User privacy in DAOs

DAO analysis

Ignas Apšega

3620557

Version Control

Version	Date of change	Change description
0.1	16-03-2023	Initial Draft
0.2	23-03-2023	Technology & Implementation
0.3	26-03-2023	Added diagrams and more code snippets of how the Semaphore protocol was implemented
0.4	27-03-2023	Added the section about The Graph, Front-end and folder structure. Added Advise and Conclusion

Table of Contents

byont

Fontys
University of Applied Sciences

1. Abstract

This document offers a comprehensive analysis of the Zero-Knowledge application (zkApp)[1], focusing on its fundamental features and operational principles. It delves into the application's inner workings, potential enhancements, and the advantages and disadvantages of different technical stack choices. The objective is to provide readers with the essential information needed to make well-informed decisions regarding the development and optimization of zkApp, with the ultimate goal of fostering a more robust and efficient implementation.

2. Introduction

In the rapidly evolving landscape of decentralized autonomous organizations (DAOs), maintaining user privacy while ensuring transparency remains a critical challenge. This technical document extends previous investigations into the complexities of balancing these two aspects within the DAO framework, with a focus on the use of smart contracts across various blockchain networks. One potential solution to address this challenge is the implementation of an anonymous voting application for DAO users, as demonstrated in this document.

Using the Byonts WEB3 template as a foundation, this document explores the development of a privacy-centric voting application for DAO users, with the goal of enhancing user privacy while preserving the transparency that is inherent in DAOs. The scope of this document includes the introduction of tooling and technologies based on blockchain that can potentially improve user privacy within DAOs.

In addition to examining the current state of privacy in DAOs and presenting the anonymous voting application, this document discusses the possible limitations and assumptions associated with such applications. By offering valuable insights into privacy-centric tools and practices, this document aims to contribute to the future development of more secure and private solutions for decentralized organizations.

3. Requirements

To fully comprehend and follow the content presented in this document, it is recommended that the reader has prior knowledge of the relevant materials listed below. These documents should ideally be read in the following order to ensure a thorough understanding of the context and concepts discussed:

1. Project Plan
2. DAO Analysis v2
3. General Research on Zero-Knowledge Proof
4. Blog 1: Zero Knowledge Proof - How it Works
5. zk-SNARKs Libraries: Circom and SnarkJS Implementation
6. Architecture Design
7. Integration and Management in Byonts WEB3 Template
8. Semaphore Protocol

Familiarity with these preceding documents will enable the reader to better grasp the ideas and concepts presented in this technical document, allowing for a more comprehensive understanding of the anonymous voting application and its potential impact on user privacy within DAOs.

Gitlab link:

https://git.fhict.nl/I412887/poc-zk-snarks-voting-template

4. Technology & Implementation

Semaphore Protocol[2] was used as main technology for implementation. Because as the previous document “Semaphore Protocol” stated it meets the criteria set for this research, including having an explicit use case for private user voting, compatibility with Byont's tech stack (JavaScript/TypeScript and Solidity). Furthermore, the Semaphore Protocol's simplicity and generic nature make it ideal for integration into Byont's WEB3 TemplateTo recall how it works - with Semaphore, you can allow your users to do the following:

