain anonymous voting with on-chain binding execution on Ethereum, without using costly recursive proofs. Users vote anonymously, and then the aggregation of all the votes and signatures verification is proved through a zkSNARK proof, which is later verified in the Ethereum EVM. Blind-OVOTE is an evolution of ideas already presented in our former project OVOTE [1].

In this document, we present the design of the system, its properties and its drawbacks. Section 2 reviews the preliminaries with a particular focus on blind signatures, as this it the key addition compared to OVOTE. In Section 3 we discuss the zkSNARK circuits, i.e. the signature verification circuit and the circuit that verifies the validity of the ballots. We also describe how we prevent double voting. Section 4 reviews the system flow and Section 5 the resulting properties. We introduce a prototype implementation in Section 6 and conclude in Section 7 with a summary, discuss differences with OVOTE and elaborate on potential next steps.

2 Preliminaries

2.1 OVOTE

OVOTE is a L2 design for voting which is similar to validity rollups ("zkRollups"). Votes are aggregated off-chain and their correct verification and aggregation is proved on-chain through a zkSNARK proof. This results in constant gas costs while scaling up to thousands of voters through a single Ethereum transaction. Further details in this regard can be found in the OVOTE document [1].

2.2 Blind signatures

In a blind signature scheme the signer learns nothing about the content of the message being signed. This is achieved by "blinding" the message before sending it to the signer. Interestingly, the resulting signature can still be verified against the original message like a standard signature, using the signer's public key. We use this concept to enable a Signer to authorize users' public keys without knowing these keys.

2.2.1 Blind Schnorr signature scheme

We use the *Blind Schnorr Signature* scheme as described in [2]. The public parameters consist of a group $\mathbb{G}$ of order p and generator G, and a cryptographic hash function $\mathcal{H} : \{0,1\}^* \to \mathbb{Z}_p$.

The private key of the Signer is a random scalar $x \in \mathbb{Z}_p$ and the corresponding public key is $X = xG$.

Any User who wants to obtain a signature for some message m without disclosing the content of that message to the Signer proceeds as follows:

1. The User sends a signing request to the Signer. This request will typically be signed; thus the Signer knows whether the request is legitimate or not.
2. If the request is legitimate, the Signer generates a random $r \in \mathbb{Z}_p$, computes $R = rG$ and sends R to the User.
3. The User selects random scalars $\alpha, \beta \in \mathbb{Z}_p$, computes the *blinding factor* $R' = R + \alpha G + \beta X$, sets $c = \mathcal{H}(R', m) + \beta \bmod p$ and sends c to the Signer.
4. The Signer computes $s = r + cx \bmod p$ and sends s to the User.
5. The User verifies that the value s received is correct by verifying that $sG = R + cX$. Setting $s' = s + \alpha \bmod p$, the signature of the message m is then $\sigma = (R', s')$.

Anyone can then verify the validity of the signature by checking the equality $s'G \overset{?}{=} R' + \mathcal{H}(R', m)X$. To see why this must hold, we can unroll the equation:

$$\begin{aligned}
s'G &= sG + \alpha G \\
&= rG + cxG + \alpha G \\
&= rG + (\mathcal{H}(R', m) + \beta)X + \alpha G \\
&= R + \alpha G + \beta X + \mathcal{H}(R', m)X \\
&= R' + \mathcal{H}(R', m)X
\end{aligned}$$

Note that blind Schnorr signatures can be subject to so-called ROS[1] attacks, but these attacks can be defended against by forbidding parallel sessions.

2.2.2 Standard Schnorr signatures

The blind-signatures are used by the Authority (Signer) to blind-sign the public keys of the Voters (Users), but the Voters themselves also need to generate a signature of the vote values with their public keys. For this we use the standard Schnorr signature scheme.

In fact, if we take the Blind Schnorr signature scheme, assume that User and Signer are identical, set $\alpha = \beta = 0$ and short-circuit the interactions between Signer and User, the result is precisely the standard Schnorr signature scheme. To sign a message m, the Signer, with secret key x and public key X as above, computes:

$$\begin{cases} R = rG \\ s = r + \mathcal{H}(R, m) \cdot x \end{cases}$$

and outputs $\sigma = (s, R)$ which is equal to (s', R') in the blinded scheme. Both blind and standard signatures can thus be verified in exactly the same way.

