Health-ID Verification Metrics with Digital Signatures

Bonafide Record of Work Done By

Deepiga S	(21Z214)
Monaleka M	(21Z231)
Subhasri Shreya S L	(21Z260)
Thirikasha S	(21Z263)
V Harsheni	(21Z264)

19Z701 - CRYPTOGRAPHY

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

BACHELOR OF ENGINEERING

Branch: COMPUTER SCIENCE & ENGINEERING

of PSG College of Technology

October 2024

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

PSG COLLEGE OF TECHNOLOGY

(Autonomous Institution)

COIMBATORE – 641 004

CONTENTS

1. ABSTRACT

This project focuses on developing a blockchain-based Health-ID verification system that measures the efficiency and accuracy of digital signatures, specifically the Elliptic Curve Digital Signature Algorithm (ECDSA), for verifying Health-IDs stored on the blockchain. The system evaluates key performance metrics such as verification time, false positive rates (identifying invalid Health-IDs as valid), and false negative rates (failing to identify valid Health-IDs). By leveraging blockchain's inherent features of immutability and transparency, the project explores how digital signatures impact Health-ID validation. The tools used include Ganache for local blockchain simulation and Truffle for development and testing. This report outlines the system architecture, implementation, and performance results of this approach.

2. INTRODUCTION

2.1 Problem Statement

Health-ID Verification Metrics with Digital Signatures: Develop mathematical models to measure the efficiency and accuracy of verifying healthIDs on the blockchain. Consider factors like verification time, false positives, and false negatives. Explore the impact of digital signatures(e.g., ECDSA) for healthID validation.

2.2 Description Of The Problem:

The problem involves verifying Health-IDs (identifiers for individuals' health records) stored on a blockchain using digital signatures such as ECDSA (Elliptic Curve Digital Signature Algorithm). Key metrics for evaluation include verification time, false positive rates (incorrectly identifying an invalid Health-ID as valid), and false negative rates (incorrectly identifying a valid Health-ID as invalid).

2.3 Objective Of The Problem

Develop mathematical models to quantitatively assess and optimize the efficiency and accuracy of Health-ID verification processes on blockchain platforms. This includes:

● Blockchain Integration: Utilize blockchain technology (Ganache) to store Health-IDs in a secure, transparent, and immutable manner, ensuring the integrity of the Health-ID verification process.
● Measurement of Efficiency: Design models to calculate and optimize the verification time required for validating Health-IDs using blockchain technology.
● Evaluation of Accuracy: Create frameworks to analyze false positives (incorrectly identifying invalid Health-IDs as valid) and false negatives (failing to recognize valid Health-IDs).
● Impact Analysis of Digital Signatures: Explore the role and effectiveness of digital signatures, such as ECDSA (Elliptic Curve Digital Signature Algorithm), in enhancing the security and reliability of Health-ID validation on the blockchain.

2.4 Scope:

Smart Contract Development: Creating smart contracts to securely store and verify Health-IDs on a blockchain. This will involve writing the necessary code and deploying it on a local blockchain environment.

Implementation Of Ecdsa: Integrating ECDSA for signing and verifying Health-IDs, ensuring that the verification process is both secure and efficient.

Mathematical Modeling: Developing models to measure verification times, FPR, and FNR, analyzing data collected from real-world testing to improve the verification process.

Frontend Development: Building a user-friendly web dashboard that allows users to submit Health-IDs, view verification results, and visualize metrics related to the verification process.

Integration With Blockchain Tools: Utilizing tools such as Ganache for local testing and Truffle for deployment to an Ethereum Testnet, facilitating a comprehensive workflow from development to testing and deployment.

3. REQUIREMENTS ANALYSIS

3.1 Stakeholder Identification

- **Healthcare Providers:** Hospitals, clinics, and doctors who need to verify Health-IDs.
- **Patients:** Individuals whose Health-IDs are being verified.
- **Regulators:** Authorities ensuring the system meets legal and ethical standards.
- **It Support Teams:** Responsible for maintaining the system and resolving issues.

3.2 Functional Requirements

- **Submit Health-Id:** Allow users to submit Health-IDs for verification.
- **Verify Health-Id:** Check if the submitted Health-ID is valid using digital signatures.
- **Retrieve Health-Id:** Fetch Health-ID records from the blockchain for verification.
- **Measure Performance:** Track how long it takes to verify Health-IDs and log this data.
- **Analyze Signatures:** Assess the impact of using ECDSA (a type of digital signature) on the verification process.

3.3 Non-Functional Requirements

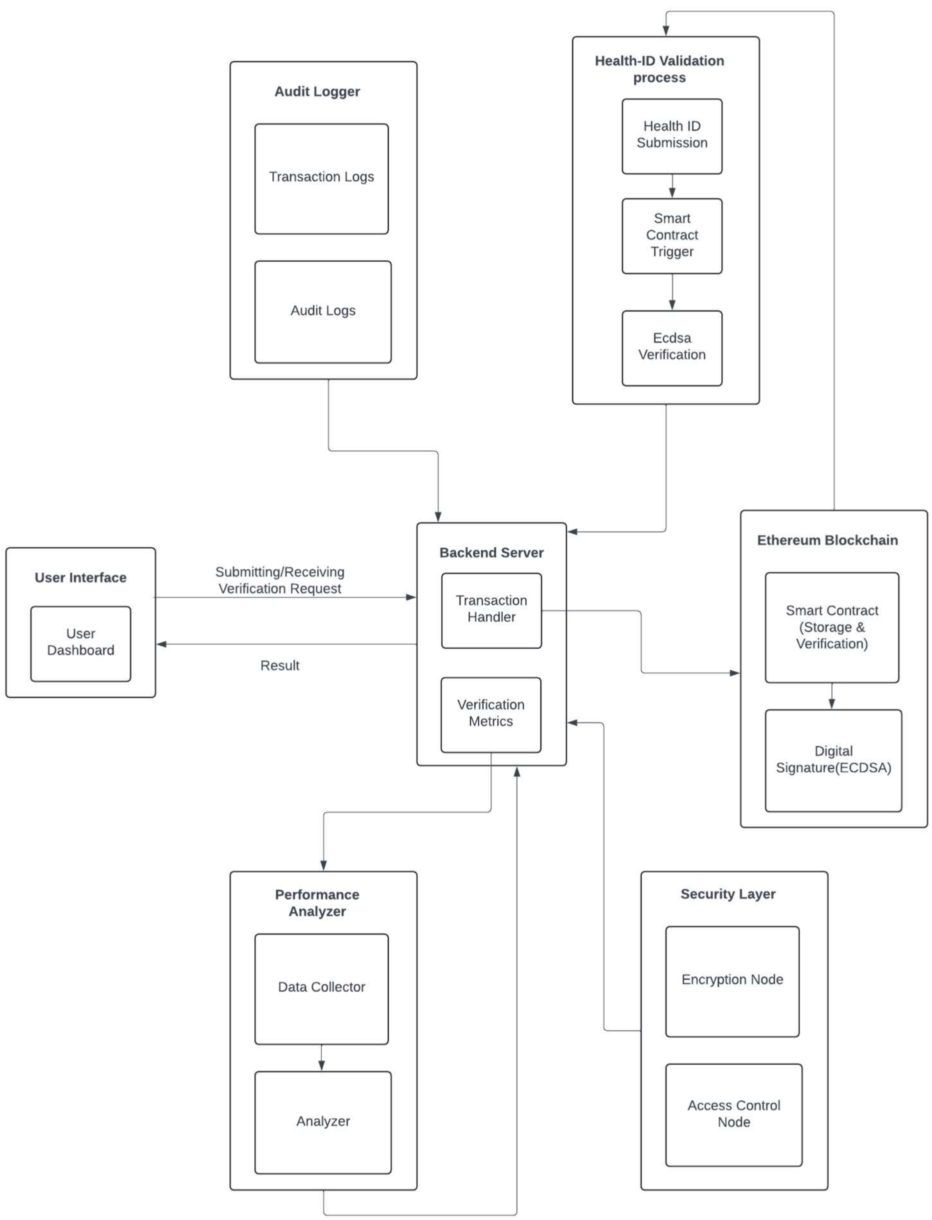
- **Response Time:** Verify Health-IDs within a few seconds.
- **Throughput:** Handle hundreds or thousands of verification requests per minute.
- **Data Security:** Encrypt Health-ID data in transit and at rest using TLS and AES, ensure data integrity during storage and transmission, and comply with GDPR and HIPAA regulations.
- **Access Control:** Implement multi-factor authentication and use role-based access controls (RBAC) to manage system access and data permissions.
- **Backup And Recovery:** Regularly backup data and have a recovery plan in place.

