GAS COST OPTIMIZATION WITH ZERO KNOWLEDGE PROOFS

Bonafide record of work done by

Devi Meena R **(21Z215)**
K C Praneethaa **(21Z223)**
Preetha Selvarasu **(21Z237)**
Vasudha R B **(21Z267)**
Yohasini U BA **(21Z271)**

19Z701 - CRYPTOGRAPHY

Dissertation submitted in the partial fulfillment of the
requirements for the degree of

BACHELOR OF ENGINEERING
BRANCH: COMPUTER SCIENCE AND ENGINEERING

OCTOBER 2024

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
PSG COLLEGE OF TECHNOLOGY
(Autonomous Institution)
COIMBATORE – 641 004

Table of Contents

1.Abstract 3

2.Introduction 5

3.Requirements Analysis 6

3.1. Stakeholder Identification 6

3.2. Functional Requirements 6

3.3. Non-Functional Requirements 7

3.4. Data Type Specifications 7

4. System Architecture 8

4.1 Architecture Overview 8

4.2 Data Flow 8

4.3 Architecture 11

5. Tools and Technologies 12

6. System Implementation 13

6.1.Proof generation and verification: 14

6.2. Deployment and Integration: 15

7. Performance Evaluation 16

7.1. Proof Generation Time 16

7.2. Proof Verification Time 16

7.3. Gas Consumption 16

7.4. User Experience 17

8. Challenges Faced 20

9. Future Enhancements 21

10. Conclusion 22

11. References : 23

12. Appendix 24

1.Abstract

Zero-Knowledge Proofs (ZKP) are emerging as a powerful cryptographic tool for ensuring data privacy and security in various sectors, including healthcare.
In this project, we integrate zk-SNARKs (Succinct Non-Interactive Arguments of Knowledge) with blockchain technology to develop a secure and efficient system for verifying blood test results within a healthcare consortium.

The system leverages Ethereum smart contracts to validate health data without revealing sensitive patient information, thus maintaining confidentiality. This solution not only improves data integrity but also ensures transparency. By employing ZoKrates for proof generation and Remix IDE for smart contract deployment, we create a framework that enables private, verifiable computations on healthcare data. The report provides a comprehensive analysis of system design, implementation, testing results, and future enhancements, showcasing the practical application of zk-SNARKs in enhancing healthcare data privacy.

2.Introduction

2.1. Problem Statement

In the healthcare industry, data privacy and security are paramount concerns, particularly when handling sensitive information such as blood test results. Traditional methods of sharing data often expose patients' private medical details to unauthorized parties, leaving the system vulnerable to data breaches, unauthorized access, and misuse. Furthermore, with the rise of blockchain technology, transaction costs (gas costs) have become a concern, especially when dealing with complex cryptographic operations.

2.2. Description of the Problem

The process of verifying sensitive data in sectors like health care involves multiple stakeholders, such as individuals, service providers, and third parties. Each party requires access to some level of data for validation, which risks breaches of confidentiality. Current systems often lack strong privacy-preserving mechanisms, leading to the exposure of personal information during verification. Additionally, centralized systems are vulnerable to issues like data manipulation, hacking, and single points of failure, compromising trust and security.

Beyond privacy concerns, the high transaction costs associated with deploying cryptographic proofs, such as zero-knowledge proofs (zk-SNARKs), on blockchain networks also pose a challenge. These costs, particularly gas fees on networks like Ethereum, can increase with complex verification processes, making it necessary to optimize gas consumption for scalable and efficient systems.

2.3. Objectives of the Project

- **Data Privacy:** To develop a system that ensures the confidentiality of sensitive blood test results during verification. This will be done using Zero-Knowledge Proofs (ZKP), allowing data validation without exposing actual test data.

- **Secure Verification:** To integrate zk-SNARKs (Succinct Non-Interactive Arguments of Knowledge) with Ethereum smart contracts to facilitate secure, decentralized verification of blood test results, ensuring that patient data remains private.
- **Gas Cost Optimization:** To create models that optimize transaction gas costs in Ethereum-based healthcare consortia. Explore the use of zero-knowledge proofs (e.g., zk-SNARKs) to reduce computational overhead while preserving privacy.
- **Blockchain Integration:** To implement the solution on the Ethereum blockchain to ensure immutability, transparency, and resistance to tampering, all while keeping transaction costs manageable.
- **User-Friendly System:** To provide a user interface for patients and healthcare providers that is intuitive and secure for inputting health data and generating proofs.

3.Requirements Analysis

3.1. Stakeholder Identification

The key stakeholders for this project are:

- **Patients:** The primary users of the system who need to share their blood test results for validation.
- **Healthcare Providers (Doctors and Laboratories):** Responsible for conducting tests and uploading results. They also validate results for further medical procedures.
- **Insurance Companies:** Require validation of medical records and test results for processing claims without accessing the actual data.
- **Developers:** The technical team responsible for implementing and maintaining the system, ensuring secure, scalable, and efficient operation.

3.2. Functional Requirements

The system must fulfill the following functional requirements:

- **Data Input:** The system must allow users (patients or healthcare providers) to input the data such as blood test results, including glucose and cholesterol levels, via a web interface.
- **Proof Generation:** Once the data is entered, the system should generate a zk-SNARK proof that validates the data without revealing the actual results.
- **Proof Verification:** A smart contract on the Ethereum blockchain must verify the zk-SNARK proof without accessing the underlying data. This process ensures that the blood test results are valid and authentic.
- **Secure Data Storage:** Test results should not be stored in plaintext on the blockchain or any database. Only the zk-SNARK proof is stored and shared for validation.
- **Error Handling:** The system should notify users in case of incorrect input, verification failure, or any other errors.

