PolkaVote

This project marks the first steps toward real life applicable ZK use cases for the Polkadot community. By leveraging the state of the art ZK tooling such Noir, ZK-SNARKS,HONKS and Pederson Commitment, “PolkaVote” introduces an anonymous voting protocol to the Polkadot eco-system for anonymous and decentralized governing.

The approach of the project is to extend the existing functionality of the Open Gov Voting mechanism that exists in the Polkadot eco-system and introduce anonymity to it.

On the technical side, Noir was chosen with developer experience as a top priority. Unlike other ZK technologies that require deep knowledge of mathematical circuit design, Noir provides a Rust/Solidity-native environment and includes a compiler that automatically generates circuits. This means developers never need to manually design circuits—everything happens under the hood within a robust Noir development workflow.

How does it work?
The protocol is designed with a privacy first approach, leveraging ZK-SNARKS (HONKS) to obscure the on-chain published votes.Voting evaluation leverages homomorphic property of the Petersen Commitment to evaluate the final voting result in an oblivious way on top of the ciphertext space.
Once the encrypted result of the aggregate votes is evaluated the protocol uses the ZK proof to assert and leak only the data that it was designed to -- the aggregated voting result.

The solution does not anonymize voters identities but only their vote (Yay or Nay), such a setup was selected for practical reasons.

At its core, the idea is to create a flexible, reusable and robust Zero Knowledge voting platform on Polkadot for Polkadot.

Submission track name

1. Kusama Zk bounty
2. Main track Polkadot

Team and Project Information

Team name

CrackedDevZ

Team Members

1. Gil Henkin, gil7788@gmail.com
2. Armando Medina, Armsves@gmail.com
3. Zach Kornberg, zkorn1@icloud.com

Project Architecture

Project's architecture comprises of several components:

1. Frontent
2. Backend
3. Solidity smart contracts
4. Noir circuits

Front End

The Front end responsible for the noir proof generation, and on chain communication, including on chain verification. The frontend creates a pedersen commitment per voter where each commitment is recorded to the chain state. Recording the commitment to state constrains the off-chain server to ensure the integrity of the result. The circuit constrains the vote output to match the summation of commitments created by each user, thus preventing voter starvation (the server withholds certain user votes) and attempts to tamper with a given user's vote.

Back End

The server is responsible to evaluate the total aggregate votes once the voting is done and over. The server introduces a single point of centralization that can be eliminated by incorporating solutions like **MPC** and Shamir Secret Sharing. As mentioned above, onchain commitments create strong guarantees regarding the authenticity of the vote. Despite a single point of failure, the server is only liable for liveness failures in the protocol. Moreover, given that the plaintext is revealed offchain, we must currently trust that the server will not dox the users.

Solidity Smart Contracts

The solidity smart contracts are responsible for the on chain state management, on chain proof verification and commitment binding. Yet another interesting functionality of the smart contract system is the nullifier mechanism that accounts for and prevents replay attacks, similar to the mechanism used in tornado cash to prevent double spends from a pool. The commitments stored onchain anchors the privacy and security guarantees of the system.

Noir Circuits

The Noir circuits are responsible for proving the legitimacy of the vote under ZK assumptions (hiding + binding) and performing additive homomorphisms over the set of commitments stored in the chain state. This circuit generates the validity proof for the reported result and asserts the summed commitments are equal to the result. The submission of the final result is done at the end of the voting period.

Single Point of Centralization

As the protocol presented in the current iteration, there exists a single point of centralization that defeats the purpose of a "fully private" solution. However it is usually the case where Cryptographic protocols are implemented and developed for a relaxed setup first and improved iteration by iteration. This hackathon marks the first milestone of the PolkaVote protocol. Check out the following section for protocol improvements and enhancements :)

Implementation Details

Polkavote solution provides a transparent end to end solution with privacy as its first concern. This section presents the flow of the application starting from voters' browser all the way through generating a ZK proof with Noir circuits, to Solidity smart contracts ZK verification and vote aggregation and eventually back to the voters browser as a plain final result of the voting.

As already was mentioned, the voting protocol starts with

1. Noir Proof Generation

The following code is responsible for Noir circuit generation, the generated circuit ensures the validity of the vote value and the correctness of the ECDSA signature on message hash, which includes - timestamp, proposal id, user id, user address and vote.

```
struct Vote {
    public_key_x: [u8; 32],
    public_key_y: [u8; 32],
    value: i8,
    message_hash: [u8; SHA256_WORD_LEN],
    signature: [u8; 64],
}


▶ Compile | Info | Execute | Debug
fn main(
        public_key_x: [u8; 32],
        public_key_y: [u8; 32],
        is_upvote: bool,
        message_hash: [u8; SHA256_WORD_LEN],
        signature: [u8; 64],
        ) {
    let v: Vote = create_vote(public_key_x, public_key_y, is_upvote, message_hash, signature);
    vote(v);
}

fn create_vote(public_key_x: [u8; 32],
               public_key_y: [u8; 32],
               is_upvote: bool,
               message_hash: [u8; SHA256_WORD_LEN],
               signature: [u8; 64]) -> Vote {
    let mut vote_value: i8 = -1;
    if is_upvote {
        vote_value = 1;
    }
    let vote: Vote = Vote {
        public_key_x,
        public_key_y,
        value: vote_value,
        message_hash,
        signature
    };

    vote
}

fn vote(vote: Vote) {
    let valid_signature: bool =
        std::ecdsa_secp256k1::verify_signature(vote.public_key_x,  vote.public_key_y, vote.signature, vote.message_hash);
     assert(valid_signature);
    assert((vote.value == 1) | (vote.value == -1));
}
```

