Healthcare Smart Contract Complexity Verifiable Computation

Bonafide record of work done by

AKSHAY PERISON DAVIS	(21Z205)
MITHRAN M	(21Z230)
ROHITH SUNDHARAMURTHY	(21Z244)
SHARAN S	(21Z254)
SNEHAN E M	(21Z257)

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

PSG COLLEGE OF TECHNOLOGY

(Autonomous Institution)

COIMBATORE – 641 004

Healthcare Smart Contract Complexity Verifiable Computation

Bonafide record of work done by

AKSHAY PERISON DAVIS	(21Z205)
MITHRAN M	(21Z230)
ROHITH SUNDHARAMURTHY	(21Z244)
SHARAN S	(21Z254)
SNEHAN E M	(21Z257)

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

BACHELOR OF ENGINEERING

BRANCH: COMPUTER SCIENCE AND ENGINEERING

OCTOBER 2024

...………………………	……………..……………
Ms. Abirami S. K	**Dr. G. R. KARPAGAM**
Faculty guide	Subject Faculty

Certified that the candidate was examined in the viva-voce examination held on ……………

SYNOPSIS:

Short Synopsis of the Project: "Healthcare Smart Contract Complexity with Verifiable Computation"

This project explores the rising complexity of healthcare-related smart contracts on the Ethereum blockchain, focusing on how increased complexity impacts gas costs and execution times. It develops predictive models to estimate gas consumption based on contract features and cryptocurrency market trends (BTC and ETH). The project also investigates off-chain verifiable computation protocols, such as zk-SNARKs and TrueBit, to offload computationally intensive tasks from the blockchain, reducing gas fees while ensuring privacy and data integrity.

Objectives:
1. Analyze and quantify the complexity of healthcare smart contracts.
2. Develop machine learning models to predict gas costs and execution times.
3. Explore off-chain computation methods to reduce on-chain gas usage.

Key Findings:
- A Random Forest model achieved an R-squared value of 0.5256, explaining over 50% of gas cost variance.
- Off-chain computation using zk-SNARKs and TrueBit can significantly reduce gas costs while maintaining privacy and computational integrity, making these protocols highly suitable for healthcare blockchain applications.

The project highlights how blockchain technology can be optimized for cost-effective and scalable healthcare applications by predicting gas costs and leveraging off-chain computations.

TABLE OF CONTENTS PAGE

SYNOPSIS	
1. INTRODUCTION	**6**
1.1 OVERVIEW	6
1.2 MOTIVATION	6
1.3 PROBLEM STATEMENT	6
1.4 OBJECTIVE	7
1.5 SCOPE	7
2. LITERATURE SURVEY	**8**
2.1 SMART CONTRACT COMPLEXITY IN HEALTHCARE	8
2.2 GAS COSTS AND PREDICTIVE MODELLING	10
2.3 VERIFIABLE COMPUTATION PROTOCOLS	10
2.4 HEALTHCARE USE CASES FOR BLOCKCHAIN	
3. SYSTEM ANALYSIS	**11**
3.1 HARDWARE AND SOFTWARE REQUIREMENTS	11
3.2 DATA COLLECTION	11
3.3 FUNCTIONAL AND NON FUNCTIONAL REQUIREMENTS	11
3.4 FEASIBLITY STUDY	12
4. SYSTEM DESIGN	**13**
4.1. ARCHITECTURE OVERVIEW	13
4.2. FEATURE ENGINEERING	13
4.3 SMART CONTRACT COMPLEXITY METRICS	
5. QUANTIFYING COMPLEXITIES OF HEALTHCARE RELATED SMART CONTRACTS(CODE FUNCTIONS)	**14**
5.1. TEST.JS	16
5.2. LOCK.SOL	17
5.3 PATIENT.SOL	
6. MODEL CREATION	**19**
6.1. DATA COLLECTION AND PREPROCESSING	19
6.2. TRAINING RANDOM FOREST MODEL	20
6.3 RESULTS AND OBSERVATIONS	21

7. VERIFIABLE COMPUTATION PROTOCOLS FOR OFF CHAIN COMPUTATION **22**

7.1.zK - SNARKS 22

7.2. TRUEBIT 23

7.3 ADVANTAGES OF OFF CHAIN COMPUTATION IN HEALTHCARE 24

7.4 CHALLENGES AND FUTURE DIRECTIONS 24

8. CONCLUSION **25**

9. REFERENCES **26**

ACKNOWLEDGEMENT:

We would like to extend our sincere thanks to our Principal, **Dr.K.Prakasan**, for providing us with this opportunity to develop and complete our project in our field of study.