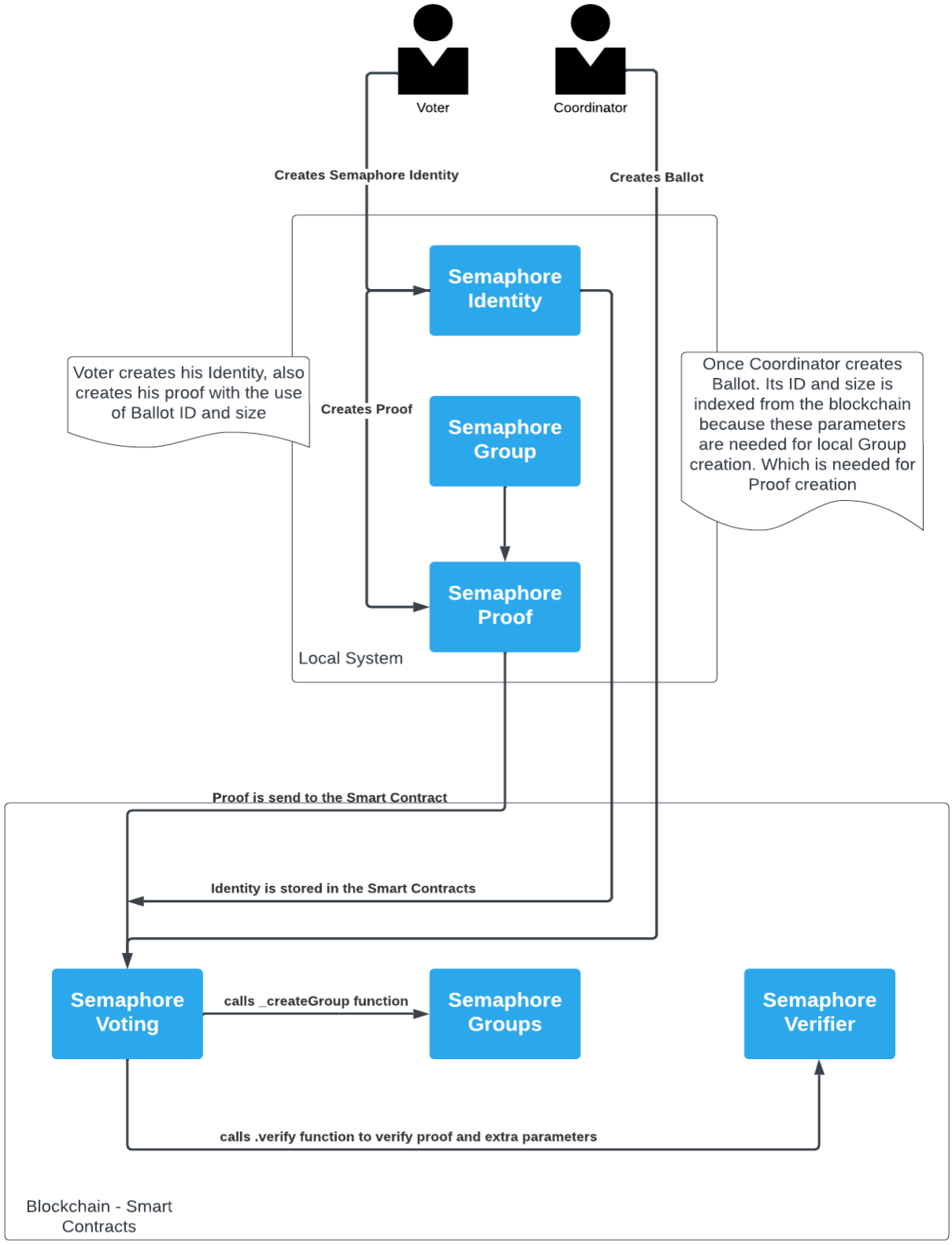
1. Create a Semaphore identity[3].
2. Add their Semaphore identity to a group (i.e. *Merkle tree*)[4].
3. Send a verifiable, anonymous signal (e.g a vote or endorsement).

Semaphore uses on-chain Solidity contracts and off-chain JavaScript libraries:

- Off chain, JavaScript libraries can be used to create identities, manage groups and generate proofs.
- On chain, Solidity contracts can be used to manage groups and verify proofs.

To expand on how the system works - Users will create Semaphore identities by generating identity commitments, which are stored on-chain and used to verify users' identities in various smart contracts. The protocol also includes support for group creation and management, with group memberships controlled by identity commitments[5]. Zero-knowledge proofs are used to verify users' votes in the SemaphoreVoting smart contract, ensuring that only valid votes are counted without revealing users' identities or choices. Overall, the Semaphore protocol combines advanced cryptography and smart contracts to provide a powerful and privacy-focused platform for decentralized applications.

Flow of Off-chain On-Chain



byont

Fontys
University of Applied Sciences

Ballot vs Group

To remove some possible confusion here is the difference between a ballot and a group

A Ballot/poll is a voting process managed by the SemaphoreVoting smart contract. It allows users to cast votes on a specific proposal, with the results being tallied by the smart contract. Polls can have different configurations, such as the duration of the voting period and the type of proposal being voted on.

A group, on the other hand, is a collection of Semaphore identities managed by the SemaphoreGroups smart contract. It allows users to organize identities into groups for various purposes, such as limiting access to certain resources or organizing voting among specific groups.