3.3. Non-Functional Requirements

- **Performance:** The zk-SNARK proof generation and verification must be optimized to ensure that the system remains responsive, even as the number of users increases.
- **Usability:** The system should provide an intuitive interface for all users, ensuring that non-technical users, such as patients, can easily input their data and receive verification without specialized knowledge.
- **Cost Efficiency:** Ethereum smart contracts involve gas fees for execution. The system should be optimized for low gas consumption to reduce operational costs.

3.4. Data Type Specifications

- **Glucose Level (Input):** The glucose level entered by the patient or healthcare provider should be a numeric value (e.g., 85 mg/dL). It should be validated for realistic ranges to avoid errors.
- **Cholesterol Level (Input):** The cholesterol level should also be a numeric value (e.g., 190 mg/dL). Similar to glucose, it should be checked against normal ranges to ensure accuracy.
- **zk-SNARK Proof (Generated):** The proof generated for the blood test results should be a structured, non-interactive cryptographic proof. It will contain necessary metadata for the smart contract to verify it.
- **User Information:** Basic information such as patient ID, doctor's ID, and laboratory IDs will be required for user authentication. This data should be encrypted and stored securely to prevent unauthorized access.

4. System Architecture

4.1 Architecture Overview

The architecture consists of three primary layers:

- **Frontend Layer:** The user interface developed using HTML, CSS, and JavaScript, allowing users to input data and interact with the system.
- **Smart Contract Layer:** Smart contracts written in Solidity to handle the proof verification process.
- **ZKP Layer:** Utilization of ZoKrates to generate and verify zk-SNARK proofs.

4.2 Data Flow

Off-Chain:

1.Data Collection/Processing:

- Input: User (Patient, Client, or IoT device) collects sensitive data (e.g., medical data, financial data, sensor readings).
- Process: Data is recorded privately, ensuring confidentiality and security.
- Output: Raw sensitive data (not shared).

2.ZKP Generation:

- Input: The organization (Healthcare Provider, Financial Institution, etc.) uses the raw sensitive data.
- Process:
  1. The organization utilizes a cryptographic tool (e.g., ZoKrates) to generate a Zero-Knowledge Proof (ZKP).
  2. A program (.zok file) is created that encodes the logic for generating the proof based on the sensitive data.
- The ZKP is generated off-chain (not on the blockchain) on the organization’s server.

- Output:

  .zok file (ZoKrates program).

  Proof file (e.g., JSON) containing the generated proof.

  Verifier.sol file (Solidity smart contract for proof verification).

<u>Backend:</u>

3. **Proof Submission:**
   - Input: The organization submits the proof generated in the previous step.
   - Process: The proof is sent to the deployed smart contract on the blockchain for verification.
   - Output: A request is made to the blockchain for verification.

4. **Result Handling:**
   - Input: The verification result (valid/invalid) is received from the blockchain.
   - Process: The backend processes the result from the smart contract.
   - Output: The verification result is sent to the UI.

<u>Ethereum Blockchain</u>

5.Smart Contract Deployment (On-Chain)

- Input: The organization (Healthcare Provider) deploys the generated Verifier.sol file.
- Process:

  The Verifier.sol contract is deployed onto the blockchain using a development environment (e.g., Remix IDE).
- Output:
  - Deployed Verifier.sol contract on the blockchain.
  - Blockchain transaction confirming the contract deployment.

6.Smart Contract Verification:

- Input: The submitted ZKP proof from the backend.
- Process: The smart contract verifies the ZKP according to the encoded logic.
- Output: A result (valid/invalid) is generated based on the verification.

UI:

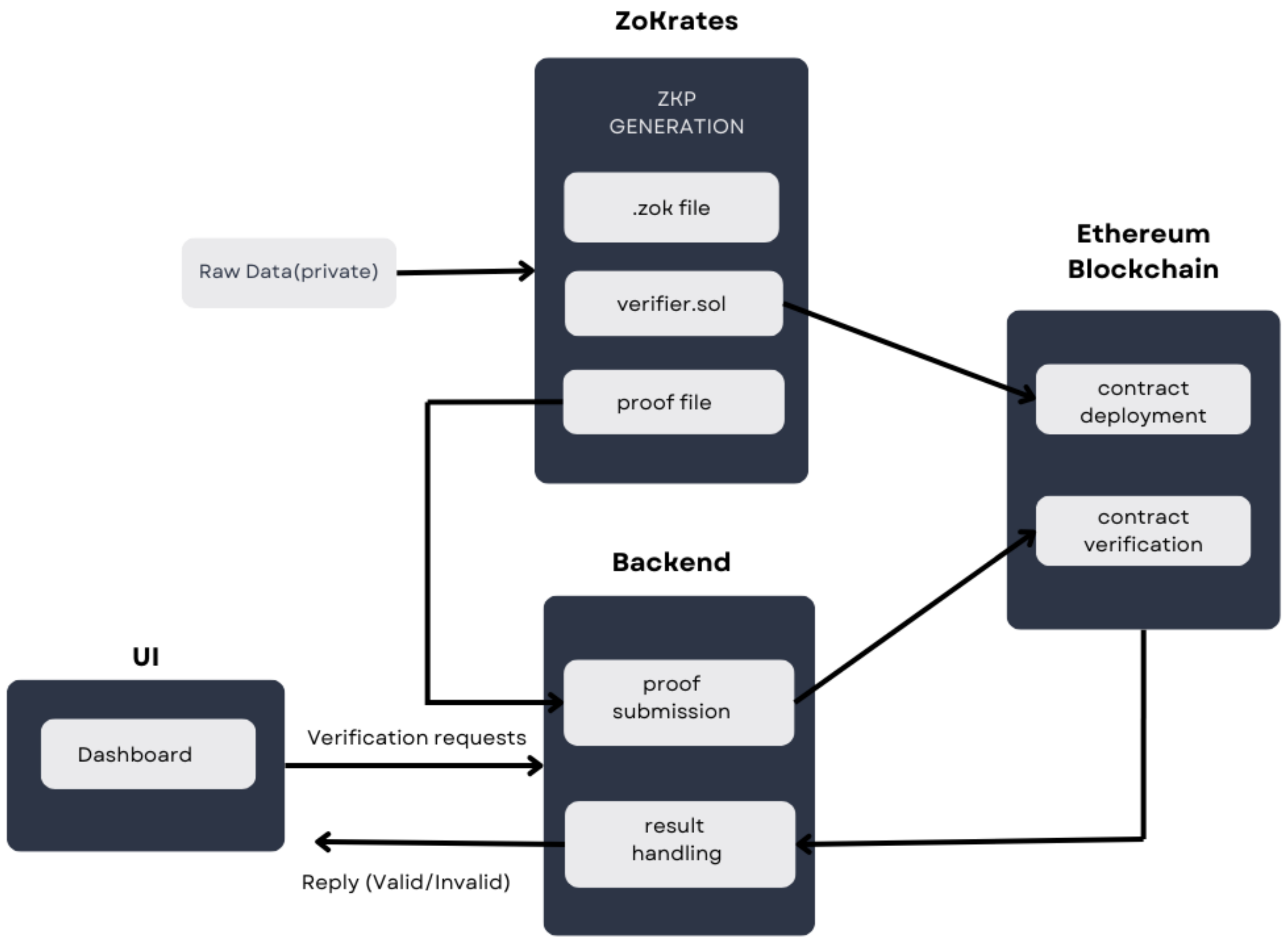
7. Verification Request:

- Input: Users (Doctors, Insurers, or other stakeholders) submit a request for verification through a front-end application.
- Process:
    1. Users enter a proof ID or transaction ID into the front-end application.
    2. The application connects to the backend to initiate verification.
- Output: A request is sent to the backend for verification.

8.Result Display:

- Input: The verification result (valid/invalid) from the backend.
- Process: The front-end application displays the result to the user.
- Output: Display of the verification result on the user interface.

4.3 Architecture:



5. Tools and Technologies

5.1.Remix IDE - Smart Contracts

Remix IDE is a browser-based development environment used for writing, testing, and deploying Solidity smart contracts on the Ethereum blockchain. It simplifies contract deployment by providing built-in support for compiling and interacting with the Ethereum network.

5.2. HTML, CSS, and JavaScript - Frontend Development

HTML, CSS, and JavaScript form the foundation of the web interface. HTML structures the web page, CSS styles it for better user experience, and JavaScript provides the interactivity, allowing users to input data and trigger transactions with the blockchain.

5.3. ZoKrates - Proof Generation

ZoKrates is a zk-SNARK toolkit that allows the creation and verification of zero-knowledge proofs. It is used to write, compile, and generate the proving and verification keys for the blood test verification circuit, ensuring privacy-preserving computations.

5.4. Web3.js - Integration

Web3.js is a JavaScript library that connects the frontend to Ethereum smart contracts. It allows the application to interact with the blockchain, handle proof verification, and trigger functions in the deployed smart contract from the browser.

6. System Implementation

Proof generation and verification:

6.1 Setting Up ZoKrates

ZoKrates is a toolbox for zk-SNARKs that allows developers to write, compile, and deploy zero-knowledge proofs on the blockchain.

Steps:

- Command: curl -LSfs get.zokrat.es | sh
    - This installs ZoKrates on your Ubuntu machine.

6.2 Writing the zk-SNARK Circuit

A circuit defines the logic for zk-SNARKs, such as validating specific conditions (e.g., checking if certain health parameters fall within acceptable ranges).

Steps:

- Write a .zok file that describes the mathematical relationship you want to prove.
- Command to compile: zokrates compile -i filename.zok

6.3 Generating Proving and Verification Keys

After compiling the circuit, proving and verification keys are generated for zk-SNARKs.

Steps:

- Command: zokrates setup
    - This generates:
        - **Proving Key**: For generating proofs.
        - **Verification Key**: For verifying the proofs.

6.4 Computing the Witness

The witness is a set of inputs that satisfy the constraints defined in the circuit.

Steps:

- Command: zokrates compute-witness -a [values]

    - Example: zokrates compute-witness -a 100 150 (e.g., for glucose = 100, cholesterol = 150)

6.5 Generating the Proof

This step generates a zk-SNARK proof confirming that the inputs meet the circuit's conditions.

Steps:

- Command: zokrates generate-proof

6.6 Exporting the Verifier

The verifier is a smart contract deployed on the blockchain to verify zk-SNARK proofs on-chain.