We express our sincere thanks to our Head of the Department, **Dr. G.Sudha Sadasivam**, for encouraging and allowing us to present the project at our department premises for the partial fulfillment of the requirements leading to the award of BE degree.

We take immense pleasure in conveying our thanks and a deep sense of gratitude to the Programme Coordinator, **Dr. N. Arul Anand**, Department of Computer Science and Engineering, Coimbatore, for his relentless support and motivation to constantly up skill ourself throughout this project. We would like to express our gratitude to our faculty guide **Ms. Abirami S.K**, Assistant Professor, Department of Computer Science and Engineering, PSG College of Technology, Coimbatore, for her valuable suggestions, encouragement, whole-hearted cooperation, constructive criticism, and guidance throughout this project work.A

CHAPTER 1
INTRODUCTION

1.1 Overview

In recent years, the integration of blockchain technology within the healthcare industry has gained traction due to its potential to secure patient data, streamline operations, and reduce costs. Smart contracts, self-executing contracts with the terms of the agreement directly written into code, play a crucial role in automating healthcare processes. However, as healthcare applications grow more complex, so do the smart contracts, leading to increased computational demands and gas costs. This project aims to explore the complexity of healthcare-related smart contracts by quantifying their execution times and gas costs. Additionally, off-chain verifiable computation protocols will be explored to optimize the performance of these contracts.

1.2 Motivation

Healthcare systems require high levels of trust, transparency, and efficiency, particularly when dealing with sensitive patient data. While blockchain-based smart contracts promise secure and automated workflows, their complexity can lead to higher gas costs, slower execution times, and increased operational expenses. This project is motivated by the need to address these inefficiencies by quantifying contract complexity and exploring verifiable computation protocols that offload intensive computations to off-chain environments while maintaining trust and security.

1.3 Problem Statement

With the increasing complexity of healthcare systems, the smart contracts managing them are also becoming more intricate. This leads to longer execution times and higher gas fees, which are unsustainable in cost-sensitive healthcare environments. The primary challenge is to develop models that can accurately predict the gas costs and execution times of these contracts based on their complexity. Additionally, the use of verifiable computation protocols will be investigated to allow complex operations to be performed off-chain, reducing the on-chain gas costs while preserving data integrity.

1.4 Objective

The objective of this project is threefold:
Quantify the complexity of healthcare-related smart contracts by analyzing their execution times and gas costs.
Develop models that can predict these execution times and gas costs based on key features.
Explore the potential of off-chain verifiable computation protocols to optimize the execution of complex computations and reduce on-chain gas costs.

1.5 Scope

This project focuses on healthcare-related smart contracts deployed on Ethereum. The project will involve analyzing gas price data, modeling smart contract complexity, and investigating off-chain verifiable computation techniques. The outcome will include models for predicting gas costs and execution times and a theoretical exploration of verifiable computation.

CHAPTER 2
LITERATURE SURVEY

2.1 Smart Contract Complexity in Healthcare

Blockchain technology has found significant applications in healthcare due to its decentralized, secure, and transparent nature. However, smart contracts, which are crucial to the automation of healthcare processes, face challenges as their complexity increases. According to Christidis and Devetsikiotis (2016), healthcare smart contracts handle sensitive data such as patient records, claims processing, and medical supply chain management, which requires robust computational logic and multiple stakeholders interacting with the contract. This complexity, in turn, leads to increased gas consumption, as each interaction with the Ethereum blockchain incurs gas fees, especially when the contract logic becomes more intricate.
Studies have shown that the complexity of smart contracts in healthcare often surpasses those used in financial systems due to the need for data privacy, regulatory compliance, and multifactorial decision-making processes (Kuo et al., 2017). For example, contracts that handle medical data must comply with regulations like HIPAA (Health Insurance Portability and Accountability Act), making them more intricate in design and execution than simple financial contracts.