While polls and groups can be related, they serve different purposes. Polls are used to manage the voting process, while groups are used to manage collections of identities. It's possible to limit voting in a poll to specific groups by requiring that voters have the identity commitments associated with the group, but this is a matter of using groups to manage access to the poll, rather than a fundamental relationship between the two concept.

In the next page, we will dug deeper on what is exactly off-chain and on-chain.

Off-chain

Currently, only 3 actions are happening off-chain and only because it is impossible to do it otherway. These actions are:

Identity creation

First step is to create an Identity using **@semaphore-protocol/identity**

```
const createIdentity = async () => {
    const _identity = new Identity();
    setIdentity(_identity);
};
```

In order to join a Semaphore group and to be able to vote, a user must first create a identity. Identity is also needed for proof creation.

Group creation

The group management is mainly done on-chain. Such as adding an user. But to be able to make a proof for voting. Group has to be created off-chain as well. Use **@semaphore-procotol/group**

```
const createNewGroup = async () => {
    // Takes ID and merkle tree depth integers to create a group
    const newGroup = new Group(parseInt(pollId),
parseInt(merkleTreeDepth));
    setGroup(newGroup);

    return newGroup;
};
```

Proof creation

Use the **`@semaphore-protocol/proof`** library to generate an off-chain proof.
In the voting system use case, once the voter has joined his identity to the ballot group, a voter can generate a proof to vote for a proposal. In the call to **`generateProof`**, the user passes his Identity, Group object (group joined), its ID and vote option. The following code sample shows how to use **`generateProof`** to generate the voting proof:

```
const makeVoteProof = async (newGroup) => {
        // Adds identity to the group. To be able to create a proof
        newGroup.addMember(identity.commitment);

        const proof = await generateProof(
          Identity,  // Semaphore Identity of a user
          newGroup,   // Ballot group a user belongs to
          pollId, // Ballot group ID
          selectedOption // User's vote
    );

    setFullProof(proof);
    return proof;
};
```

On-chain

The main magic is happening on-chain. There 3 smart contracts interacting with each other for the group management, ballot creation and verification. These contracts come from Semaphore Protocol but also were adjusted for our case:

- **SemaphoreVoting** - responsible for creating, starting and ending Ballots. Also, adding a user to the Ballot, casting a vote.
- **SemaphoreGroups** - responsible for creating a Ballot group, adding a user, has utility functions to return info about groups (ID, merkleTreeDepth and etc).
- **SemaphoreVerifier** - verifiers users proof.

Smart Contract - SemaphoreVoting

One of the main functions in the contract is to create a Ballot. The function below showcases how a Ballot/Poll is created which also calls **`_createGroup`** function from **SemaphoreGroups**:

```
function createPoll(
        uint256 pollId, // ID of a Ballot/Voting Poll
        address coordinator, // Wallet address of the poll maker
        uint256 merkleTreeDepth, // How many users can join to vote
        string memory title, // Title of the Poll
        string memory description, // Description of the Poll
        string[] memory votingOptions // Voting options of the Poll e.g Trump or Biden
) public override {
        if (merkleTreeDepth < 16 || merkleTreeDepth > 32) {
        revert Semaphore__MerkleTreeDepthIsNotSupported();
}
        _createGroup(pollId, merkleTreeDepth);

        polls[pollId].coordinator = coordinator;
        polls[pollId].title = title;
        polls[pollId].description = description;
        polls[pollId].votingOptions = votingOptions;

  emit PollCreated(
        pollId,
        coordinator,
        title,
        description,
        merkleTreeDepth,
        votingOptions
        );
```

Besides creating a poll the smart contract has more functions such as:

- **`startPool`** - starts a Ballot/Voting Pool and only then users can vote. This was done so coordinator could be sure about his Ballot before letting anyone to vote.

- **addVoter** - user uses this to join a Ballot/Voting Pool.
- **endPool** - changes Pool state to ended. Thus no one can join and vote anymore.
- **castVote** - is used for an user to cast his vote.

```
  function castVote(
      uint256 vote, // Users Vote
      uint256 nullifierHash, // used to prove that user is not double
voting also.
      uint256 pollId, // Ballot/Voting Poll id
      uint256[8] calldata proof, // Generated proof
      uint256 merkleTreeRoot // Merkle Tree Root of the Ballot users group

  ) public override {
    if (polls[pollId].state != PollState.Ongoing) {
      revert Semaphore__PollIsNotOngoing();
    }

    if (polls[pollId].nullifierHashes[nullifierHash]) {
      revert Semaphore__YouAreUsingTheSameNillifierTwice();
    }

    uint256 merkleTreeDepth = getMerkleTreeDepth(pollId);

    verifier.verifyProof(
      merkleTreeRoot,
      nullifierHash,
      Vote,
      pollId,
      Proof,
      merkleTreeDepth
   );
    polls[pollId].nullifierHashes[nullifierHash] = true;
    emit VoteAdded(pollId, vote, merkleTreeRoot, merkleTreeDepth);
```