Steps:

- Command: zokrates export-verifier
    - This generates a Solidity contract to verify zk-SNARK proofs.

6.7 Verifying the Proof

Once the proof and verifier contract are deployed, the proof is verified on-chain to confirm the correctness of the zk-SNARK.

Steps:

- Command: zokrates verify

Deployment and Integration:

Deploying the Verifier Using Remix IDE

After generating the verification contract, it can be deployed to the blockchain using Remix IDE.

Steps:

- Open Remix IDE in your browser.
- Paste the Solidity contract (verifier.sol) into the editor.

- Compile and deploy it using MetaMask.

<u>Front-End Integration:</u>

Integrating the zk-SNARK process with a front-end allows users to interact with it through a web interface.

Steps:

- Create a front-end (HTML/CSS/JavaScript) that interacts with the deployed contract via Web3.js or Ethers.js.
- Users input their data, and the app generates zk-SNARK proofs, submits them to the contract, and verifies them on the blockchain.

7. Performance Evaluation

The key metrics used for evaluation include proof generation time, verification time, gas consumption and user experience.

7.1. Proof Generation Time

- **Evaluation**: The time taken by ZoKrates to generate a proof from the blood test data was evaluated. Efficient proof generation ensures minimal delay for the user, but zk-SNARK computations can be resource-intensive.
- **Result**: The proof generation time is affected by the complexity of the circuit and the processing power of the host machine. Optimizing the circuit logic led to acceptable generation times for typical user inputs.

7.2. Proof Verification Time

- **Evaluation**: The verification process performed by the Solidity smart contract was assessed to ensure that the proof can be verified quickly on-chain. Lower verification times improve user experience and reduce on-chain computational costs.
- **Result**: Verification time was found to be minimal since zk-SNARKs allow for constant-time verification, making it suitable for on-chain execution with low overhead.

7.3. Gas Consumption

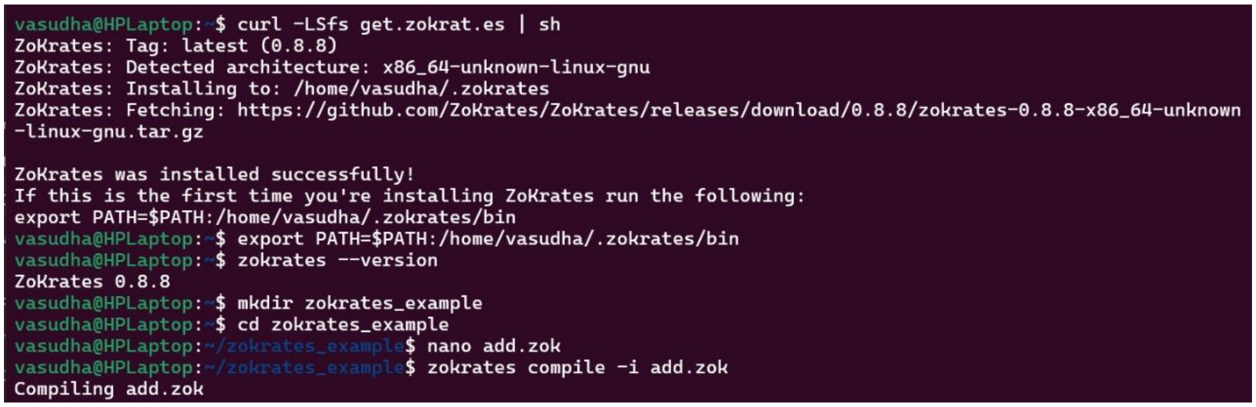
- **Evaluation**: The gas cost of deploying the smart contract and verifying proofs was measured. Smart contracts interacting with zero-knowledge proofs tend to consume more gas, so optimizing the contract logic is critical for reducing costs.
- **Result**: The contract consumed moderate gas for proof verification. However, optimizations in the contract's structure and use of minimal storage helped reduce overall costs to a manageable level.

7.4. User Experience

- **Evaluation**: The frontend's responsiveness and interaction with the smart contract were evaluated. This included form submission times, proof generation, and transaction feedback for the user.
- **Result**: Users experienced minimal delays when submitting forms and verifying proofs, as optimizations in frontend interaction (using Web3.js) ensured seamless blockchain interaction.

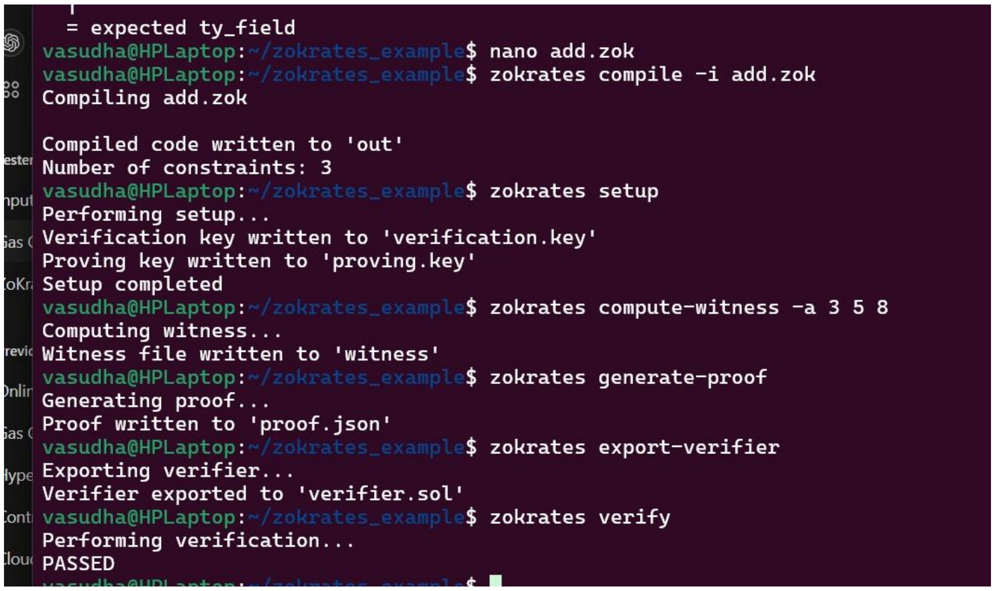
Screenshots:

Zokrates Installation:



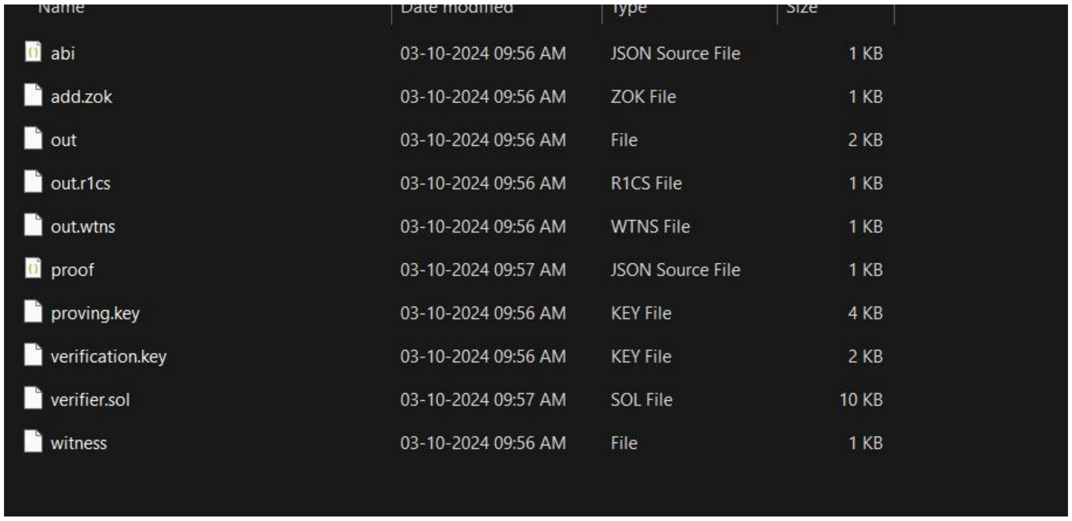
```
vasudha@HPLaptop:~$ curl -LSfs get.zokrat.es | sh
ZoKrates: Tag: latest (0.8.8)
ZoKrates: Detected architecture: x86_64-unknown-linux-gnu
ZoKrates: Installing to: /home/vasudha/.zokrates
ZoKrates: Fetching: https://github.com/ZoKrates/ZoKrates/releases/download/0.8.8/zokrates-0.8.8-x86_64-unknown
-linux-gnu.tar.gz

ZoKrates was installed successfully!
If this is the first time you're installing ZoKrates run the following:
export PATH=$PATH:/home/vasudha/.zokrates/bin
vasudha@HPLaptop:~$ export PATH=$PATH:/home/vasudha/.zokrates/bin
vasudha@HPLaptop:~$ zokrates --version
ZoKrates 0.8.8
vasudha@HPLaptop:~$ mkdir zokrates_example
vasudha@HPLaptop:~$ cd zokrates_example
vasudha@HPLaptop:~/zokrates_example$ nano add.zok
vasudha@HPLaptop:~/zokrates_example$ zokrates compile -i add.zok
Compiling add.zok
```

Proof Generation and Verification:



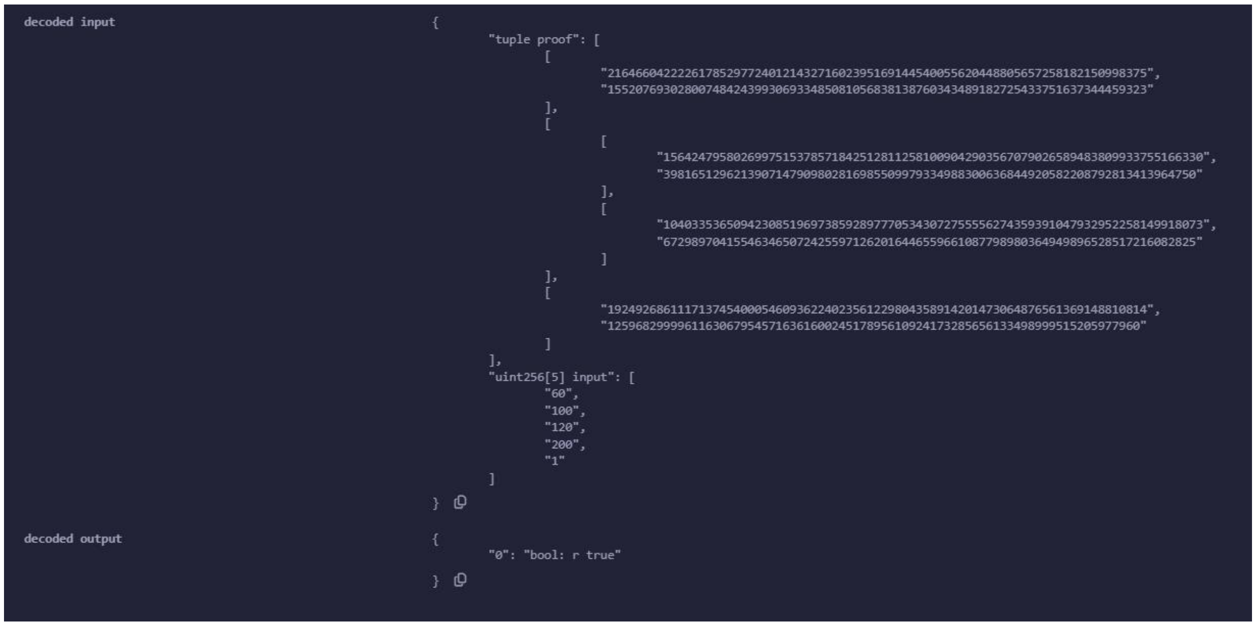
```
  = expected ty_field
vasudha@HPLaptop:~/zokrates_example$ nano add.zok
vasudha@HPLaptop:~/zokrates_example$ zokrates compile -i add.zok
Compiling add.zok

Compiled code written to 'out'
Number of constraints: 3
vasudha@HPLaptop:~/zokrates_example$ zokrates setup
Performing setup...
Verification key written to 'verification.key'
Proving key written to 'proving.key'
Setup completed
vasudha@HPLaptop:~/zokrates_example$ zokrates compute-witness -a 3 5 8
Computing witness...
Witness file written to 'witness'
vasudha@HPLaptop:~/zokrates_example$ zokrates generate-proof
Generating proof...
Proof written to 'proof.json'
vasudha@HPLaptop:~/zokrates_example$ zokrates export-verifier
Exporting verifier...
Verifier exported to 'verifier.sol'
vasudha@HPLaptop:~/zokrates_example$ zokrates verify
Performing verification...
PASSED
vasudha@HPLaptop:~/zokrates_example$
```

Files created after Proof Generation and Verification:



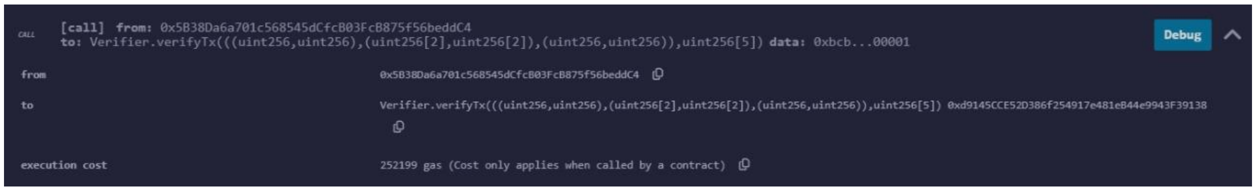
Name	Date modified	Type	Size
abi	03-10-2024 09:56 AM	JSON Source File	1 KB
add.zok	03-10-2024 09:56 AM	ZOK File	1 KB
out	03-10-2024 09:56 AM	File	2 KB
out.r1cs	03-10-2024 09:56 AM	R1CS File	1 KB
out.wtns	03-10-2024 09:56 AM	WTNS File	1 KB
proof	03-10-2024 09:57 AM	JSON Source File	1 KB
proving.key	03-10-2024 09:56 AM	KEY File	4 KB
verification.key	03-10-2024 09:56 AM	KEY File	2 KB
verifier.sol	03-10-2024 09:57 AM	SOL File	10 KB
witness	03-10-2024 09:56 AM	File	1 KB

Input and Output (In Backend):



```
decoded input                   {
                                        "tuple proof": [
                                                [
                                                        "21646604222261785297724012143271602395169144540055620448805657258182150998375",
                                                        "15520769302800748424399306933485081056838138760343489182725433751637344459323"
                                                ],
                                                [
                                                        [
                                                                "15642479580269975153785718425128112581009042903567079026589483809933755166330",
                                                                "3981651296213907147909802816985509979334988300636844920582208792813413964750"
                                                        ],
                                                        [
                                                                "10403353650942308519697385928977705343072755556274359391047932952258149918073",
                                                                "6729897041554634650724255971262016446559661087798980364949896528517216082825"
                                                        ]
                                                ],
                                                [
                                                        "19249268611171374540005460936224023561229804358914201473064876561369148810814",
                                                        "12596829999611630679545716361600245178956109241732856561334989995152059779600"
                                                ]
                                        ],
                                        "uint256[5] input": [
                                                "60",
                                                "100",
                                                "120",
                                                "200",
                                                "1"
                                        ]
                                }

decoded output                  {
                                        "0": "bool: r true"
                                }
```