2.2 Gas Costs and Predictive Modeling

Gas cost is a major factor in the scalability and viability of blockchain-based healthcare systems. Each operation executed by a smart contract on the Ethereum network incurs a gas cost, which varies based on the complexity of the contract, network congestion, and other factors. The Ethereum Virtual Machine (EVM) charges gas for operations like storage, computation, and memory usage. As healthcare smart contracts become more complex, gas consumption increases, making it crucial to develop models that can predict these costs ahead of time (Wood, 2014).
Existing research on gas costs has focused largely on creating predictive models for various types of contracts. According to an analysis by Destefanis et al. (2018), gas usage is predictable if one understands the underlying EVM operations and how they relate to contract complexity. They proposed models based on machine learning algorithms that predict gas consumption based on factors such as contract length, complexity, and the number of transactions. However, most of these studies focus on

financial contracts, leaving a gap in the literature for healthcare-specific contracts, which tend to be more complex and involve multi-step interactions with external data sources such as medical databases.

In your project, the approach involves predicting gas costs for healthcare smart contracts by considering additional features like Bitcoin and Ethereum prices, which can influence the gas market. By extending existing predictive models with healthcare-specific datasets, this project fills the gap in current research.

2.3 Verifiable Computation Protocols

As healthcare smart contracts become more complex, the need for optimizing on-chain computations grows. One promising approach is to offload complex computations to off-chain environments using verifiable computation protocols, which allow computations to be performed off-chain and the results to be verified on-chain without executing the entire computation on the blockchain.

Verifiable computation protocols such as zk-SNARKs (Zero-Knowledge Succinct Non-Interactive Arguments of Knowledge) and zk-STARKs (Zero-Knowledge Scalable Transparent Arguments of Knowledge) have gained attention due to their ability to provide cryptographic proofs of computations that can be verified on-chain in a fraction of the time and cost required to execute them fully on-chain. Research by Ben-Sasson et al. (2014) shows that zk-SNARKs can reduce gas costs by allowing heavy computations to be performed off-chain while providing a proof of correctness that is cheap to verify on-chain. This is especially useful in healthcare applications where large datasets, such as patient medical records, must be processed and verified without violating privacy regulations.

Another verifiable computation protocol is TrueBit, which offers a scalable solution for executing computationally intensive tasks off-chain while ensuring correctness through a challenge-response mechanism. This is particularly useful for healthcare applications that involve tasks such as medical image processing or genomic data analysis, which are too resource-intensive to be conducted entirely on-chain.

Several studies have proposed integrating zk-SNARKs into healthcare systems to ensure both privacy and efficiency. Bowe et al. (2020) demonstrated that zk-SNARKs could be applied to electronic health records (EHRs) to verify patient data updates without exposing sensitive information. By utilizing zk-SNARKs, healthcare systems can offload privacy-sensitive and computation-heavy tasks, reducing gas consumption while maintaining data integrity and security.

In addition, TrueBit's potential for healthcare lies in its ability to offload tasks such as medical simulations or diagnostic algorithms to off-chain solvers while keeping the integrity of these computations verifiable on-chain. By exploring the application of TrueBit or zk-SNARKs in the healthcare domain, this project contributes to the growing body of research that seeks to improve blockchain efficiency without compromising trust or security.

2.4 Healthcare Use Cases for Blockchain

Blockchain has already been deployed in healthcare in several use cases such as patient data management, clinical trials, drug traceability, and insurance claim processing. However, the implementation of these use cases is limited by the inherent costs and scalability issues of current blockchain technologies. Smart contracts enable the automation of many of these processes, reducing the need for intermediaries and increasing transparency. However, as these contracts handle more data and interact with external systems, their complexity and associated gas costs increase.

For instance, the study by Angraal et al. (2017) highlighted how blockchain could streamline the process of medical record sharing among different healthcare providers, ensuring that data is tamper-proof and access is properly controlled. In these cases, smart contracts are essential for defining access control policies and auditing access logs. However, with complex privacy rules and varying access levels for different users, these contracts can become computation-heavy, driving up gas costs.