3.4 Data Type Specification

- **Health-Id Data:** Unique identifiers associated with health records.
- **Transaction Data:** Information about each verification transaction (e.g., time taken, result).
- **Signature Data:** Digital signatures used for validating Health-IDs.

4. SYSTEM ARCHITECTURE

4.1 Architecture diagram



The architecture diagram provides a visual representation of the system's components and how they interact. The core flow involves:

- **User Interaction**: Users submit their Health-IDs through a React-based web interface.
- **Backend Processing**: The backend server processes the Health-ID request and interacts with the blockchain.

- **Blockchain Execution**: The blockchain smart contracts handle Health-ID storage and verification using the ECDSA digital signature algorithm.
- **Performance Analysis**: The Python script analyzes and visualizes performance metrics such as verification time and the accuracy of the system (false positives/negatives).
- **Audit Logging**: Every transaction and action is logged for auditing purposes, providing an immutable record of all interactions.

4.2 Key Components

Blockchain (Ganache & Ethereum)

- **Ganache & Truffle**: A local simulated blockchain environment used to prototype, test, and debug the system before deploying it to the Ethereum Testnet.

- **Ethereum Blockchain**: The platform where the smart contracts are deployed for real-world Health-ID verification. The blockchain stores Health-IDs immutably and validates them using digital signatures.

  Smart Contracts for Health-ID Storage and Verification:Written in Solidity and deployed on the blockchain, these smart contracts handle:

  - **Health-ID Submission**: Securely storing Health-IDs on the blockchain.
  - **ECDSA Verification**: Authenticating submitted Health-IDs using the Elliptic Curve Digital Signature Algorithm (ECDSA).
  - **Transaction Logging**: Every interaction with the Health-ID (e.g., submission, verification) is logged, ensuring an immutable audit trail.

Backend (Python)

- The backend, built in Python, manages the interaction between the UI, blockchain, and performance analytics. Its core tasks include:
  - **Transaction Handler**: Submitting and retrieving Health-IDs from the blockchain using API calls.
  - **Verification Metrics**: Collecting key performance metrics such as verification time, false positive rates, and false negative rates. The backend uses libraries like NumPy, Pandas for analysis, and Matplotlib, Seaborn for visualizing results.
  - **Smart Contract Interaction**: Facilitates smooth interaction between the frontend and blockchain, triggering contract execution when Health-IDs are submitted for verification.

Frontend (Web Interface)

- **User Interface (React.js)**: The web interface, built using React.js, provides an intuitive dashboard for users (e.g., healthcare providers, regulators) to submit Health-IDs for verification.

It displays real-time verification results and performance metrics, and logs interactions for easy monitoring and regulatory compliance.

- **Web3.js**: An Ethereum JavaScript API that interacts with the blockchain, enabling the UI to communicate with smart contracts.

Performance Analysis Tools (Python)

- **performance_metrics.py**: A Python script designed to evaluate the system’s efficiency and accuracy. It connects to the blockchain to measure key performance indicators such as:
    - **Verification Time**
    - **False Positive Rates**
    - **False Negative Rates**

5. TOOLS AND TECHNOLOGIES

5.1 Technologies Used

1. **Blockchain Technology**
    - **Ethereum**: A decentralized platform for deploying smart contracts, enabling secure Health-ID verification.
2. **Smart Contract Development**
    - **Solidity**: The programming language used to write smart contracts for Health-ID storage and verification.
    - **Truffle**: A development framework that simplifies smart contract compilation, migration, and testing.
3. **Local Blockchain Simulation**
    - **Ganache**: A personal blockchain for testing, allowing rapid prototyping and debugging of smart contracts.
4. **Frontend Development**
    - **React.js**: A JavaScript library for building an interactive user interface.
    - **Web3.js**: An Ethereum JavaScript API for seamless blockchain interaction from the frontend.
5. **Backend Development**
    - **Python**: Manages interactions between the UI and blockchain and analyzes performance metrics.

5.2 Prerequisites

- **Node.js and npm**
- **Truffle**
- **Ganache**
- **Python** (with web3 library installed) is required to run the system.

6. IMPLEMENTATION

.

6.1 Smart Contract Development

The smart contracts, written in Solidity, form the backbone of the Health-ID verification process. Key aspects include:

- **HealthIDVerification.sol**: This main smart contract manages Health-ID storage and verification. It includes:
  - **Storage Functionality**: Securely stores Health-IDs on the blockchain, ensuring data integrity and immutability.
  - **ECDSA Signature Verification**: Implements the logic to verify the authenticity of Health-IDs using ECDSA digital signatures.
  - **Transaction Logging**: Records all interactions (submissions and verifications) to create an immutable audit trail.