Proof vs Vote

There might be confusion between thus, to make it more explicit here are the properties of proof and vote:

Proof:

- Zero-knowledge proof used to ensure vote validity without revealing voter identity
- Proof doesn't contain specific vote information, only verifies voter eligibility and uniqueness of vote

Vote:

- Required parameter in contract to track specific choice made by voter
- Emitted in VoteAdded event to record and tally votes off-chain
- Necessary to record the actual choice made by the voter

Smart Contract - SemaphoreGroup

This smart contract is used to manage Semaphore Identities on the blockchain. A Semaphore group is an [incremental Merkle tree](), and group members are tree leaves. Semaphore groups set the following two parameters:

- **Group id**: a unique identifier for the group
- **Tree depth**: the maximum number of members a group can contain

```
  function _createGroup(
      uint256 groupId,
      uint256 merkleTreeDepth

  ) internal virtual {
      if (getMerkleTreeDepth(groupId) != 0) {
      revert Semaphore__GroupAlreadyExists();
  }
// The zeroValue is an implicit member of the group, or an implicit leaf
of the Merkle tree.
// Although there is a remote possibility that the preimage of
// the hash may be calculated, using this value we aim to minimize the
risk.
    uint256 zeroValue = uint256(keccak256(abi.encodePacked(groupId))) >>
8;
    merkleTrees[groupId].init(merkleTreeDepth, zeroValue);

    emit GroupCreated(groupId, merkleTreeDepth, zeroValue);
  }
```

Besides creating groups. The **SemaphoreGroup** smart contract is responsible for functions such as:

- **`_addMember`**- adds a member to the group
- **`_updateMember`** - updates users identity with a new identity
  Helper functions such as:
- **`getMerkleTreeRoot`** - returns the last root hash of a group
- **`getMerkleTreeDepth`** - returns the depth of the tree of a group

- **`getNumberOfMerkleTreeLeaves`** - returns the number of tree leaves

Smart Contract - SemaphoreVerifier

Finnaly, **SemaphoreVerifier** is used to verify users Semaphore Identity and the inputs of the vote and ballot.

Firstly, the verify function is called in previously discussed and showcased SemaphoreVoting **`.castVote`** function:

```
verifier.verifyProof(
    merkleTreeRoot, // Merkle Tree Root of the Ballot users group
    nullifierHash, // used to prove that user is not double voting also.
    Vote, // Users Vote
    pollId, // Ballot/Voting Poll id
    Proof, // Generated proof
    merkleTreeDepth // The depth of the tree of a group
 );
```

And here is the function itself in **SemaphoreVerifier**:

```
function verifyProof(
  uint256 merkleTreeRoot,
  uint256 nullifierHash,
  uint256 signal,
  uint256 externalNullifier,
  uint256[8] calldata proof,
  uint256 merkleTreeDepth

) external view override {
  signal = _hash(signal);
  externalNullifier = _hash(externalNullifier);

  Proof memory p;

  p.A = Pairing.G1Point(proof[0], proof[1]);
  p.B = Pairing.G2Point([proof[2], proof[3]], [proof[4], proof[5]]);
  p.C = Pairing.G1Point(proof[6], proof[7]);
```

```
    VerificationKey memory vk = _getVerificationKey(merkleTreeDepth - 16);
    Pairing.G1Point memory vk_x = vk.IC[0];
    vk_x = Pairing.addition(vk_x, Pairing.scalar_mul(vk.IC[1],
merkleTreeRoot));
    vk_x = Pairing.addition(vk_x, Pairing.scalar_mul(vk.IC[2],
nullifierHash));
    vk_x = Pairing.addition(vk_x, Pairing.scalar_mul(vk.IC[3], signal));
    vk_x = Pairing.addition(
    vk_x,
    Pairing.scalar_mul(vk.IC[4], externalNullifier)
    );

    Pairing.G1Point[] memory p1 = new Pairing.G1Point[](4);
    Pairing.G2Point[] memory p2 = new Pairing.G2Point[](4);

    p1[0] = Pairing.negate(p.A);
    p2[0] = p.B;
    p1[1] = vk.alfa1;
    p2[1] = vk.beta2;
    p1[2] = vk_x;
    p2[2] = vk.gamma2;
    p1[3] = p.C;
    p2[3] = vk.delta2;

    Pairing.pairingCheck(p1, p2);
  }
```

The **Pairing.G1Point** and **Pairing.G2Point** types and are part of elliptic curve cryptography and zero-knowledge proofs.