[1]Random inhomogeneities in a Overdetermined Solvable system of linear equations

2.3 zkSNARKs

For this project we use arkworks-rs [3], a well-known set of Rust libraries for zk-SNARK programming. We use the Groth16 [4] scheme and the pairing-friendly BN254 elliptic curve. This is the only curve for which elliptic curve addition, scalar multiplication and pairing are precompiled on Ethereum, and is therefore the most practical choice as we want to verify the proofs in Ethereum's EVM. This constrains the other cryptographic primitives we use: the Poseidon hash [5] over the scalar field of BN254, and the BabyJubJub elliptic curve [6] for the signature scheme.

3 Circuits

3.1 Signature verification circuit

The inputs of the R1CS circuit that verifies the voter signatures (both blind and non-blind ones) are shown in Table 1.

Input	Description
m	array containing the signed values
σ	array of signatures $(s, R) \in \mathbb{Z}_p \times \mathbb{G}$
X	public key $\in \mathbb{G}$

Table 1: Inputs to signature verification circuit.

$\mathbb{G}$ is the group of the BabyJubJub curve and $\mathbb{Z}_p$ its base field, which is also the scalar field of the BN254 curve. The circuit checks the equation $sG \overset{?}{=} R + \mathcal{H}(R, m)X$, where $\mathcal{H}$ is the Poseidon hash. Note that all circuits are computed for the BN254 curve, while we use BabyJubJub for the signature scheme.

3.2 Blind-OVOTE circuit

The main circuit verifies the validity of the individual ballots, each of which includes a signature of the Signer and of the voter, along with the vote itself. This circuit also verifies that the result of the votes aggregation was computed correctly. Its inputs are shown in the Table 2.

Public inputs		Private inputs	
Input	Description	Input	Description
chainID	blockchain identifier; hardcoded in the Contract deployment	pk_{u_i}	users public keys
processID	process identifier; determined by process creation	w_i	weight of each pk_{u_i}
pk_{A_j}	Authorities public keys, determined by process creation	v_i	votes values
R	result	σ_{A_i}	authority signatures over $pk_{u_i} + w_i$
		σ_{u_i}	users signatures over v_i

Table 2: Blind-OVOTE circuit public and private inputs.

Note that the weights are an optional feature.

The checks defined by the circuit constraints are:

- $pk_{u_i} + w_i$ are signed by pk_{A_j}
- v_i is signed by pk_{u_i}
- $v_i \in \{0, 1\}$
- $R = \sum v_i \cdot w_i$
- There are no repeated pk_{u_i}

3.3 Preventing double voting - nullifier approach

To ensure that public keys are not used reprepeated in a results proof, we define a nullifier of the form:

$$N_i = \mathcal{H}(chain_{ID},\ process_{ID},\ pk_{u_i}.x,\ pk_{u_i}.y)$$

The elements being hashed are: $chain_{ID}$ to prevent the reuse of a vote on a chain with a different chain id; $process_{ID}$ to prevent the reuse of a vote in another voting process; finally $pk_{u_i}.x$ and $pk_{u_i}.y$ which are the coordinates of the voter's public key and prevent double voting.

3.3.1 Traditional approach

The nullifier computation is usually checked inside the zkSNARK circuit, the Ethereum smart contract then checks that there are no duplicates. While costly in terms of gas, it is useful if different zkproofs need to be combined sequentially over time. As we do not have this requirement, we can use a simpler solution.

3.3.2 Our approach

Instead of checking nullifiers in the smart contract, we do so inside the SNARK circuit. As we cannot use a hashmap nor do a loop over all the nullifiers, our solution to ensure that there are no repetitions is the following:

- In the circuit inputs, the ballots are sorted by nullifier value;
- The circuit checks that the nullifiers are in strictly increasing order.