6.2 Backend Implementation

The backend, developed in Python, is responsible for managing interactions between the user interface and the blockchain. Its primary functions include:

- **Transaction Management**: Handles requests for submitting and retrieving Health-IDs through API calls to the blockchain.
- **Performance Metrics Analysis**: Utilizes Python to analyze key metrics, including verification time, false positive rates, and false negative rates.
- **Smart Contract Interaction**: Facilitates communication between the frontend and the blockchain, ensuring efficient execution of smart contracts.

6.3 Frontend Development

The user interface, built with React.js, provides a seamless experience for users interacting with the Health-ID verification system:

- **User Dashboard**: Allows users to submit Health-IDs for verification and view results in an intuitive layout.
- **Real-time Updates**: Displays verification results, performance metrics, and transaction logs dynamically, keeping users informed of the process.

6.4 Performance Metrics Evaluation

The project includes a dedicated Python script (performance_metrics.py) that evaluates the efficiency and accuracy of the Health-ID verification process:

- **Data Collection**: The script gathers metrics related to verification time, false positives, and false negatives, allowing for comprehensive performance analysis.

6.5 Testing and Deployment

- **Local Testing on Ganache**: The entire system is initially tested on the Ganache local blockchain to ensure all functionalities operate as expected.
- **Deployment to Ethereum**: Using Truffle, the smart contracts are deployed to the Ethereum Testnet, and the React app is made accessible to users.

6.6 Project Structure:

- contracts/: Contains Solidity smart contracts for the Health-ID verification system. HealthIDVerification.sol: Main smart contract for storing and verifying Health-IDs.
- client/: React-based frontend to interact with the smart contract. src/HealthIDVerification.js: Main React component for Health-ID storage and verification. src/HealthIDVerification.css: Styles for the UI.
- migrations/: Truffle migrations to deploy the contract.
- build/contracts/: Compiled contract artifacts (JSON files generated by Truffle).
- performance_metrics.py (inside build/contracts/): Python script to measure the efficiency and accuracy of the verification process. It connects to the blockchain and evaluates factors like verification time, false positives, and false negatives.
- truffle-config.js: Truffle configuration file.

6.7 Smart Contract Compilation And Deployment:

1. truffle init -> Initialize truffle project.
2. truffle compile -> Compile the smart contracts.
3. truffle migrate --network development -> Deploy the contracts to the development network.

   After deployment, Truffle generates a corresponding JSON file for each smart contract inside the build/contracts/ directory that contains important information about the deployed contracts.

6.8 Running The React UI:

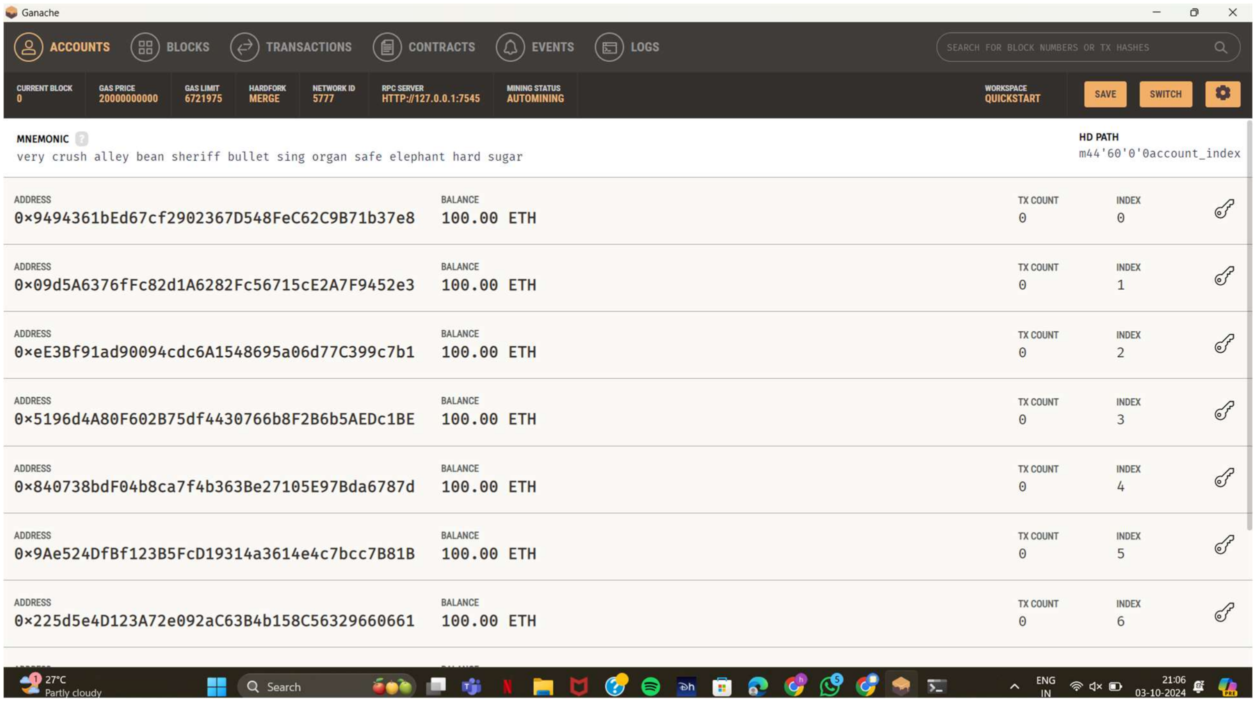
1. Navigate to the client directory.
2. npm start -> To start the React app.

6.9 Performance Metrics Evaluation:

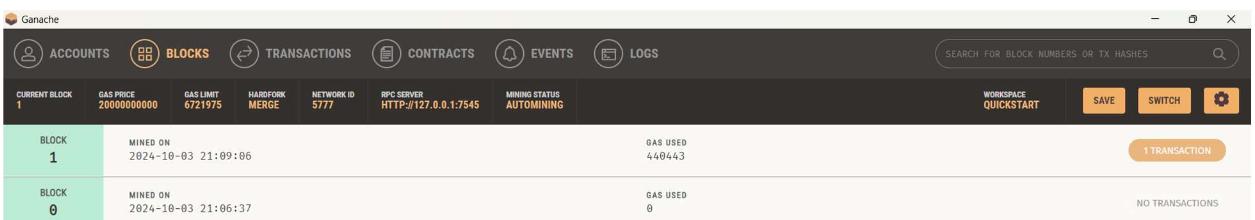
1. Ensure local blockchain (Ganache) is running.
2. Run the script: python performance_metrics.py