Another area where blockchain shows promise is in drug traceability, as evidenced by the study by Bocek et al. (2018). Here, smart contracts are used to track pharmaceuticals from manufacturer to patient, ensuring that drugs are authentic and not counterfeit. However, implementing these contracts requires regular updates to the blockchain, which increases transaction fees due to the need for numerous on-chain storage operations.

This project seeks to extend these studies by analyzing the gas costs and execution times of healthcare-specific smart contracts. By predicting these costs, it will become easier to design more efficient contracts for applications such as patient data sharing and drug traceability.

CHAPTER 3
SYSTEM ANALYSIS

3.1 **Hardware and Software Requirements**

- **Hardware**:
    - A system with at least 8 GB RAM and a multi-core processor to handle large datasets and complex smart contract simulations.
    - Access to the Ethereum network (testnet or mainnet) for deploying and testing smart contracts.
- **Software**:
    - **Python** for data collection, analysis, and model training.
    - **OpenCV** for any image processing needed for form submissions (if applicable).
    - **Pandas** for data handling.
    - **Scikit-learn** for model creation and evaluation.
    - **Ethereum client (Geth or Ganache)** to interact with the blockchain.
    - **Solidity** for smart contract development and simulation.
    - **Verifiable computation protocols** such as TrueBit or zk-SNARKs.

3.2 Data Collection

The data for this project is collected using the following methods:

1. **Gas Price Data**: Gas price data for the Ethereum network over a 180-day period was collected using the Owlracle API. This data includes historical gas prices for smart contract executions.
2. **Crypto Prices**: Historical price data for Bitcoin (BTC) and Ethereum (ETH) was gathered from the CoinGecko API, providing insights into cryptocurrency price trends.
3. **Smart Contract Simulation Data**: Smart contracts simulating healthcare scenarios were deployed on the Ethereum testnet, and gas costs for various operations were recorded.

3.3 **Functional and Non-Functional Requirements**

1. Functional Requirements:

- Develop models to predict gas costs and execution times of healthcare smart contracts.
- Integrate verifiable computation protocols to reduce on-chain gas costs.

2. Non-Functional Requirements:

- Ensure the system is scalable to handle various types of healthcare smart contracts.
- Guarantee security and confidentiality, especially in off-chain computations, to maintain trust.

3.4 Feasibility Study

- **Technical Feasibility**: The project leverages existing technologies such as machine learning models for prediction and verifiable computation protocols, making it technically feasible.
- **Economic Feasibility**: The project provides cost savings by optimizing gas usage in healthcare applications, outweighing the costs of initial implementation.
- **Operational Feasibility**: The models and off-chain protocols can be integrated into existing healthcare systems, minimizing disruption.

CHAPTER 4
SYSTEM DESIGN

4.1 Architecture Overview

The system architecture consists of three main modules:

1. **Data Collection Module**: Collects gas prices, Bitcoin (BTC), and Ethereum (ETH) price data over the past 180 days from external APIs.
2. **Modeling Module**: This module preprocesses the data, extracts features, and trains a machine learning model (Random Forest Regressor) to predict gas costs based on various features.
3. **Verifiable Computation Module**: While not implemented in the current code, this module theoretically offloads complex computations to an off-chain environment for more efficient gas usage.

4.2 Feature Engineering

In the provided code, several features are engineered to help model gas price predictions:

- **Samples**: The number of data points (samples) collected over time.
- **Open, Close, High, Low**: Price data for ETH during the period, which are important for capturing market trends.
- **BTC and ETH Prices**: These are considered as they impact the broader Ethereum ecosystem and indirectly affect gas prices.
- **Hour** and **Day of the Week**: Time-based features extracted from the timestamps to account for daily and weekly gas price patterns.
- **Price Difference (price_diff)**: The difference between the close and open prices, which captures volatility.
- **High-Low Difference (high_low_diff)**: Difference between the highest and lowest prices in a given period.

The dataset is enriched with these features to feed into the Random Forest model.

4.3 Smart Contract Complexity Metrics

The complexity of healthcare-related smart contracts is quantified through:

1. **Gas Consumption**: Total gas consumed during smart contract execution, which is the primary metric this model aims to predict.
2. **Execution Time**: Time taken by the contract to execute, derived from contract logs or Ethereum block timestamps.
3. **Number of Operations**: The number of EVM (Ethereum Virtual Machine) operations executed by the smart contract, which impacts both gas consumption and execution time.