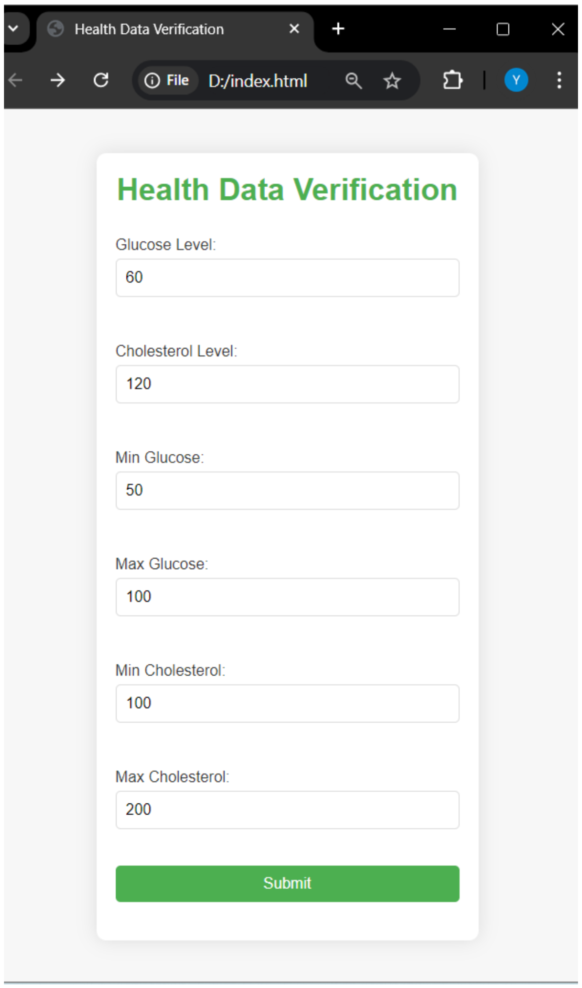
Optimized Gas Cost for Transaction:



```
CALL   [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
       to: Verifier.verifyTx(((uint256,uint256),(uint256[2],uint256[2]),(uint256,uint256)),uint256[5]) data: 0xbcb...00001

from                  0x5B38Da6a701c568545dCfcB03FcB875f56beddC4

to                    Verifier.verifyTx(((uint256,uint256),(uint256[2],uint256[2]),(uint256,uint256)),uint256[5]) 0xd9145CCE52D386f254917e481eB44e9943F39138

execution cost        252199 gas (Cost only applies when called by a contract)
```

=

User Interface:



Health Data Verification

Glucose Level: 60

Cholesterol Level: 120

Min Glucose: 50

Max Glucose: 100

Min Cholesterol: 100

Max Cholesterol: 200

8. Challenges Faced

8.1. Switching from Circom to ZoKrates:
Initially, we chose Circom for the zero-knowledge proof circuit but faced a lack of resources. We switched to ZoKrates, but it also had limited documentation, forcing us to rely on various websites and forums to complete the implementation.

8.2.Windows to Ubuntu Migration:
Running ZoKrates on Windows caused several command errors due to compatibility issues. We migrated to Ubuntu, where the proof generation worked smoothly without errors.

8.3. MetaMask Integration Challenges:
Initially, we attempted to deploy the smart contracts without using MetaMask as we didn’t have any Ether. Later, we realized MetaMask was necessary for integration, so we acquired free Ether and successfully deployed the contracts.

9. Future Enhancements

The future scope of this project includes several key enhancements aimed at improving functionality, accessibility, and scalability:

1. **User Authentication:** Adding a secure user authentication system is a crucial next step. This will ensure that only authorized individuals, such as doctors, patients, and healthcare providers, can access and verify sensitive medical information. Features like multi-factor authentication (MFA) and role-based access control (RBAC) could be implemented to strengthen security.

2. **Broader Integration:** The system can be expanded to include additional medical tests and parameters beyond blood tests, making it applicable to a wider range of healthcare needs. This would allow for verification of more diverse medical data, such as X-rays, MRI results, or other diagnostics.

3. **Mobile Application Development:** Developing a mobile application will enhance accessibility, making it easier for patients and healthcare providers to interact with the system. A mobile interface would offer convenient access to medical verifications and results on the go, improving user experience and engagement.

These improvements will further solidify the system’s role in protecting patient privacy while offering greater utility and ease of use for medical professionals and patients alike

10. Conclusion

This project explores the use of zero-knowledge proofs (zk-SNARKs) to enhance data privacy within the healthcare sector, specifically focusing on blood test result verification. Zero-knowledge proofs allow the verification of specific data without revealing the actual information, ensuring the privacy of sensitive patient details.

Healthcare providers can verify whether a patient's blood test results meet certain health conditions (such as acceptable glucose or cholesterol levels) without accessing the raw data itself. This fosters trust between patients and providers by maintaining confidentiality while still enabling accurate verification. The integration of zk-SNARKs with blockchain technology ensures secure, decentralized proof generation, making the system transparent and tamper-resistant.

Additionally, the project focuses on optimizing transaction gas costs, a common challenge when deploying cryptographic proofs on Ethereum. By exploring zk-SNARKs, the project aims to reduce computational overhead, making the system more efficient and scalable. This approach provides a privacy-preserving, cost-effective solution for sharing and verifying medical information, potentially transforming data management in healthcare.