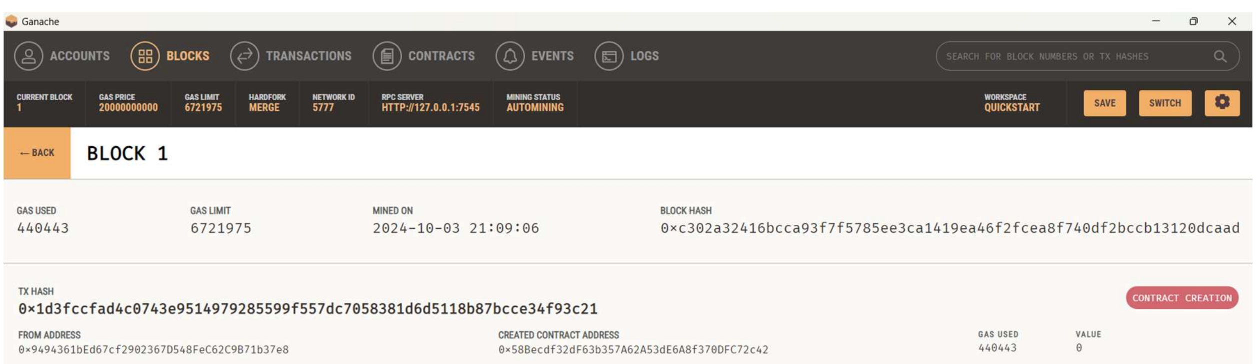
7. PERFORMANCE AND RESULTS

Quick Start Ethereum:



Smart Contract Compilation And Deployment:





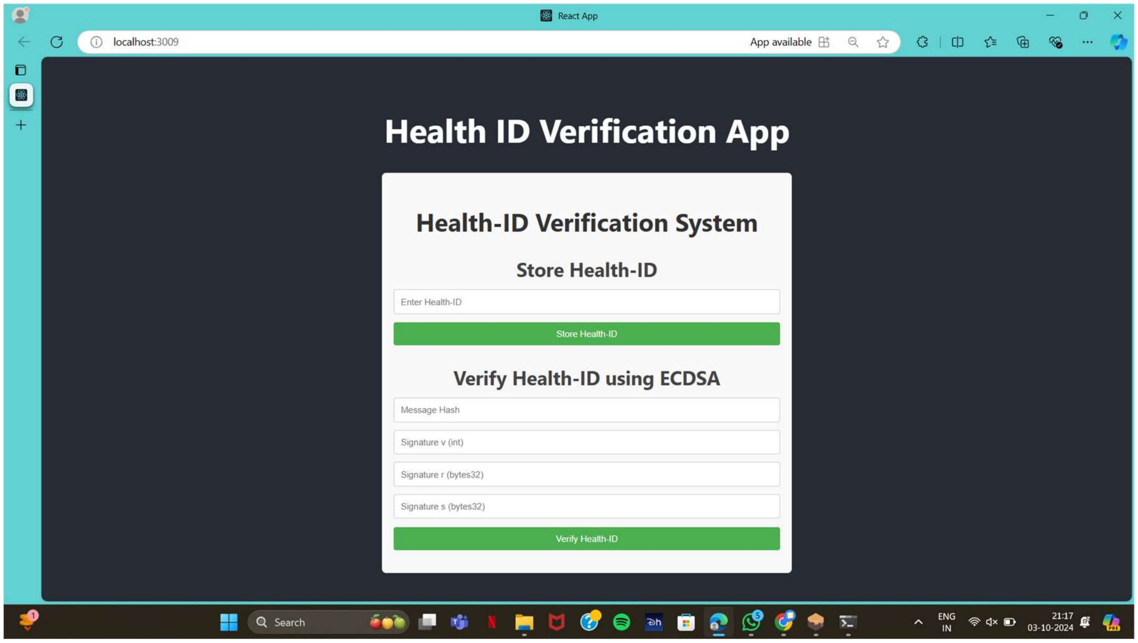
```
D:\STUDY MATERIALS\SEM 7\Cryptography\health-id-project>truffle compile

Compiling your contracts...
===========================
> Compiling .\contracts\HealthIDVerification.sol
> Artifacts written to D:\STUDY MATERIALS\SEM 7\Cryptography\health-id-project\build\contracts
> Compiled successfully using:
   - solc: 0.8.21+commit.d9974bed.Emscripten.clang
```

```
D:\STUDY MATERIALS\SEM 7\Cryptography\health-id-project>truffle migrate --network development

Compiling your contracts...
===========================
> Compiling .\contracts\HealthIDVerification.sol
> Artifacts written to D:\STUDY MATERIALS\SEM 7\Cryptography\health-id-project\build\contracts
> Compiled successfully using:
   - solc: 0.8.21+commit.d9974bed.Emscripten.clang


Starting migrations...
======================
> Network name:    'development'
> Network id:      5777
> Block gas limit: 6721975 (0x6691b7)


2_deploy_healthid.js
====================

   Replacing 'HealthIDVerification'
   --------------------------------
   > transaction hash:    0x1d3fccfad4c0743e9514979285599f557dc7058381d6d5118b87bcce34f93c21
   > Blocks: 0            Seconds: 0
   > contract address:    0x58Becdf32dF63b357A62A53dE6A8f370DFC72c42
   > block number:        1
   > block timestamp:     1727969946
   > account:             0x9494361bEd67cf2902367D548FeC62C9B71b37e8
   > balance:             99.998513504875
   > gas used:            440443 (0x6b87b)
   > gas price:           3.375 gwei
   > value sent:          0 ETH
   > total cost:          0.001486495125 ETH

   > Saving artifacts
   -------------------------------------
   > Total cost:      0.001486495125 ETH

Summary
=======
> Total deployments:   1
> Final cost:          0.001486495125 ETH
```