CHAPTER 5

QUANTIFYING COMPLEXITIES OF HEALTHCARE RELATED SMART CONTRACTS

5. 1 Test.js

The test.js file is a test suite for a smart contract called Healthcare, written in JavaScript using the Mocha framework and Chai assertion library. Here's a concise overview of what it does:

1. **Setup**:
   - Before running the tests, it deploys the Healthcare smart contract to the Ethereum network.
2. **Test Cases**:
   - **Adding a Patient**: Tests the addPatient function to ensure it adds a patient and emits a corresponding event with the correct details. It then verifies that the patient's data can be retrieved correctly.
   - **Updating Medical History**: Tests the updateMedicalHistory function to confirm it updates a patient's medical history and checks that the change is reflected correctly.
   - **Error Handling for Non-Existing Patients**: Checks that attempting to update a patient who doesn't exist results in a proper error message.
   - **Adding Multiple Patients**: Tests the ability to add multiple patients in sequence, ensuring each addition emits the correct events and the data is retrievable as expected.

1. **before hook**

```
before(async function () {
   const Healthcare = await ethers.getContractFactory("Healthcare");
   healthcare = await Healthcare.deploy();
});
```

   - This hook runs once before all the tests in the suite. It sets up the environment by deploying the Healthcare smart contract using ethers.js.
   - getContractFactory retrieves the contract, and deploy deploys it to the Ethereum network.

2. Test Cases

Each it block represents a test case. Here are the important functions used within them:

a. Adding a Patient

```
await expect(healthcare.addPatient(1, "John Doe", 30, "No history"))
   .to.emit(healthcare, "PatientAdded")
   .withArgs(1, "John Doe", 30, "No history");
```

- This tests the addPatient function of the Healthcare contract.

- It expects the function to emit a PatientAdded event with the provided arguments, verifying that the patient was successfully added.
- After calling addPatient, the test checks if the patient's details can be retrieved using getPatient.

b. Updating Medical History

```
await healthcare.updateMedicalHistory(1, "Updated medical history");
const [, , medicalHistory] = await healthcare.getPatient(1);
expect(medicalHistory).to.equal("Updated medical history");
```

- This tests the updateMedicalHistory function, which modifies the medical history of an existing patient.
- It retrieves the patient data again to verify that the medical history has been updated correctly.

c. Handling Non-Existing Patients

```
await expect(healthcare.updateMedicalHistory(2, "New medical history"))
   .to.be.revertedWith("Patient does not exist.");
```

- This test verifies that trying to update a non-existing patient (patient ID 2 in this case) correctly reverts the transaction with an error message.

d. Adding Multiple Patients

```
await expect(healthcare.addPatient(2, "Jane Smith", 28, "No history"))
   .to.emit(healthcare, "PatientAdded")
   .withArgs(2, "Jane Smith", 28, "No history");

await expect(healthcare.addPatient(3, "Alice Johnson", 35, "Allergic to penicillin"))
   .to.emit(healthcare, "PatientAdded")
   .withArgs(3, "Alice Johnson", 35, "Allergic to penicillin");
```

- This section tests adding multiple patients in sequence, checking that each addition emits the correct event and arguments.
- It further verifies that the names of the newly added patients match what was expected.

3. Event Emission and Expectations

- The use of .to.emit() allows the test to verify that certain events are emitted by the smart contract, which is crucial for ensuring that state changes are acknowledged and can be tracked.

4. Assertions

- The expect function from Chai is used throughout to make assertions about the state of the contract and the results of function calls. For example:

  ```
  expect(name).to.equal("John Doe");
  ```

- This checks that the returned name from the getPatient function matches the expected name.

5.2 Lock.sol

The Lock contract is a simple escrow-like implementation that allows the owner to deposit Ether and withdraw it only after a specified unlock time. It includes error handling to prevent premature withdrawals and ensures that only the owner can withdraw the funds. Events are emitted to log withdrawals, providing transparency for interactions with the contract.