Inside the SNARK circuit we cannot make use of boolean operators such as `if h[i-1] >= h[i] then fail`. Instead, we use a trick from circomlib's circom implementation of the *'LessThan'* circuit [7]. It relies on the following property:

Let a and b be integers between 0 and $2^n - 1$ of bit-length $\leq n$. Let $c = 2^n + a - b$. Denote by $c_b[n]$ the factor of 2^n in the binary representation of c. We then have:

$$c_b[n] = \begin{cases} 0 \text{ if } a < b \\ 1 \text{ otherwise} \end{cases}$$

This property also holds for operations modulo $q \geq a^{n+1}$, i.e when q is of bit-length of at least $n + 2$. In the circuit, c is computed as follows, with $\ll$ representing the *shift left* operation:

$$c = a + (1 \ll n) - b$$

For this to work, we need to set $n = 252$ and crop the Poseidon hash by two bits, from 254 to 252 bits, for a negligible loss of security. This is because BN254 is defined over a prime field $\mathbb{F}_r$ whose bit-length is 254.

4 Flow

There are five main phases: census and process creation, credentials issuance, vote casting, votes aggregation, results publishing.

As shown in Figure 1, the census and process creation phase is comprised of the following steps:

1. Organizer sends a list of valid EthAddresses to Authority
2. Authority creates pk_{A_j}
3. Authority sends their pk_{A_j} to the Organizer
4. Organizer creates new process in the Ethereum smart contract: newProcess(pk_{A_j}, txHash, ...)

The credentials issuance process presented Figure 2 entails the following steps:

5. Voter requests blinding parameters
6. Authority returns blinding parameters
7. Voter creates pk_{u_i}

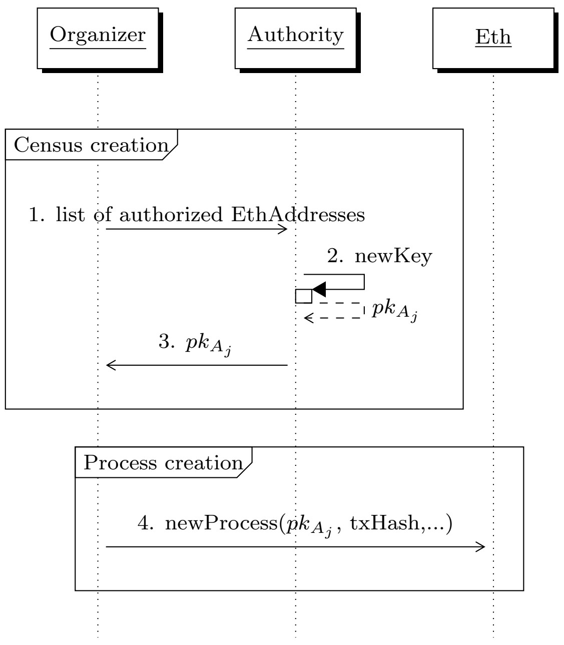

Figure 1: Census creation.

8. Voter blinds pk_{u_i}, generating $[pk_{u_i}]_B$. Here, if weights are also needed, voter would include the weight value in the blinding, and provide a zkproof to the Authority, proving that the correct weight was blinded and included in the blinded package.
9. Voter sends $[pk_{u_i}]_B$ to Authority
10. Authority authenticates voter EthAddr, and blind signs voter's $[pk_{u_i}]_B$
11. Authority returns the blinded signature over voter's public key: $\sigma_A([pk_{u_i}]_B)$
12. Voter unblinds $\sigma_A([pk_{u_i}]_B)$, obtaining $\sigma_A(pk_{u_i})$

Notice that credentials issuance process could be performed with multiple authorities, cf. Multiple Authorities section (4.2). Also, step 3 can be done by any party.

The Authority role described in the previous steps from Figure 2, is referred to as Credential Service Providers (CSP) in the Vocdoni protocol [8]. In our description the CSP is authenticating the users based on their Ethereum addresses and signatures, but for other use cases it could be done with other means such as emails, SMS codes, etc.

