PROJECT REPORT

(Submitted for Mini Project in partial fulfilment of the B.Tech. Degree in Information Technology under the Indian Institute of Engineering Science and Technology)

Title:

Securing Decentralized Voting Applications Against Cyber Attacks Using Proof of Work on Ethereum

---

Submitted by:

Abhinav Gupta *(2023ITB065)*
Diparghya Mallik *(2023ITB066)*
Aniket Gond *(2023ITB068)*

Under the supervision of

Prof. Surajit Kumar Roy
Associate Professor
Department of Information Technology
IIEST, Shibpur

Month & Year of Submission: **May, 2025**

Acknowledgement

We would like to express our profound gratitude to **Prof. Surajit Kumar Roy**, Associate Professor, Information Technology, for his valuable guidance and support throughout the completion of our project titled - **Securing Decentralized Voting Applications Against Cyber Attacks Using Proof of Work on Ethereum**. We would also like to extend our special thanks to **Prof. Tuhina Samanta**, Head of the Information Technology Department, and all the respected professors for their continuous support and encouragement during this semester. Your suggestions and guidance were truly helpful in shaping and completing this project. We are sincerely thankful to the entire department administration.

We also wish to thank our seniors of M. Tech. for their helpful feedback and cooperation, which contributed to the successful completion of this project. We hereby declare that this project was completed solely by us.

To Whom It May Concern

This is to certify that **Abhinav Gupta (2023ITB065)**, **Diparghya Mallik (2023ITB066)**, and **Aniket Gond (2023ITB068)** have successfully completed their mini project titled - **Securing Decentralized Voting Applications Against Cyber Attacks Using Proof of Work on Ethereum** in partial fulfilment of the requirements for the degree of **B.Tech. in Information Technology**.

During this period, they have carried out the project work diligently. The submitted report meets all the necessary requirements as per the institute's regulations and attains the standard expected for academic submission

Content

Chapter 1: Introduction

In an age where digital democracy and trust less systems are in high demand, blockchain-based voting systems offer a resilient alternative to traditional, centralized voting mechanisms. Trust, transparency, and verifiability are the pillars of any electoral process, and blockchain provides a platform where these attributes can be inherently encoded.

This project aims to design, implement, and critically analyze a Decentralized Voting (D-Voting) application on Ethereum using Solidity smart contracts, the Truffle development framework, Ganache GUI for local testing, and MetaMask for user interaction. A unique focus of this project is the inclusion of a simulated Proof of Work (PoW) mechanism and the deliberate introduction of key security vulnerabilities. These vulnerabilities are then exploited in a controlled environment to understand their impact and demonstrate how to implement defenses effectively.

Chapter 2: Literature Survey

Multiple studies and projects have laid the foundation for blockchain-based electoral systems:

- **Follow My Vote**: Advocates blockchain as a medium for secure, transparent, and accessible elections.
- **Voatz** (MIT): Explored the potential of mobile blockchain voting, tested in several state-level elections in the USA.
- **Agora**: Demonstrated real-world blockchain voting during Sierra Leone's elections, though not officially used for tallying.

Challenges identified include:

- Voter authentication and registration.
- Defense against replay and Sybil attacks.
- UI-related vulnerabilities that allow phishing or misdirection.

This project expands upon these insights by performing actual attack simulations on a controlled blockchain system and mitigating the threats post-analysis.

Chapter 3: System Architecture and Attack Simulation Framework

Tools and Technologies:

- **Solidity**: Smart contract programming.
- **Truffle**: Development and deployment.
- **Ganache GUI**: Local Ethereum simulation.
- **Metamask**: Wallet and identity manager.
- **Web3.js**: Bridge between smart contract and frontend.

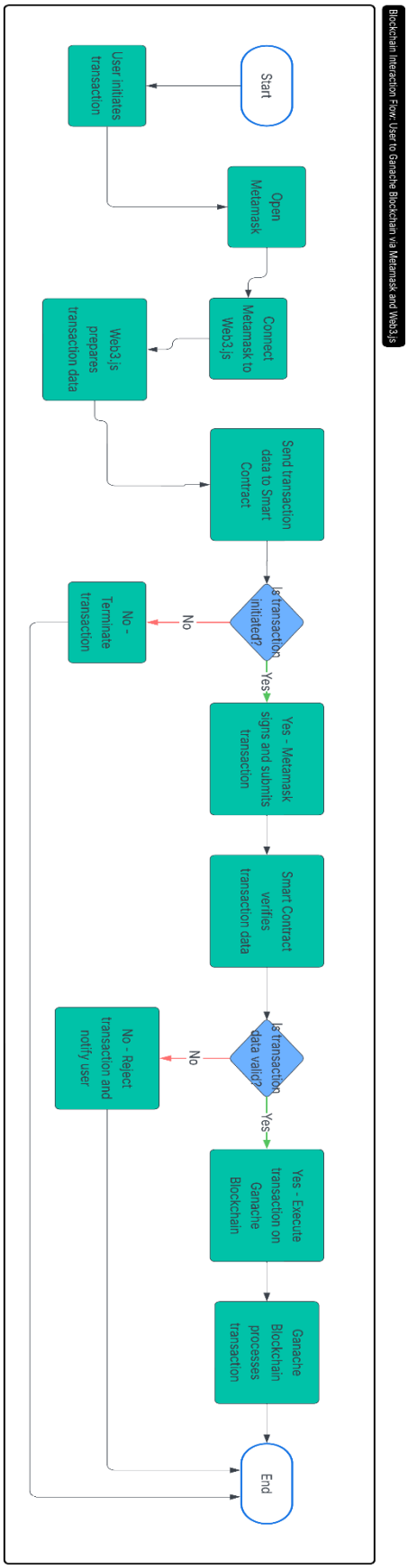
Architecture Explanation:

1. **User**: The participant interacting with the application through the web interface.
2. **MetaMask** : Provides Ethereum wallet capabilities to sign transactions and manage accounts.
3. **Frontend(Web3.js)**: JavaScript library to facilitate interaction with the smart contract. It allows sending transactions, querying blockchain state, and handling events.
4. **Smart Contract (DVoting.sol)**: Deployed on Ganache, it contains business logic for candidate registration, voting, and PoW validation.
5. **Ganache**: A personal blockchain for Ethereum development that emulates a real network.