11. References :

[1] J.Eberhardt and S. Tai, ”ZoKrates - Scalable Privacy-Preserving Off-Chain Computations,” 2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), Halifax, NS, Canada, 2018, pp. 1084-1091, doi: 10.1109/Cybermatics_2018.2018.00199. keywords: {Blockchain;Peer-to-peer computing;Computational modeling;Smart contracts;Scalability;Bitcoin;Privacy;ZoKrates;zkSNARKs;off-chain;scalability;privacy},

[2] https://zokrates.github.io/

[3]https://medium.com/coinmonks/zokrates-zksnarks-on-ethereum-made-easy-8022300f8ba6

[4] https://remix-ide.readthedocs.io/en/latest/

[5] https://docs.circom.io/

[6] https://dev.to/turupawn/zk-speedrun-3-dsls-in-15-minutes-noir-circom-zokrates-g3d

12.Appendix

1.Zokrates Circuit file (bloodtest.zok) :

```
def main(
    private field glucose_level,
    private field cholesterol_level,
    field min_glucose,
    field max_glucose,
    field min_cholesterol,
    field max_cholesterol
) -> field {
    // Check if glucose level is within the healthy range
    field glucose_check_min = if glucose_level >= min_glucose { 1 } else { 0
};
    field glucose_check_max = if glucose_level <= max_glucose { 1 } else { 0
};

    // Check if cholesterol level is within the healthy range
    field cholesterol_check_min = if cholesterol_level >= min_cholesterol { 1
} else { 0 };
    field cholesterol_check_max = if cholesterol_level <= max_cholesterol { 1
} else { 0 };

    // Ensure both glucose and cholesterol checks pass
    assert(glucose_check_min == 1);
    assert(glucose_check_max == 1);
    assert(cholesterol_check_min == 1);
    assert(cholesterol_check_max == 1);

    // Return 1 if all checks pass (as a signal of success)
    return 1;
}
```

2.index.html :

```
//index.html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Health Data Verification</title>
    <link rel="stylesheet" href="styles.css"> <!-- Link to the CSS file -->
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/web3/1.7.0/web3.min.js"></script>
<!-- Web3.js -->
    <script src="app.js" defer></script> <!-- Link to the JS file -->
</head>
```

```
<body>
    <div class="container">
        <h1>Health Data Verification</h1>
        <form id="healthForm">
            <label for="glucose">Glucose Level:</label>
            <input type="number" id="glucose" required><br>

            <label for="cholesterol">Cholesterol Level:</label>
            <input type="number" id="cholesterol" required><br>

            <label for="min_glucose">Min Glucose:</label>
            <input type="number" id="min_glucose" required><br>

            <label for="max_glucose">Max Glucose:</label>
            <input type="number" id="max_glucose" required><br>

            <label for="min_cholesterol">Min Cholesterol:</label>
            <input type="number" id="min_cholesterol" required><br>

            <label for="max_cholesterol">Max Cholesterol:</label>
            <input type="number" id="max_cholesterol" required><br>

            <button type="submit">Submit</button>
        </form>

        <p id="result"></p>
    </div>
</body>
</html>
```

3.styles.css

```
//index.html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Health Data Verification</title>
    <link rel="stylesheet" href="styles.css"> <!-- Link to the CSS file -->
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/web3/1.7.0/web3.min.js"></script>
<!-- Web3.js -->
    <script src="app.js" defer></script> <!-- Link to the JS file -->
</head>
<body>
    <div class="container">
        <h1>Health Data Verification</h1>
        <form id="healthForm">
            <label for="glucose">Glucose Level:</label>
            <input type="number" id="glucose" required><br>

            <label for="cholesterol">Cholesterol Level:</label>
            <input type="number" id="cholesterol" required><br>
            <label for="min_glucose">Min Glucose:</label>
            <input type="number" id="min_glucose" required><br>

            <label for="max_glucose">Max Glucose:</label>
            <input type="number" id="max_glucose" required><br>

            <label for="min_cholesterol">Min Cholesterol:</label>
            <input type="number" id="min_cholesterol" required><br>

            <label for="max_cholesterol">Max Cholesterol:</label>
            <input type="number" id="max_cholesterol" required><br>

            <button type="submit">Submit</button>
        </form>

        <p id="result"></p>
    </div>
</body>
</html>
```

4.app.js

```
//app.js
let web3;
let contract;

const contractAddress = "YOUR_DEPLOYED_CONTRACT_ADDRESS"; // Replace with the
contract address
const contractABI = YOUR_CONTRACT_ABI; // Replace with the contract's ABI from
Remix

window.addEventListener('load', async () => {
    // Check if the web3 instance is injected (e.g., by MetaMask)
    web3 = new Web3(Web3.givenProvider || "http://localhost:8545");

    // Initialize the contract using ABI and address
    contract = new web3.eth.Contract(contractABI, contractAddress);
});

document.getElementById('healthForm').addEventListener('submit', async
function(event) {
    event.preventDefault();

    const glucose = document.getElementById('glucose').value;
    const cholesterol = document.getElementById('cholesterol').value;
    const min_glucose = document.getElementById('min_glucose').value;
    const max_glucose = document.getElementById('max_glucose').value;
    const min_cholesterol = document.getElementById('min_cholesterol').value;
    const max_cholesterol = document.getElementById('max_cholesterol').value;

    // Prepare the proof (dummy values, you'll need real proof)
    const proof = {
        a: [0, 0], // Replace with real proof values
        b: [[0, 0], [0, 0]], // Replace with real proof values
        c: [0, 0], // Replace with real proof values
        input: [glucose, cholesterol] // Inputs for your verifier contract
    };

    try {
        const result = await contract.methods.verifyProof(proof.a, proof.b,
proof.c, proof.input).call();
        document.getElementById('result').innerHTML = Verification Result:
${result};
    } catch (error) {
        console.error("Error verifying proof:", error);
        document.getElementById('result').innerHTML = "Error verifying
proof.";
    }
});
```