As shown in Figure 3, the remaining steps are the following:

13. Voter signs their vote option $v_i \in \{0, 1\}$, obtaining $\sigma_{pk_{u_i}}(v_i)$
14. Voter sends the ballot to RollupNode, $b_i = \{v_i,\ pk_{u_i},\ \sigma_A(pk_{u_i}),\ \sigma_{pk_{u_i}}(v_i)\}$

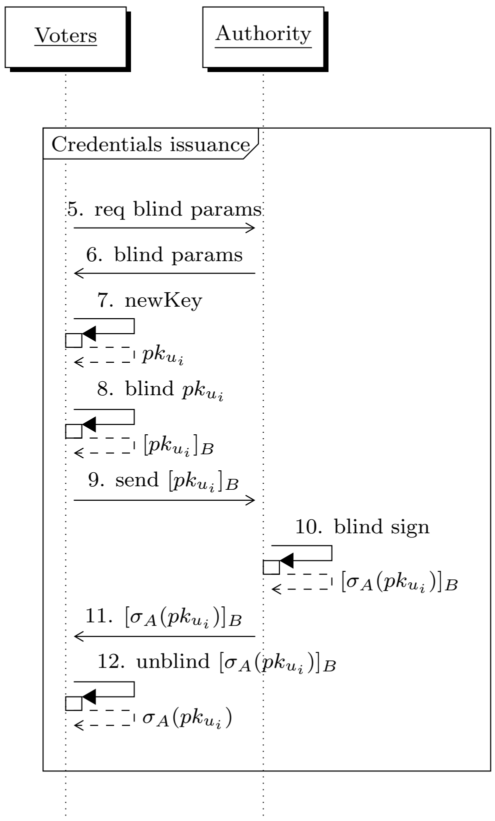

Figure 2: Credentials issuance.

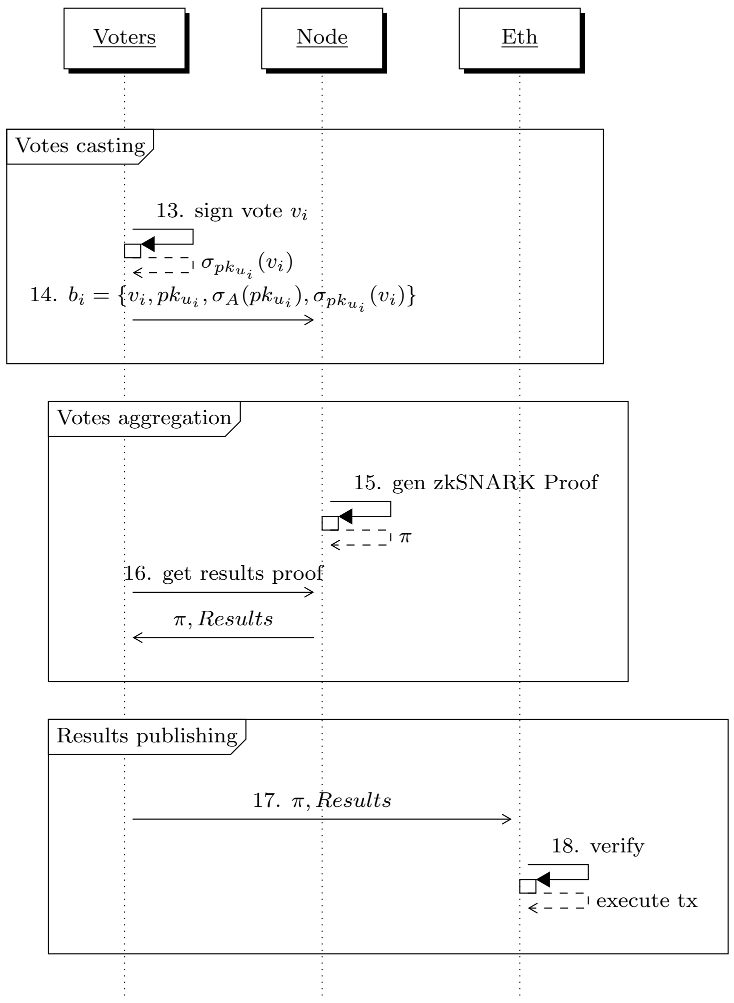

Figure 3: Votes casting, votes aggregation and results publication.