In elliptic curve cryptography, cryptographic operations are performed on points on an elliptic curve over a finite field. The G1 and G2 groups are two specific groups of points on the elliptic curve, and they are used in different ways in the Semaphore protocol.

In zero-knowledge proofs, points on the elliptic curve are used to represent various values and secrets being proven, and these points are manipulated and combined to create the zero-knowledge proof.

Blockchain indexing - The Graph

There are several solutions to index and listen to the events happening on the smart-contracts/blockchain.

1. **Infura**: Infura provides Ethereum Virtual Machine (EVM) compatible blockchains. This includes networks like Ethereum, Binance Smart Chain, and Polygon. Infura also supports IPFS, a decentralized file storage system. So, if you are developing on any of these networks, Infura can be a suitable solution for accessing blockchain data and events.
2. **The Graph**: The Graph is an indexing and querying protocol for blockchain data. It allows developers to build decentralized applications that can quickly and easily access blockchain data and events.
3. **Blockchain explorers**: These are web-based tools that provide APIs and also allow users to navigate the choosen blockchain and inspect individual transactions, blocks, and addresses. Some popular EVM-based block explorers include:
    a. BscScan: A block explorer for Binance Smart Chain that provides access to blockchain data and events.
    b. Polygonscan: A block explorer for Polygon (previously Matic) that provides access to blockchain data and events.
    c. Explorer: A block explorer for Fantom that provides access to blockchain data and events.
    d. And any other EVM based blockchain
4. **Web3.js**: Web3.js is a popular JavaScript library for interacting with the Ethereum blockchain. It provides a wide range of functionalities for interacting with smart contracts, fetching blockchain data, and listening to events.

However, **The Graph** was chosen for indexing smart contract events and showcasing blockchain data because:

- It's a widely used technology with a robust developer community and many existing subgraphs to build off of.
- **The Graph** simplifies the process of querying blockchain data, making it much easier to build dApps and services on top of Ethereum.
- **The Graph's** decentralized architecture ensures that data remains publicly accessible, which is critical for building trustless applications.
- By using **The Graph**, I can easily query data across multiple smart contracts and

integrate this data into my application in a way that is scalable and efficient.

It has 3 properties or files:

- **Schema** defines the types and fields of data that the subgraph will query and return.
- **Mapping** specifies how data from the blockchain will be indexed and transformed into the schema.
- **Subgraph manifest** is a configuration file that ties together the schema, mapping, and other metadata to deploy and serve the subgraph.

```
type Poll @entity {
  id: ID! # The unique identifier for the poll
  title: String! # The title of the poll
  description: String! # The description of the poll
  merkleTreeDepth: String! # The depth of the Merkle Tree used in the poll
  votingOptions: [String!]! # An array of all possible voting options in
the poll
  voteCounts: [VoteCount!]! # An array of VoteCount objects to store the
count of each voting option
  blockNumber: BigInt! # The block number when the poll was created
  blockTimestamp: BigInt! # The timestamp when the poll was created
  transactionHash: Bytes! # The transaction hash of the Ethereum
transaction that created the poll
```

The Poll and VoteCount entities were created to store and query the necessary data for displaying the relevant information on the frontend.

```
type VoteCount @entity {
  id: ID! # The unique identifier for the vote count
  poll: Poll! # A reference to the poll that this vote count belongs to
  option: String! # The voting option that this vote count is counting
  count: BigInt! # The count of the number of votes for this voting option
}
```

Below it is showcased how the indexed data was transformed.