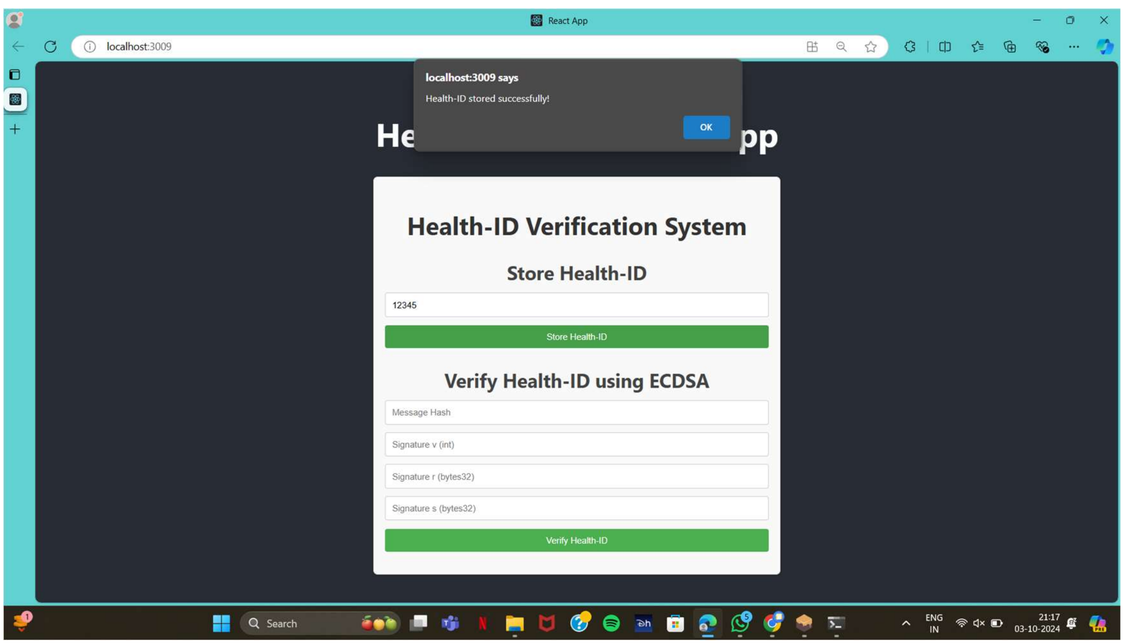
Starting React App:

```
D:\STUDY MATERIALS\SEM 7\Cryptography\health-id-project>cd client

D:\STUDY MATERIALS\SEM 7\Cryptography\health-id-project\client>npm start

> client@0.1.0 start
> react-scripts start
```

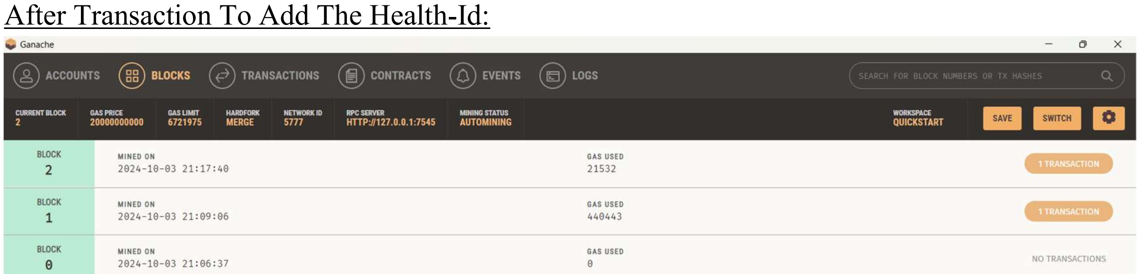
Initial Frontend:



Storing Health-Id:

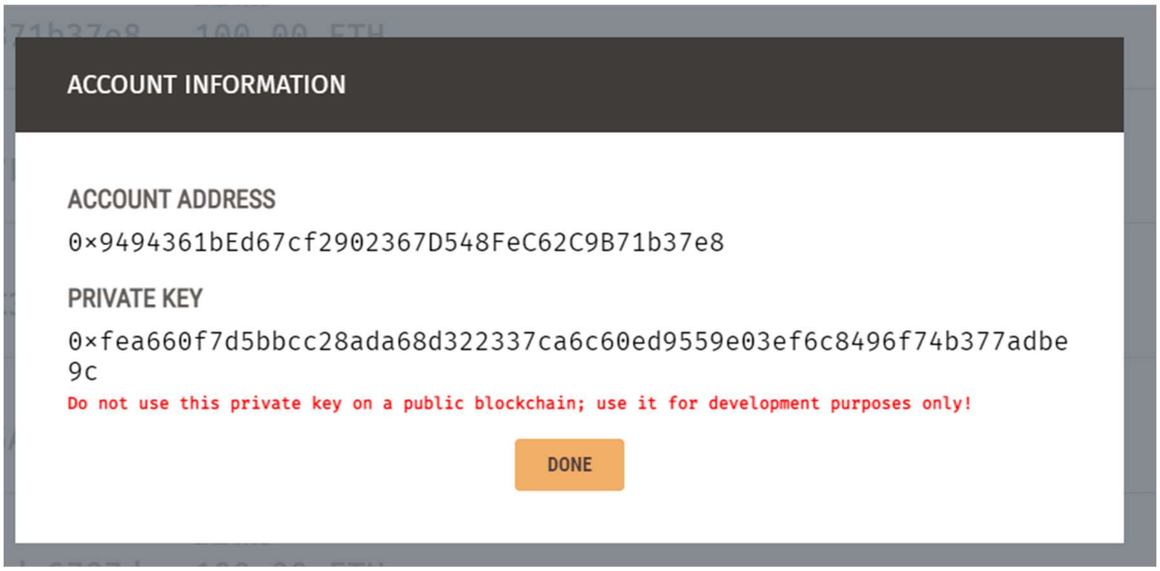


After Transaction To Add The Health-Id:



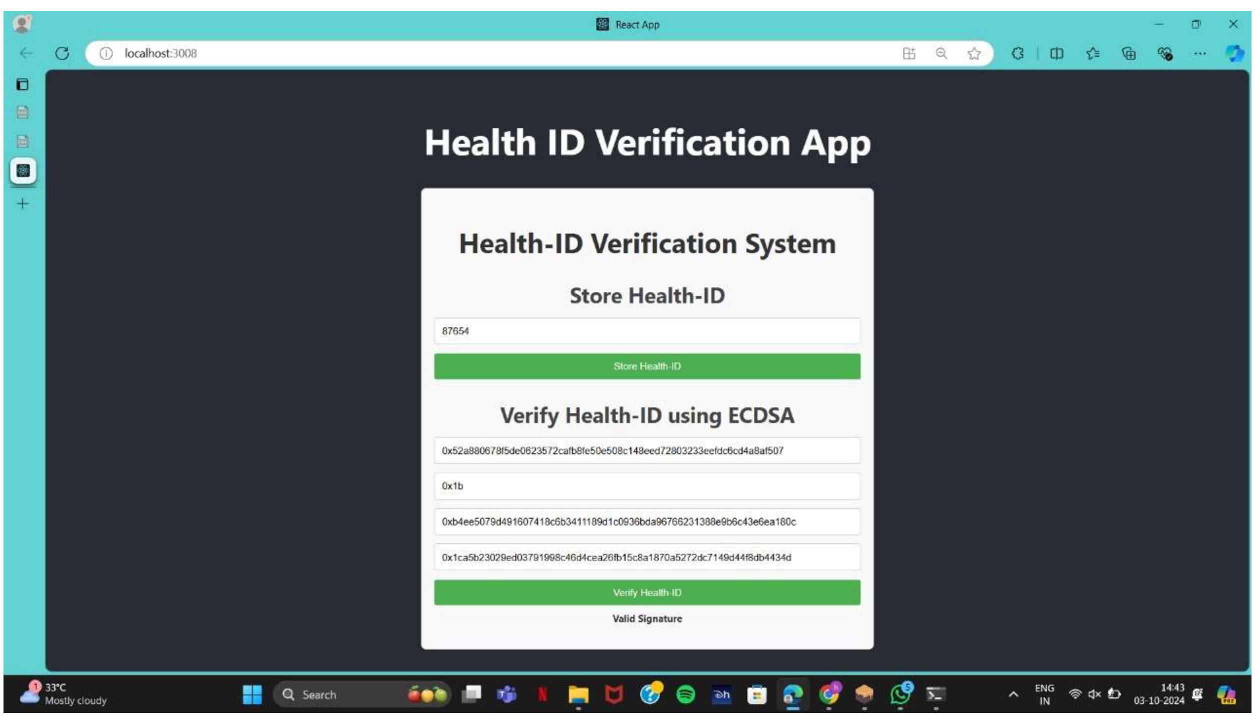
Sample I/O:

```
D:\STUDY MATERIALS\SEM 7\Cryptography\health-id-project>truffle console
truffle(development)> const healthID = "12345";
undefined
truffle(development)> const messageHash = web3.utils.sha3(healthID);
undefined
truffle(development)> console.log("Message Hash:", messageHash);
Message Hash: 0x1841d653f9c4edda9d66a7e7737b39763d6bd40f569a3ec6859d3305b72310e6
undefined
truffle(development)> const privateKey = "0xfea660f7d5bbcc28ada68d322337ca6c60ed9559e03ef6c8496f74b377adbe9c"; //private key taken from ganache
undefined
truffle(development)> const signature = await web3.eth.accounts.sign(messageHash, privateKey);
undefined
truffle(development)> console.log("Signature:", signature);
Signature: {
  message: '0x1841d653f9c4edda9d66a7e7737b39763d6bd40f569a3ec6859d3305b72310e6',
  messageHash: '0x650d029228ac8fd4f890d1478336fe7b45678e1ffeb78c7ff18d08d14f7fccee',
  v: '0x1c',
  r: '0x431b9b10f6fe6e6d8cd134d9f8494b53c078d8249a8827b112244156 4abf9df8',
  s: '0x0953ec99e5e443300fe230a5ec93fc2cde01641694e034620019d5faf481d412',
  signature: '0x431b9b10f6fe6e6d8cd134d9f8494b53c078d8249a8827b1122441564abf9df80953ec99e5e443300fe230a5ec93fc2cde01641694e034620019d5faf481d4121c'
}
undefined
truffle(development)> |
```

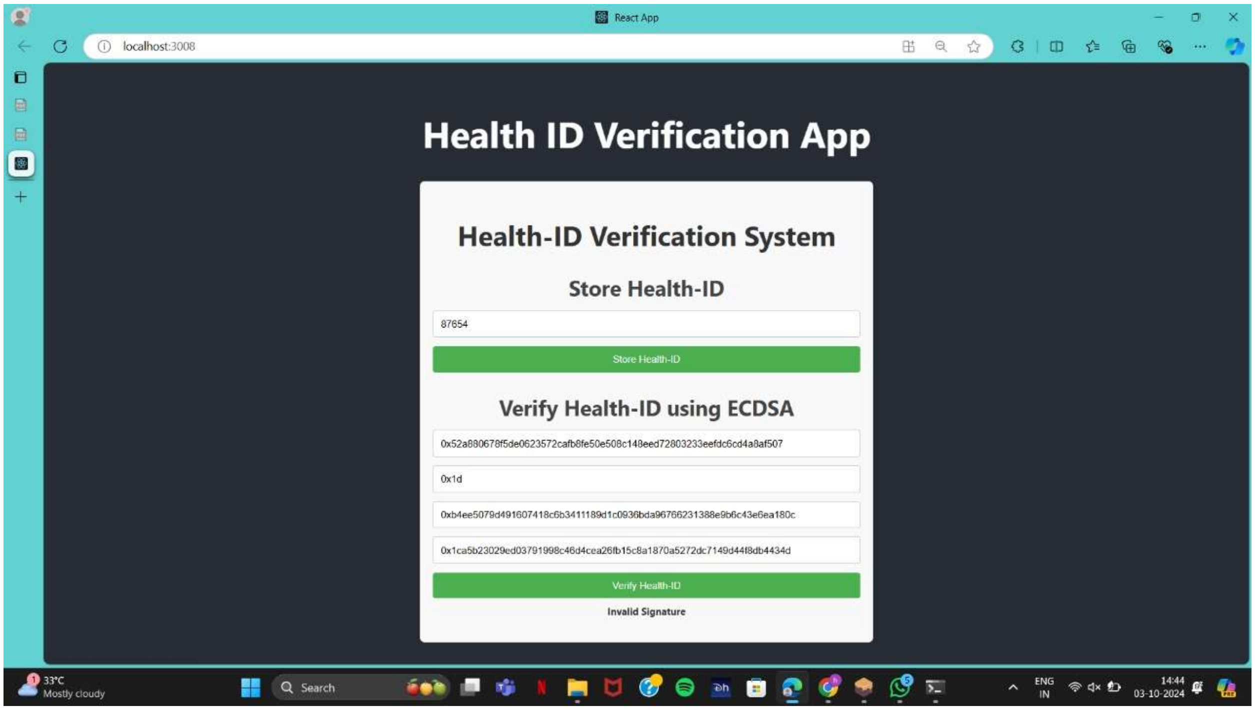


Private Key from ganache

Valid Signature:



Invalid Signature:



Performance Metrics:

```
PS D:\STUDY MATERIALS\SEM 7\Cryptography\health-id-project\build\contracts> python performance_metrics.py
Connected to Blockchain
Average Verification Time: 0.028607 seconds
False Positives: 0
False Negatives: 49
```

```
D:\STUDY MATERIALS\SEM 7\Cryptography\health-id-project\build\contracts>python performance_metrics.py
Connected to Blockchain
Average Verification Time: 0.015927 seconds
False Positives: 0
False Negatives: 0
```

8. CHALLENGES FACED

1. **Blockchain Environment Configuration**
   Setting up and configuring the local blockchain environment using Ganache and integrating it with the Ethereum Testnet proved to be more intricate than anticipated. Ensuring seamless connectivity between the development environment and the testnet required troubleshooting various network and configuration issues.