15. Rollup Node generates zkSNARK proof π (which proves the conditions listed in section 3.2)
16. Any user can retrieve $\pi + Results$
17. Any user can send $\pi + Results$ to the smart contract
18. SmartContract verifies π for the given $Results$ and the initially defined pk_A, and executes the predefined transaction

4.1 Multiple Nodes

The results publication approach in Blind-OVOTE is the same described in OVOTE [1] section *4.8.1 Multiple nodes*. Once the voting period is over and *ResultsPublishingStartBlock* is reached, any party can submit an Ethereum transaction with the result and the zkSNARK proof.

Having a single aggregator node could lead to censorship, as the node operator could ban certain votes. To prevent this, there can be multiple nodes, run by different non-colluding parties. The first node to reach the minimum amount of positive votes can generate a valid proof that will be accepted by the smart contract.

4.2 Multiple Authorities

To prevent a single Authority from introducing fake voters, the system can be set up to require several blind signatures from different authorities. In the case of a DAO for example, well-known members of the community could be tasked with doing the blind signatures, so that the census will be safe as long as they don't collude. In later iterations, we could add thresholds for the authority signatures (e.g. 2 out of 3).

4.3 Costs

4.3.1 Gas costs

Gas costs of Blind-OVOTE are mainly the cost of Groth16 zkSNARK on-chain verification. With the current EVM, verifying a Groth16 proof costs about 250k gas. This cost is constant for any number of votes being aggregated.

4.3.2 R1CS constraints

The system requires approximately 14k R1CS constraints per voter with a single Authority, and an additional 6,052 constraints per voter for each additional Authority. This means that with a single Authority, we need about 14 million constraints for 1,000 Voters, and 143 million constraints for 10,000 voters. With 3 Authorities the number of constraints for 1,000 (resp. 10,000) voters increases to 26 million (resp. 260 million).

The number of R1CS constraints directly affects the proof generation time. However, unlike for economic transaction rollups, in our use case the proving time is not critical. The proof is only computed once at the end of the voting process and it is acceptable to wait some minutes, or even some hours, for the generation of the proof.

4.4 Vote receipts

Vote receipts allow users to verify that their vote was included in the published results, without being able to prove the value of their vote. This is an optional feature of OVOTE which has not been included in the initial implementation of Blind-OVOTE, but could be added in a later iteration. More details on how votes receipts work can be found in OVOTE [1] section *4.8.2 Vote receipt*.

5 Properties

The following properties are shared with OVOTE:

Universal verifiability The way we use the validity proofs makes the results verfiable by any actor.

Off-chain/gasless voting users vote off-chain, and the RollupNode aggregates the computation and verification of all the votes, signatures and census-proofs, in a succinct validity proof, which is sent to the Smart Contract. The only transactions executed on-chain are the *process creation* and the *results publishing.*

Scalability Thousands of votes can be aggregated in a single Ethereum transaction. This is enables multisigs orders of magnitude larger than current on-chain solutions, among other benefits.

Chain agnostic census The census is build off-chain, and the proof of correct results computation can be published into any EVM chain (furthermore, into any chain that supports Pairing computation). So a Blind-OVOTE census could be used in Ethereum mainnet, but also in other chains.

User anonymity By the usage of blind signatures, user identity is kept anonymous in front of all the different parties (Authority, Rollup Node, Ethereum).

In addition, Blind-OVOTE has the following property:

User anonymity By the usage of blind signatures, user identity is kept anonymous in front of all the different parties (Authority, Rollup Node, Ethereum).

In the cases where the census is accepted to be correct and the Authorites to perform honestly (see 5.1 and 4.2), the following property also applies:

Binding execution Due to the universal verifiability property, the proof verification can trigger on-chain actions (e.g. moving funds of a DAO) in a trustless way, directly from the voting process result.

5.1 Drawbacks

As with OVOTE, there some UX friction when having an interactive census creation due the nature of the scheme of blind signatures.

One additional drawback of the current design is that that the census is not auditable, so that the **unforgeability** property is lost when Blind-OVOTE is used with a single Authority which would be able to add fake voters and it would not be possible to detect this. A mitigation to this was described in the section Multiple Authorities (4.2) above.