Each vote follows this flow:

- The user selects a candidate on the frontend.
- The frontend runs a PoW script to generate a valid nonce.

System Architecture:



- The transaction is sent to the DVoting contract via Metamask.
- Ganache mines the transaction and updates the state accordingly.

Chapter 4: Attack Implementations and Mitigation Strategies

1. Double Voting Attack

- **Nature**: An account tries to vote multiple times.
- **Vulnerable Version**:

```
function vote(uint _candidateId, uint nonce) public {
  candidates[_candidateId].voteCount++;
}
```

- **Hardened Version**:

```
mapping(address => bool) public hasVoted;

function vote(uint _candidateId, uint nonce) public {
  require(!hasVoted[msg.sender], "Already voted");
  hasVoted[msg.sender] = true;
  candidates[_candidateId].voteCount++;
}
```

- **Explanation**: This ensures that a user can only vote once by tracking voters with hasVoted.

2. Sybil Attack

- **Nature**: One user votes from multiple fake accounts.
- **Vulnerable Code**:

```
function vote(uint _candidateId, uint nonce) public {
  ...
}
```

- **Possible Prevention**: Only whitelisted addresses are allowed to vote, blocking Sybil attacks.

3. Replay Attack / PoW Bypass

- **Nature**: Reusing a previously successful nonce.
- **Vulnerable Logic**:

```
bytes32 hash = keccak256(abi.encodePacked(msg.sender, _candidateId, nonce));
```

- **Mitigation Using Nonce Tracking**:

```
mapping(bytes32 => bool) public usedNonces;


function vote(uint _candidateId, uint nonce) public {
  bytes32 hash = keccak256(abi.encodePacked(msg.sender, _candidateId, nonce));
  require(!usedNonces[hash], "Nonce already used");
  usedNonces[hash] = true;
  ...
}
```

- **Explanation**: Prevents hash reuse by storing every valid vote hash.

4. Gas Griefing / Out-of-Gas Attack

- **Nature**: Huge nonces sent to waste gas.
- **Mitigation**:

```
require(nonce < 1000000, "Nonce value too large");
```

- **Explanation**: Ensures gas usage remains within reasonable bounds.

5. User Interface Phishing Attack

- **Nature**: Fake frontend tricks users into misvoting.
- **Explanation**: Displaying candidate name and transaction preview ensures user knows exactly what they are approving in Metamask.

Chapter 5: Results and Conclusion

Results:

- All planned attacks were executed.
- Effective mitigations were successfully implemented.
- Frontend interacted smoothly with backend via Web3.
- Users could see the effect of protections like vote-blocking and registration.

Conclusion:

The D-Voting project serves as both a functional decentralized application and a security lab environment. It effectively demonstrates how smart contracts can be vulnerable and how

even simple fixes like flag variables, whitelisting, or PoW hardening can make a system significantly more secure.

References

1. EthereumWhitepaper - https://ethereum.org/en/whitepaper/
2. Solidity Documentation - https://docs.soliditylang.org
3. Truffle Suite Docs - https://trufflesuite.com/docs
4. MetaMask Docs - https://docs.metamask.io
5. Ganache by Truffle - https://trufflesuite.com/ganache/

Appendix: Code Snippets

Smart Contract: DVoting.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
contract DVoting {
  struct Candidate {
    uint id;
    string name;
    uint voteCount;
  }

  mapping(uint => Candidate) public candidates;
```

```
    mapping(address => bool) public hasVoted;
    mapping(bytes32 => bool) public usedNonces;

    uint public candidatesCount;
    uint public difficulty = 3;

    constructor() {
        addCandidate("Alice");
        addCandidate("Bob");
    }

    function addCandidate(string memory _name) internal {
        candidatesCount++;
        candidates[candidatesCount] = Candidate(candidatesCount, _name, 0);
    }

    function vote(uint _candidateId, uint nonce) public {
        require(!hasVoted[msg.sender], "Already voted");
        require(_candidateId > 0 && _candidateId <= candidatesCount, "Invalid candidate");

        bytes32 hash = keccak256(abi.encodePacked(msg.sender, _candidateId, nonce));
```

```
        require(uint256(hash) < type(uint256).max / (10 ** difficulty), "Invalid PoW");
        require(!usedNonces[hash], "Nonce reused");
        hasVoted[msg.sender] = true;
        usedNonces[hash] = true;
        candidates[_candidateId].voteCount++;
    }
}
```

Frontend JS (Simplified)

```
async function voteCandidate(id) {
    const nonce = await findValidNonce(account, id);
    await contract.methods.vote(id, nonce).send({ from: account });
}
async function findValidNonce(account, id, difficulty = 3) {
    let nonce = 0;
    const target = BigInt("0x" + "f".repeat(64)) / BigInt(10 ** difficulty);
    while (true) {
        const hash = web3.utils.soliditySha3({ type: 'address', value: account }, { type: 'uint', value: id }, { type: 'uint', value: nonce });
        if (BigInt(hash) < target) return nonce;
        nonce++;
    }
}
```