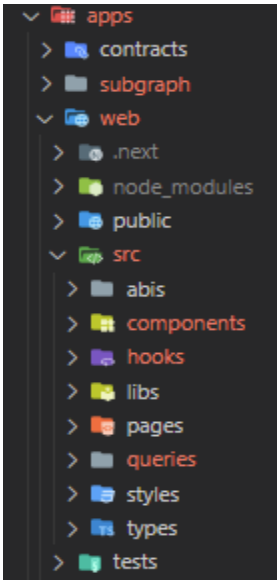
```
export function handlePollCreated(event: PollCreatedEvent): void {
  let poll = new Poll(event.params.pollId.toString());

  poll.title = event.params.title;
  poll.description = event.params.description;
  poll.merkleTreeDepth = event.params.merkleTreeDepth.toString();
  poll.votingOptions = event.params.votingOptions.map<String>(
    (option) => option as String
  );

  poll.blockNumber = event.block.number;
  poll.blockTimestamp = event.block.timestamp;
  poll.transactionHash = event.transaction.hash;

  poll.save();
}
```

Front-end

- The front-end of the application is built on top of the WEB3 Template from Byont, using NextJS and TypeScript.
- The Chakra UI library was used for the front-end components, as it is a widely used library in Web3 and React applications. Particularly, Ethereum foundation used for their webapp.
- The application uses Wagmi hooks for reading and writing to the blockchain. This choice was made because Wagmi hooks are React hooks, which integrate well with the project.
- The RainbowKit wallet provider was used to provide a seamless experience for the user to interact with the choosen blockchain

Folder structure

This project follows simple project folder structure. The project's root directory contains a number of different directories, including an "apps" directory which holds everything related to the application itself.



apps has subdirectories: "contracts", "subgraph", and "web"
contracts - everything related to the smart contracts
subgraph - everything related to indexing blockchain data with The Graph, including subgraph manifests, schema files, and mapping functions.
web - everything related to the front-end of the application, including Next.js and React components, TypeScript code, and various utility functions.
web/src subdirectory is where most of the front-end code lives, and contains several subdirectories of its own:
abis contains the ABI files for each of the smart contracts used by the application.
components - React components that are used to render the user interface of the application.
hooks - custom hooks built on top of the wagmi library for interacting with the blockchain.

pages - Next.js pages that correspond to different routes within the application, including the "voter", "coordinator", and "index" pages.

queries - GraphQL queries that are used to fetch data from The Graph, which is then

used to render various parts of the application's UI.

Below is showcased one of the WAGMI React hooks. This hook connects to the blockchains and adds Voters Semaphore Identity to the particular Ballot

```tsx
import { useState } from 'react';
import { usePrepareContractWrite } from 'wagmi';
import { SemaphoreVotingAbi } from '../abis/SemaphoreVoting';
import { useSigner } from 'wagmi';
import { ethers } from 'ethers';

// This hook returns joinBallot function which can be called to join a ballot,
// and loading and error state variables for the joinBallot function.
export const useJoinBallot = (pollId: string, identityCommitment: string) => {
  // State variables for loading and error
  const [loading, setLoading] = useState(false);
  const [hookError, setHookError] = useState(null);

  // Get signer from Wagmi hook
  const { data: signer } = useSigner();

  // Prepare contract call configuration using Wagmi hook usePrepareContractWrite
  const { config, error } = usePrepareContractWrite({
    address: '0x84c403687c0811899A97d358FDd6Ce7012B1e6C0', // Smart contract address
    abi: SemaphoreVotingAbi, // Smart contract ABI
    functionName: 'addVoter', // Smart contract function name
    args: [pollId, identityCommitment], // Arguments for the smart contract function
  });

  // Join ballot function which will call smart contract function to add the voter
  const joinBallot = async () => {
    if (!signer || !config) {
      return null;
    }

    // Set loading to true and reset error
    setLoading(true);
    setHookError(null);

    try {
      // Create ethers.js contract instance
      const contract = new ethers.Contract(config.address, config.abi, signer);
      // Call smart contract function using the function name and arguments from the config object
      const transaction = await contract[config.functionName](...config.args);
      // Wait for the transaction to be confirmed on the blockchain
      await transaction.wait();
      // Set loading to false
      setLoading(false);
    } catch (error) {
      console.error('Error in joinBallot:', error);
      setLoading(false);
      // Set error state
      setHookError(error);
    }
  };

  // Return joinBallot function and loading and error state variables
  return {
    joinBallot,
    loading,
    error: hookError,
  };
};
```