2. **ECDSA Signature Integration**
   Implementing and verifying ECDSA (Elliptic Curve Digital Signature Algorithm) signatures in Solidity posed a significant challenge. This process required a deep understanding of cryptographic principles, especially in translating complex cryptographic operations into Solidity's limited built-in functions.

3. **Time Management**
   Balancing the complexity of cryptographic integration with blockchain functionality within the project timeline was demanding. Managing time effectively became essential, with task prioritization and scheduling playing a key role in keeping the project on track.

4. **Performance Optimization**
   During the testing phase, performance bottlenecks were identified, particularly when handling multiple concurrent verification requests. These bottlenecks prompted the need for optimizations in both the backend logic and smart contract execution to enhance efficiency and scalability.

5. **Frontend-Backend Integration**
   Connecting the React frontend with the Ethereum smart contracts via Web3.js posed several integration challenges. Ensuring that the frontend seamlessly interacted with the blockchain, while handling asynchronous calls and error management, required extensive debugging and optimization.

9. FUTURE ENHANCEMENTS

1. **Scalability**:
   - The system can be enhanced to accommodate larger transaction volumes by migrating to more scalable blockchain platforms or incorporating off-chain solutions, such as **Layer 2 protocols.**
   - This will help reduce latency and lower transaction costs, enabling efficient handling of increased user demand.

2. **Integration with Healthcare Systems**:
   - Integrating the Health-ID verification system with existing **electronic health record (EHR)** systems will facilitate seamless access to health data for verified users.
   - This integration can streamline workflows for healthcare providers and improve patient outcomes by ensuring timely access to accurate health information.

3. **Advanced Cryptographic Techniques**:
   - Exploring more advanced cryptographic techniques, such as **Zero-Knowledge Proofs (ZKPs)**, can further enhance the security and privacy of the system.
   - ZKPs would allow for the verification of Health-IDs without exposing sensitive user data, thus instilling trust in the system.

4. **Real-World Deployment**:
   - Transitioning from a testnet environment to a **mainnet deployment** is crucial for real-world application.
   - This step will involve ensuring compliance with healthcare regulations and industry standards to facilitate widespread adoption while maintaining data integrity and user privacy.

10. CONCLUSION

In our implementation of Health-ID Verification Metrics with Digital Signatures, we used ECDSA (Elliptic Curve Digital Signature Algorithm) to securely verify health IDs on a blockchain. Key metrics such as verification time, false positives, and false negatives were tracked to measure the system's efficiency and accuracy. The use of ECDSA ensures cryptographic security while maintaining relatively fast verification times.

The results show that ECDSA provided reliable and efficient identity verification with minimal false positives and negatives. Our smart contract effectively utilized digital signatures to authenticate health IDs, ensuring both security and privacy within the system.

The impact of this project is significant, as it demonstrates the potential of integrating ECDSA with blockchain to enhance security in healthcare systems. This approach can prevent identity fraud, improve data integrity, and ensure trusted access to sensitive health information.

11. REFERENCES

[1]https://www.dappuniversity.com/articles/blockchain-app-tutorial

[2]https://www.ncbi.nlm.nih.gov/pmc/articles/PMC8230390/

[3]https://www.researchgate.net/publication/359920244_Securing_Medical_Records_of_COVID-19_Patients_Using_Elliptic_Curve_Digital_Signature_Algorithm_ECDSA_in_Blockchain

[4]https://cryptobook.nakov.com/digital-signatures/ecdsa-sign-verify-messages

[5]https://archive.trufflesuite.com/ganache/

[6]https://ethereum-blockchain-developer.com/2022-06-nft-truffle-hardhat-foundry/03-truffle-setup/

[7]https://cli.github.com/manual/

12. APPENDIX

SMART CODE DEPLOYMENT

1) HealthIDVerification.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract HealthIDVerification {
    // Mapping to store the Health-IDs
    mapping(string => bool) private healthIDs;
    // Event to log when a Health-ID is stored
    event HealthIDStored(string healthID, address sender);
        // Event to log the result of Health-ID verification
    event HealthIDVerified(string healthID, bool isValid);

    // Function to store a Health-ID
    function storeHealthID(string memory healthID) public {
        healthIDs[healthID] = true;
        emit HealthIDStored(healthID, msg.sender);
    }

    // Function to verify the Health-ID signature using ECDSA
    function verifyHealthID(
    string memory healthID,
    bytes32 hash,
    uint8 v,
    bytes32 r,
    bytes32 s
) public returns (bool) {
    // Recover the signer from the signature
    address signer = ecrecover(hash, v, r, s);
    // Check if the Health-ID exists and the signer is valid
    bool isValid = (healthIDs[healthID] && signer != address(0));
    // Emit the verification result
    emit HealthIDVerified(healthID, isValid);
    return isValid;
}
}
```

2) 2_deploy_healthid.js

```
const HealthIDVerification = artifacts.require("HealthIDVerification");
module.exports = function (deployer) {
    deployer.deploy(HealthIDVerification);
};
```

REACT APP

1) HealthIDVerification.js

```
import React, { useState } from 'react';
import web3 from '../utils/web3';
import HealthID from '../artifacts/HealthIDVerification.json';
import './HealthIDVerification.css'; // Create a CSS file for styling

const HealthIDVerification = () => {
  const [healthID, setHealthID] = useState('');
  const [messageHash, setMessageHash] = useState('');
  const [v, setV] = useState('');
  const [r, setR] = useState('');
  const [s, setS] = useState('');
  const [verificationResult, setVerificationResult] = useState('');

  const handleStoreHealthID = async () => {
    try {
      const networkId = await web3.eth.net.getId();
      const deployedNetwork = HealthID.networks[networkId];
      const contract = new web3.eth.Contract(HealthID.abi, deployedNetwork.address);

      if (!healthID) {
        console.error("Please enter a valid Health-ID");
        return;
      }

      const accounts = await web3.eth.getAccounts();
      await contract.methods.storeHealthID(healthID).send({ from: accounts[0] });

      // Pop-up alert for success
      alert("Health-ID stored successfully!");
    } catch (error) {
      console.error("Error storing Health-ID:", error);
    }
  };

  const handleVerifyHealthID = async () => {
    try {
      const networkId = await web3.eth.net.getId();
      const deployedNetwork = HealthID.networks[networkId];
      const contract = new web3.eth.Contract(HealthID.abi, deployedNetwork.address);

      if (!messageHash || !v || !r || !s) {
        console.error("Invalid ECDSA signature inputs");
        return;
      }

      const accounts = await web3.eth.getAccounts();
```

```
        const result = await contract.methods
          .verifyHealthID(healthID, messageHash, v, r, s)
          .call();

        console.log("Verification Result:", result);
        setVerificationResult(result ? 'Valid Signature' : 'Invalid Signature');
      } catch (error) {
        console.error("Error in verifying Health-ID:", error);
      }
    };

    return (
      <div className="container">
        <h2>Health-ID Verification System</h2>