6 Implementation

We did a prototype implementation of this scheme using arkworks [3]. First we implemented the blind signatures verification scheme inside a SNARK circuit (R1CS constraints), together with a Rust library implementing the whole signature scheme over the BabyJubJub elliptic curve. And secondly, we implemented the Blind-OVOTE scheme, making use of the blind-signatures and adding all the vote verification logic inside the SNARK circuit.

Both implementations can be found in:

- aragonzkresearch/**ark-ec-blind-signatures**: Blind signatures over elliptic curve implementation (native & R1CS constraints). Github [9]
- aragonzkresearch/**blind-ovote**: Blind-OVOTE scheme implementation, contains the library to be used in Voter's browsers, the Authority server, and the Rollup node. Github [10]

7 Conclusions, comparison to OVOTE and future work

Both schemes, OVOTE and Blind-OVOTE, aim for scalability, where users vote off-chain but the results are verified on-chain, aggregating the census proofs verification and voters signatures in a single zkSNARK proof which is sent in a single Ethereum transaction.

The main difference between OVOTE and Blind-OVOTE is the approach used for the census.

In the case of OVOTE, the census is contained in a MerkleTree which can be publicly audited. Anyone can verify that all of the voters' public keys are in the Census Tree and that no extra keys were added.

In the case of Blind-OVOTE we use blind signatures over the user's public keys. The benefit of this approach is the voter's anonymity, but it comes with the drawback of non-auditability of the whole census. This is mitigated by having more than one Authority participating in the census creation, as explained in section Multiple Authorities (4.2).

As previously mentioned, Blind-OVOTE does not use recursive proofs to keep the number of constraints low while using R1CS tooling. A potential next step is to look into recursive proofs as they can enable, for example, the voters to generate a zkproof of membership in the browser without revealing who they are in the census (built for instance as a MerkleTrees). These zkproofs can be aggregated into a single zkproof which can then be verified in Ethereum. Proof recursivity would also allow for further designs where different results from different aggregators are merged into a single proof while avoiding double vote counting.

While not an ideal solution, Blind-OVOTE might be another step towards a future of highly scalable voting systems with off-chain aggregation and on-chain verification in Ethereum.

Acknowledgements

We would like to thank Vincenzo Iovino for useful feedback on the Schnorr signatures, and Rebekah Mercer for her comments on the nullifiers approach. Also we want to thank Pau Escrich (Vocdoni) with whom during the last year we have been iterating ideas for scalable voting solutions using zkSNARKs. It is worth noting that Vocdoni uses blind-signatures in Vochain, Vocdoni's blockchain.

This work has been funded by Aragon Association.

References

[1] OVOTE. https://research.aragon.org/docs/ovote.

[2] Georg Fuchsbauer, Antoine Plouviez, and Yannick Seurin. Blind schnorr signatures and signed elgamal encryption in the algebraic group model. Cryptology ePrint Archive, Paper 2019/877, 2019. https://eprint.iacr.org/2019/877.

[3] arkworks contributors. `arkworks` zkSNARK ecosystem. https://arkworks.rs, 2022.

[4] Jens Groth. On the size of pairing-based non-interactive arguments. Cryptology ePrint Archive, Paper 2016/260, 2016. https://eprint.iacr.org/2016/260.

[5] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Poseidon: A new hash function for zero-knowledge proof systems. Cryptology ePrint Archive, Paper 2019/458, 2019. https://eprint.iacr.org/2019/458.

[6] Barry WhiteHat, Jordi Baylina, and Marta Bellés. iden3's babyjubjub elliptic curve. https://iden3-docs.readthedocs.io/en/latest/_downloads/33717d75ab84e11313cc0d8a090b636f/Baby-Jubjub.pdf.

[7] circomlib less-than trick. https://github.com/iden3/circomlib/blob/master/circuits/comparators.circom#L89.

[8] Vocdoni csp. https://docs.vocdoni.io/architecture/census/off-chain-csp.html.

[9] Blind signatures over elliptic curve implementation (native & r1cs constraints). https://github.com/aragonzkresearch/ark-ec-blind-signatures.

[10] Blind-OVOTE implementation. https://github.com/aragonzkresearch/blind-ovote.