2. **ZK Browser**

The proof is generated on a chain leveraging the **Barretenberg** solution provided by **Aztec**.

```
const message = `${timestamp},${proposalId},${address},${userId},${is_upvote}`;
// Sign Message
const result = await getSignatureAndPublicKey(message, signMessageAsync);

if (result) {
  const { publicKeyX, publicKeyY, message, signatureBytes } = result;

  const input = {
    public_key_x: publicKeyX,
    public_key_y: publicKeyY,
    is_upvote,
    message_hash: message,
    signature: signatureBytes,
  };

  // Generate Witness
  const { witness } = await noir.execute(input);
  setWitness([JSON.stringify(witness)]);
  show(setLogs, `Generated witness ✅`);

  // Prove witeness/message constraints
  const proofData = await backend.generateProof(witness);
  const {publicInputs, proof} = proofData;
```

3. **On Chain State Management with Paseo**

Paseo is used as a Solidity based blockchain to manage state. While the original plan was to host all the smart contract systems on the **Paseo** chain, we ran (once again) into a contract limit size - it was verified and approved by several mentors (including Tiago and Torsten). As an unfortunate work around we had to bridge the gap of the technical capability and deploy the smart contract system onto 2 different block chains - **Paseo** and **Ethereum Sepolia**.

4. **Leveraging Additive Homomorphism of Pedersen Commitment for Oblivious Evaluation On Chain**

Pedersen commitments were specifically selected so certain computation could be performed on chain without compromising voter privacy and security. **Pedersen Commitments** under **ZK** provide an interesting combination of private commitments that are still flexible to support evaluation on the ciphertext mainly because there is a **homomorphic** relation between the committed and plain value; **Homomorphism in a Nut Shell:**

Formally, a map $f : A \to B$ preserves an operation μ of arity k, defined on both A and B if

$$f(\mu_A(a_1, \ldots, a_k)) = \mu_B(f(a_1), \ldots, f(a_k)),$$

for all elements $a_1, \ldots, a_k$ in A.

In our case **F** is the **Pedersen Commitment** map and **a**'s are committed values.

5. **Revealing The Final Result of a Bit Commitment**

The last part of the protocol is the trickiest, in the solution that we provided today it requires a single point of centralization - a web server that is able to compromise the privacy of all the voters at the end of the vote, however with future research and development a solution with **Multi Party Computation (MPC)** can be advised to eliminate the single point of centralization.

Implementation Challenges

I. **Barretenberg backend**
   A. Barretenberg can compile to web assembly and thus the cryptographic primitives it provides are callable from the browser. **Aztec** provides glue code for the WASM module to be accessed by the javascript runtime public APIs on top of the glue code for a better developer experience.
   B. While we were implementing the frontend, we were able to successfully initialize **Barretenberg**; however, despite providing field elements to the **pederson commitment** call and the blinding factor, the call to the method hangs indefinitely. This can be any number of things.

II. **Circuit**
   A. One challenge we encountered while implementing the circuits was figuring out how to do the **homomorphic** operations on the **pedersen commitments**. At first I did not recognize that the output of the **pedersen commitment** is a point on the curve. Given that the output is a curve point, it is straightforward to add two curve points together where the group operation is addition.
   B. In the implementation we tried to pass in an array of field values for the commitments; however, we could not cast Field to an EmbeddedCurvePoint. We were able to go the other way -- changing the Field type to EmbeddedCurvePoint. Thus, we had to find some way to convert the commitments directly to curve points in order to pass the information successfully to the verifier onchain.

Followup Milestones

The focal agenda of the following milestones strongly revolves around protocol resiliency, privacy and decentralization. The first milestone focussed on a rather simplistic and relaxed model of security, paving the technological way for a more complex and robust technical implementation for the second milestone.

Deliverable 1

Problem Statement and Research propose a new cryptographic primitives that suit the needs of the protocol

Deliverable 2

Commitment Scheme Implementation Implementation of the new candidate to the commitment scheme variants that was proposed in the research phase

Deliverable 3

Off Chain Merkle Tree Implementation of a server that stores the voting commitments off chain and leverages blockchain for integrity by posting Merkle Tree root on chain.

Open Suggestions for Enhancements and Future Integrations

1. Implement verifier on chain with Ink!
2. Contribution to the governing pallet in Github using ink!
3. Store the hash of a Merkle root on chain while maintaining the full Merkle Tree off chain for efficient and lightweight gas usage and data integrity
4. Change commitment scheme to KZG commitment scheme to introduce efficient and privacy preserving solution
5. Replace the centralize server with MPC to eliminate single point of centralization
6. Integration HyperBridge
7. Weight based model
8. Automatic treasury disbursement