Advise

General Guidelines building with Semaphore Protocol

1. Familiarize yourself with the Semaphore protocol documentation and whitepaper to gain an understanding of how the protocol works and its features.
2. Determine the use case for which Semaphore will be implemented, such as voting, identity verification, or data privacy.
3. Develop and deploy smart contracts that implement the Semaphore protocol on the blockchain. Also, edit them if needed according to your use-case.
4. Index the smart contract events using a subgraph on The Graph, which will allow you to efficiently query and access the data.
5. Implement the front-end application using a modern framework like Next.js and React, and use custom hooks like wagmi to interact with the smart contracts and blockchain.
6. Use Chakra UI to build reusable UI components, which is widely used in web3 and React apps.
7. Integrate with a Web3 wallet provider like RainbowKit to provide a seamless user experience for signing transactions and interacting with the blockchain.
8. Test the application thoroughly to ensure that it is functioning as expected and that the Semaphore protocol is being implemented correctly.
9. Once the application is live, continue to monitor and maintain it to ensure that it remains secure and up-to-date with the latest best practices and security standards.

Tips for working with Wagmi hooks:

- Start by familiarizing yourself with the Wagmi library and its documentation to understand how to use the provided hooks for interacting with smart contracts on the blockchain.
- Use custom hooks to abstract away complex logic and make your code more modular and reusable.
- Be mindful of the limitations of the blockchain, such as gas costs and transaction times, and design your application accordingly.

Tips for working with The Graph:

- Determine the use case for which The Graph will be implemented, such as voting, token transfers, or identity verification. This will help you design the schema and mapping functions needed to index the data correctly.
- Use the schema to define the types and fields of data that the subgraph will query and return. This is important for ensuring that the data is structured in a way that makes sense for your application.
- Use the mapping functions to specify how data from the blockchain will be indexed and transformed into the schema. This is where you will define the logic for filtering, aggregating, and indexing the data.
- Create a subgraph manifest, which is a configuration file that ties together the schema, mapping, and other metadata to deploy and serve the subgraph.
- Test the subgraph thoroughly to ensure that it is correctly indexing and querying the data. This is especially important if you are using The Graph to power a live application.
- Take advantage of the decentralized architecture of The Graph to ensure that data remains publicly accessible and censorship-resistant. This is critical for building truly trustless applications.
- Use The Graph to efficiently query data across multiple smart contracts and integrate this data into your application in a way that is scalable and efficient. This will help ensure that your application can handle large volumes of data and remain responsive to user requests.(e.g have queries folder and a Typescript file per query)

Conclusion

The Semaphore Protocol voting system is a well-designed and robust solution that leverages various technologies to ensure secure and private voting. Through the use of zero-knowledge proofs, Semaphore Identities, and the blockchain, the system provides strong guarantees of voter eligibility and vote uniqueness, while also maintaining voter anonymity.

The system uses smart contracts to manage voting, groups, and verifications. Off-chain proofs are generated using the Semaphore protocol to ensure voter eligibility and vote uniqueness, while actual anonymous voting data is stored on-chain as events that can be queried using The Graph.

The system includes two types of users: coordinators and voters.

- **Coordinators** - responsible for creating polls, managing the voting process, and verifying the results.
- **Voters** - can join a poll group and cast their vote using an off-chain proof.

The Semaphore protocol is used to generate proofs that prove voter eligibility and uniqueness without revealing the voter's identity. The smart contracts used in the system include:

SemaphoreVoting - is responsible for creating, starting, and ending polls, as well as adding users to the poll and casting votes

SemaphoreGroups - is used to manage Semaphore identities on the blockchain, including creating groups and adding and updating group members. SemaphoreVerifier

SemaphoreVerifier - is used to verify users' identities and inputs to ensure the integrity of the voting process.

The system's front-end is built on top of the WEB3 Template from Byont, using NextJS and TypeScript. Chakra UI is used for front-end components, and the application uses Wagmi hooks for reading and writing to the blockchain. The RainbowKit wallet provider is used to provide a seamless experience for users to interact with the chosen blockchain. In conclusion, this voting system is a well-designed and secure implementation of a decentralized application built on the blockchain.