        <div className="section">
          <h3>Store Health-ID</h3>
          <input
            type="text"
            placeholder="Enter Health-ID"
            value={healthID}
            onChange={(e) => setHealthID(e.target.value)}
            className="input-field"
            style={{ fontSize: '16px' }} // Increase font size for better readability
          />
          <button onClick={handleStoreHealthID} className="action-button">
            Store Health-ID
          </button>
        </div>

        <div className="section">
          <h3>Verify Health-ID using ECDSA</h3>
          <input
            type="text"
            placeholder="Message Hash"
            value={messageHash}
            onChange={(e) => setMessageHash(e.target.value)}
            className="input-field"
            style={{ fontSize: '16px' }} // Increase font size for better readability
          />
          <input
            type="text"
            placeholder="Signature v (int)"
            value={v}
            onChange={(e) => setV(e.target.value)}
            className="input-field"
            style={{ fontSize: '16px' }}
          />
          <input
            type="text"
            placeholder="Signature r (bytes32)"
            value={r}
            onChange={(e) => setR(e.target.value)}
            className="input-field"
            style={{ fontSize: '16px' }}
```

```
        />
        <input
          type="text"
          placeholder="Signature s (bytes32)"
          value={s}
          onChange={(e) => setS(e.target.value)}
          className="input-field"
          style={{ fontSize: '16px' }}
        />
        <button onClick={handleVerifyHealthID} className="action-button">
          Verify Health-ID
        </button>
        {verificationResult && <p className="result">{verificationResult}</p>}
      </div>
    </div>
  );
};

export default HealthIDVerification;
```

2) HealthIDVerification.css

```
.container {
  max-width: 700px;
  margin: 0 auto;
  padding: 20px;
  border: 1px solid #ddd;
  border-radius: 8px;
  box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
  background-color: #f9f9f9;
}

h2 {
  text-align: center;
  color: #333;
  margin-bottom: 20px;
}

.section {
  margin-bottom: 20px;
}

h3 {
  color: #444;
  margin-bottom: 10px;
}

.input-field {
  width: 100%;
  padding: 12px;
  margin-bottom: 10px;
  border: 1px solid #ccc;
  border-radius: 4px;
  font-size: 16px;
  box-sizing: border-box; /* Ensures the padding and border are included in the width */
```

```
}

.action-button {
   background-color: #4CAF50;
   color: white;
   padding: 12px 20px;
   border: none;
   border-radius: 4px;
   cursor: pointer;
   width: 100%;
   font-size: 16px;
   box-sizing: border-box; /* Ensures the button fits within the container */
}

.action-button:hover {
   background-color: #45a049;
}

.result {
   text-align: center;
   font-size: 16px;
   font-weight: bold;
   margin-top: 10px;
   color: #333;
}
```

PERFORMANCE METRICS

1) performance_metrics.py

```
import json
import time
from web3 import Web3
from random import choice, randint
from eth_utils import to_bytes  # Import to_bytes for conversion

# Connect to the Ganache blockchain
web3 = Web3(Web3.HTTPProvider('http://127.0.0.1:7545'))  # Replace with your provider if necessary

# Check connection
if web3.is_connected():
   print("Connected to Blockchain")
else:
   print("Connection failed")

# Load ABI and contract address from the JSON file
def load_contract_data():
   with open('HealthIDVerification.json') as f:  # Replace with the correct path to your contract JSON file
      contract_json = json.load(f)
      abi = contract_json['abi']  # ABI is under the 'abi' key
      contract_address = contract_json['networks']['5777']['address']  # Replace '5777' with your network ID (usually
Ganache is 5777)
   return abi, contract_address

abi, contract_address = load_contract_data()
contract = web3.eth.contract(address=contract_address, abi=abi)
```

```python
# Accounts
account = web3.eth.accounts[0]  # Default account

# Metrics
verification_times = []
false_positives = 0
false_negatives = 0

# Function to verify a Health-ID and measure time
def verify_health_id(health_id, message_hash, v, r, s, expected_validity):
    # Convert message_hash to bytes32
    message_hash_bytes32 = Web3.to_hex(message_hash).rjust(66, '0')  # Ensure it is 32 bytes
    start_time = time.time()  # Start the timer
    result = contract.functions.verifyHealthID(
        health_id,
        message_hash_bytes32,
        int(v),  # Ensure v is an integer
        to_bytes(r),  # Convert r to bytes32
        to_bytes(s)   # Convert s to bytes32
    ).call()
    end_time = time.time()  # End the timer

    # Calculate verification time
    verification_time = end_time - start_time
    verification_times.append(verification_time)

    # Check if the result matches the expected validity (True or False)
    if result != expected_validity:
        if expected_validity:
            global false_negatives
            false_negatives += 1
        else:
            global false_positives
            false_positives += 1

    return result

# Function to run multiple verifications and collect metrics
def evaluate_metrics(num_iterations=100):
    valid_health_id = "12345"
    private_key = '0xb105e5044dfd3f226c02cd551099914c4dd167db0c7ad8c012329a8c4195fed9'  # Example private key
    message_hash = web3.keccak(text=valid_health_id)  # Hash the valid health ID
    signed_message = web3.eth.account._sign_hash(message_hash, private_key=private_key)  # Sign the hashed message
    v, r, s = signed_message.v, signed_message.r, signed_message.s

    for _ in range(num_iterations):
        # Randomly select whether to verify a valid or invalid Health-ID
        if choice([True, False]):
            # Valid Health-ID case
            verify_health_id(valid_health_id, message_hash, v, r, s, expected_validity=True)
        else:
            # Invalid Health-ID case (use a random invalid health ID)
            invalid_health_id = str(randint(10000, 99999))
            invalid_hash = web3.keccak(text=invalid_health_id)
```

```
    verify_health_id(invalid_health_id, invalid_hash, v, r, s, expected_validity=False)

# Evaluate metrics over 100 test cases
evaluate_metrics(100)

# Output metrics
average_verification_time = sum(verification_times) / len(verification_times) if verification_times else 0
print(f"Average Verification Time: {average_verification_time:.6f} seconds")
print(f"False Positives: {false_positives}")
print(f"False Negatives: {false_negatives}")
```