Off Chain Voting and On Chain Contribution

Introduction

This document outlines the use of ZKP in the BP-EY Climate DAO project protocol. Variable names may be different from those in the code.

Registering (whitelisting)

User will register with Application. BP will perform KYC and confirm registration giving users access to voting and contribution.

The application will only allow votes and submit contributions from these users.

Voting and Contribution

Let us say there are 3 Projects A, B and C

- User will sign into application and user will then contribute values $C_{A_0}, C_{B_0} and\ C_{C_0}$ (these contribution values could be non-perfect squares)
- Application will then calculate the square root for each of these, multiply to shift the decimal point, and floor:

$$\mathrm{V}_{A_0} = \mathrm{floor}(10^6\sqrt{C_{A_0}})$$
$$\mathrm{V}_{B_0} = \mathrm{floor}(10^6\sqrt{C_{B_0}})$$
$$\mathrm{V}_{C_0} = \mathrm{floor}(10^6\sqrt{C_{C_0}})$$

  Where V_{A_0}, V_{B_0} and V_{C_0} are whole numbers approximating the square root of their respective contributions. These represent votes in quadratic funding.

- Application will then encrypt these along with C_{A_0}, $C_{B_0}$and C_{C_0} with the public key P_{op} of the operator. To perform encryption of each message, we need this to be represented as a point on the elliptic curve. We use repeated point addition of the generator G to calculate:

$$M_0 = V_{A_0} * \mathrm{G}$$
$$M_1 = V_{B_0} * G$$
$$M_2 = V_{C_0} * G$$
$$M_3 = C_{A_0} * G$$
$$M_4 = C_{B_0} * G$$
$$M_5 = C_{C_0} * G$$

  For every message M_i that needs to be encrypted, we pick an ephemeral key k_i which is a random nonzero number in field F_p. We will then encrypt to generate cipher text $(R, S) = \left(kG, M + kP_{op}\right)$ for each message as follows and then store in database. For the votes of project A:

$$(R_{t_A}, S_{t_A}) = (R_{t_A} + R_{A_0}, \quad S_{t_A} + S_{A_0}) = (R_{t_A} + k_{A_0}G, \quad S_{t_A} + V_{A_0}G + k_{A_0}P_{op})$$
$$= \left(\left(\sum_{i=0}^{n-1} k_{A_i}\right)G, \quad \left(\sum_{i=0}^{n-1} V_{A_i}\right)G + \left(\sum_{i=0}^{n-1} k_{A_i}\right)P_{op}\right)$$

Similarly, the votes for projects B and C (encrypted in (R_{t_B}, S_{t_B}) and (R_{t_C}, S_{t_C})) and the contributions for all projects (encrypted in (R_{C_A}, S_{C_A}) , (R_{C_B}, S_{C_B}), and (R_{C_C}, S_{C_C})) are stored as an encrypted running total. Due to the above homomorphic property, this running total is the tally of votes or contributions and allows us to decrypt just a single value for each.

- Ordinarily, the encrypted tally would be calculated and stored in the contract, since the addition $(R_{t_A}, S_{t_A}) = (R_{t_A} + R_{A_0}, S_{t_A} + S_{A_0})$ reveals no private values and can be completed trustlessly. To save gas, we instead perform this off-chain.

- Users can see in UI that their contribution will be sent to the contract via a USDC transfer and their vote is not *exactly* the square root of their contribution, but it is very close (under 0.5% inaccuracy depending on how close the contribution is to a square). If they agree to this, they will sign the meta transaction which will send contributions to $contributeSigned()$ function on contract via BP.

- BP will check the below and then send the transaction to the $contributeSigned()$ function:
  - if the encrypted data is encrypted using P_{op}
  - if the contribution matches what is held in the encrypted data
  - if the user is in the whitelist

- If the user wishes to send the transaction themselves, instead BP will provide a signature if the above checks pass. This signature allows the user to call $contributeSigned()$.

- $contributeSigned()$ function will transfer $C_{A_0} + C_{B_0} + C_{C_0}$ USDC to its contract address by calling the relevant function in USDC contract. This value will also be stored against the user in $total\_contrib_{user\_1}$. It will also be added to the value in $total\_contrib_{all\_users}$ (originally initialised to 0).