Citation and References

1. “AnonyVote: A DAO Tooling Platform for Anonymous Voting With ZK.” *Harmony forum*, 11 March 2022, https://talk.harmony.one/t/anonyvote-a-dao-tooling-platform-for-anonymous-voting-with-zk/13810. Accessed 28 October 2022.
2. “Blockchain Facts: What Is It, How It Works, and How It Can Be Used.” *Investopedia*, https://www.investopedia.com/terms/b/blockchain.asp. Accessed 26 October 2022.
3. Book, Adrien. “Everything Wrong With DAOs |.” *cult by Honeypot*, 5 July 2022, https://cult.honeypot.io/reads/everything-wrong-with-daos/. Accessed 27 October 2022.
4. “Collab.Land.” *Welcome to Collab.Land*, https://collab.land/. Accessed 26 October 2022.
5. “DAO Nation — Clay.” *Clay*, 16 August 2021, https://clay.mirror.xyz/DwJ60O0R1IyRiPAZFBw4L05L3fd8PPxWnzDNedKtOas. Accessed 27 October 2022.
6. “Ethereum.” *Wikipedia*, https://en.wikipedia.org/wiki/Ethereum. Accessed 26 October 2022.
7. Frankenfield, Jake. “What Are Crypto Tokens, and How Do They Work?” *Investopedia*, 20 May 2022, https://www.investopedia.com/terms/c/crypto-token.asp. Accessed 26 10 2022.
8. “Introduction to dapps | ethereum.org.” *Ethereum.org*, https://ethereum.org/en/developers/docs/dapps/. Accessed 28 October 2022.

9. “On Chain Transactions (Cryptocurrency) Definition.” *Investopedia*, https://www.investopedia.com/terms/c/chain-transactions-cryptocurrency.asp. Accessed 26 October 2022.
10. Ouaddah, Aafaf. “Arithmetic Circuit.” *ScienceDirect*, 2019, https://www.sciencedirect.com/topics/engineering/arithmetic-circuit. Accessed 25 10 2022.
11. “Testnet.” *Wikipedia*, https://en.wikipedia.org/wiki/Testnet. Accessed 28 October 2022.
12. “Uniswap.” *Wikipedia*, https://en.wikipedia.org/wiki/Uniswap. Accessed 26 October 2022.
13. “Web3.” *Wikipedia*, https://en.wikipedia.org/wiki/Web3. Accessed 27 October 2022.
14. “Welcome to Snapshot!” *Home - snapshot*, https://docs.snapshot.org/. Accessed 26 October 2022.
15. “What are smart contracts on blockchain?” *IBM*, https://www.ibm.com/topics/smart-contracts. Accessed 26 October 2022.
16. “What is SushiSwap? (SUSHI).” *Kraken*, https://www.kraken.com/learn/what-is-sushiswap-sushi. Accessed 26 October 2022.
17. “Zero-knowledge proof.” *Wikipedia*, https://en.wikipedia.org/wiki/Zero-knowledge_proof. Accessed 27 October 2022.
18. “ZKU-Vote: Anonymous voting within DAO - zkDAO - Harmony Community Forum.” *Harmony forum*, 12 May 2022, https://talk.harmony.one/t/zku-vote-anonymous-voting-within-dao/18423. Accessed 28 October 2022.

Appendix

Term	Explanation
1. Zero-Knowledge Application (zkApp)	zkApps are a new type of decentralized application (dapp) that use cryptography to conduct trustless blockchain transactions. Using the power of zk-SNARKs, or zero knowledge Succinct Non-interactive Arguments of Knowledge, zkApps can prove knowledge without needing to expose all pieces of information. (“What are zkApps?”)
2. Semaphore Protocol	Using zero knowledge, Semaphore allows Ethereum users to prove their membership of a group and send signals such as votes or endorsements without revealing their original identity.
3. Semaphore Identity	In order to join a Semaphore group, a user must first create a Semaphore identity.
4. Merkle Tree	A hash tree, also known as a Merkle tree, is a tree in which each leaf node is labeled with the cryptographic hash of a data block, and each non-leaf node is labeled with the cryptographic hash of its child nodes' labels.
5. Identity commitment	Is **a public value used in Semaphore groups to represent the identity of a group member**. The secret values are similar to Ethereum private keys and are used to generate Semaphore zero-knowledge proofs and authenticate signals. (“Identities | Semaphore”)

6. Ethereum Virtual Machine (EVM)	The **Ethereum Virtual Machine** (EVM) is the computation engine for Ethereum that manages the state of the blockchain and enables smart contract functionality. The EVM is contained within the client software (e.g., Geth, Nethermind, and more) that you need in order to run a node on Ethereum (Kochan)