- After voting is done (at the end of a predefined time), operator will generate a proof that shows they can decrypt the cipher texts such that

$$f((R_{t_A}, S_{t_A}), (R_{t_B}, S_{t_B}), (R_{t_C}, S_{t_C}), (R_{C_A}, S_{C_A}), (R_{C_B}, S_{C_B}), (R_{C_C}, S_{C_C}))$$
$$= (total\_votes_A, total\_votes_B, total\_votes_C, total\_contrib_A, total\_contrib_B, total\_contrib_C)$$

- This proof will also show that:

$$public\_input\_hash = sha256((R_{t_A}, S_{t_A})', (R_{t_B}, S_{t_B})',$$

$$(R_{t_C}, S_{t_C})',$$
$$(R_{C_A}, S_{C_A})',$$
$$(R_{C_B}, S_{C_B})',$$
$$(R_{C_C}, S_{C_C})',$$
$$total\_votes_A, total\_votes_B, total\_votes_C, total\_contrib_A, total\_contrib_B, total\_contrib_C)$$

Where $(R, S)'$ is the compressed version of (R, S), found using Edwards elliptic curve compression. This hash allows us to compute the expensive verification calculation on just one public input, rather than 20+, while losing no integrity.

- On successful verification of this proof by the function $verifyVotesAndDistribute(\quad)$, the contract will confirm the calculation of $public\_input\_hash$ and distribute the funds based on the quadratic funding formula:

$$T := total\_votes_A^2 + total\_votes_B^2 + total\_votes_C^2$$

$$matching\_pool\_contrib_A = \left(\frac{total\_votes_A^2}{T}\right) * matching\_pool$$
$$matching\_pool\_contrib_B = \left(\frac{total\_votes_B^2}{T}\right) * matching\_pool$$
$$matching\_pool\_contrib_C = \left(\frac{total\_votes_C^2}{T}\right) * matching\_pool$$
$$assert(total\_contrib_A + total\_contrib_B + total\_contrib_C == total\_contrib_{all\_users})$$

$$total\_contrib_A = matching\_pool\_contrib_A + total\_contrib_A$$
$$total\_contrib_B = matching\_pool\_contrib_B + total\_contrib_B$$
$$total\_contrib_C = matching\_pool\_contrib_C + total\_contrib_C$$

and will then transfer $total\_contrib_A$, $total\_contrib_B$ and $total\_contrib_C$ to corresponding project addresses which will be registered in the contract at the time of deployment.

Trust assumptions

- Since we are voting off-chain, BP may provide any encrypted tally it wishes if the $total\_contrib_{all\_users}$ matches the contract value. The data source (here, an off-chain database queried by BP) must be trusted by the user.
- What is stopping BP from submitting wrong votes on behalf of a user that still is equal to the value of the contribution that the user signed for? Such as
    - BP can change the split of votes between projects made by a user
    - BP can manipulate the encrypted votes from a user such that one vote from one user can be split into more than one vote coming from different users for the same project while calculating the sum of the sqrt of the votes. All the while making sure that the sum of these manipulated votes is equal to the contribution made by the user. This can be used to manipulate the matching funds outcome for each project.

If the users submit the vote directly on chain and if the validity of this vote is proved by verifying a zero-knowledge proof on chain then the above situations are restricted. This is part of the design of Solution 2

suggested by EY which uses on-chain voting with Homomorphic Encryption as well as Solution 3 which is a MACI like approach.

- BP controls who are added to the whitelist
- BP can add its own addresses to the whitelist and manipulate the outcome by submitting multiple votes from different addresses for the same project
- BP can censor transactions sent by whitelisted users

Solution 2 will provide censorship resistance to the extent that the cost of transaction for a user to submit their vote and contribution on chain is sensible against the contribution they intend to make. Solution 3 suggested by EY will provide censorship resistance.

Statement Proven

We are proving that all the contributions that have been submitted on chain will be used in calculating the total users' contribution for all projects. However, the calculation for total users' contribution could be compromised because the operator could have manipulated the votes.

To Note

- Because KYC is not on chain, contributions are submitted by BP because BP needs to ensure the votes and contributions are the same and if they are in the whitelist. If users can do this directly, then this can't be ensured.
- We are using zkp to hide the private keys of the operator not the votes by a user for each project
- We are using Homomorphic encryption to hide the votes by a user for each project
- Who holds the users' signing keys for meta transaction signing? The user in their browser and not EY or BP
- BP can choose to decrypt a total that does not include a user's contribution even though their contribution is accepted by the contract, in which case, all the transfers for projects will fail because of the assertion statement that will ensure the sum of project contributions is equal to the balance held by the contract
- BP can choose to decrypt a wrong total that is more than the sum held by the smart contract, in which case, all the transfers for projects will fail because of the assertion statement that will ensure the sum of project contributions is equal to the balance held by the contract
- If the contract holds less than the value calculated by circuit for each project, then contributions were not made for all the votes used in the tallying of $total\_votes^2_{project}$ and $total\_contrib_{project}$ tally. If the contract holds any balance, then not all the votes corresponding to the contributions on chain made were used in the tallying of $total\_votes^2_{project}$ and $total\_contrib_{project}$ tally. Each of these situations will fail because of the $assert$ statement