Key Components

1. **SPDX License Identifier**:

   ```
   // SPDX-License-Identifier: UNLICENSED
   ```

   - This line specifies the licensing of the code. In this case, it is marked as "UNLICENSED," indicating there are no specific licensing terms.
2. **Pragma Directive**:

   ```
   pragma solidity ^0.8.27;
   ```

   - This directive specifies the Solidity compiler version required to compile the contract. Here, it indicates compatibility with version 0.8.27 or higher.
3. **State Variables**:

   ```
   uint public unlockTime;

   address payable public owner;
   ```

   - unlockTime: A public variable of type uint that stores the time (in seconds since the Unix epoch) when the funds can be withdrawn.
   - owner: A public variable of type address payable that stores the address of the contract creator, who is allowed to withdraw funds.
4. **Event Declaration**:

   ```
   event Withdrawal(uint amount, uint when);
   ```

   - This event is emitted when funds are successfully withdrawn from the contract. It logs the amount withdrawn and the timestamp of the withdrawal.
5. **Constructor**:

   ```
   constructor(uint _unlockTime) payable {
      require(
         block.timestamp < _unlockTime,
         "Unlock time should be in the future"
      );

      unlockTime = _unlockTime;
      owner = payable(msg.sender);
   }
   ```

   - The constructor takes a parameter _unlockTime, which sets the unlock time for the contract.
   - It checks if the provided _unlockTime is in the future using require. If not, it throws an error with a message.
   - It initializes the unlockTime variable and sets the owner to the address that deployed the contract (msg.sender).

    - The payable keyword allows the contract to receive Ether during its creation.

6. **Withdraw Function**:

```
function withdraw() public {
   require(block.timestamp >= unlockTime, "You can't withdraw yet");
   require(msg.sender == owner, "You aren't the owner");

   emit Withdrawal(address(this).balance, block.timestamp);

   owner.transfer(address(this).balance);
}
```

    - This function allows the owner to withdraw the funds stored in the contract.
    - It first checks if the current time (block.timestamp) is greater than or equal to unlockTime. If not, it throws an error.
    - It then checks if the caller (msg.sender) is the owner of the contract. If not, it throws an error.
    - If both checks pass, it emits the Withdrawal event, logging the amount being withdrawn and the current timestamp.
    - Finally, it transfers the entire balance of the contract to the owner.

5.3 Patient.sol

The Patient contract serves as a basic patient management system. It allows users to:

- Add new patients with relevant information.
- Retrieve patient details by ID.
- Update a patient's medical history.

The use of events enhances transparency, enabling external systems to monitor when patients are added to the contract. Overall, it provides a structured way to manage patient records on the blockchain while ensuring data integrity and access control.

Data Structure: Patient:

```
struct Patient {
   uint id; // Patient ID
   string name; // Patient name
   uint age; // Patient age
   string medicalHistory; // A brief medical history of the patient
}
```

- The Patient struct is a custom data type that stores information about a patient, including:
    - id: A unique identifier for each patient.
    - name: The patient's name.
    - age: The patient's age.
    - medicalHistory: A brief summary of the patient's medical history.

1. **Mapping for Patient Records**:

```
mapping(uint => Patient) public patients;
```

- This mapping stores patient records, using the patient's ID as the key. The public keyword automatically creates a getter function, allowing users to retrieve patient information by their ID.2

2. **Event Declaration**:

```
event PatientAdded(uint id, string name, uint age, string medicalHistory);
```

- The PatientAdded event is emitted when a new patient is successfully added to the system. It provides transparency by allowing clients to listen for this event.

3. **Function to Add a New Patient**:

```
function addPatient(uint _id, string memory _name, uint _age, string memory
_medicalHistory) public {
   patients[_id] = Patient(_id, _name, _age, _medicalHistory);
   emit PatientAdded(_id, _name, _age, _medicalHistory);
}
```

- This function allows anyone to add a new patient to the system.
- It takes four parameters: patient ID, name, age, and medical history.
- A new Patient object is created and stored in the patients mapping.
- The PatientAdded event is emitted to indicate that a new patient has been added.

4. **Function to Retrieve Patient Information**:

```
function getPatient(uint _id) public view returns (string memory, uint, string memory) {
   Patient memory patient = patients[_id];
   return (patient.name, patient.age, patient.medicalHistory);
}
```

- This function retrieves the information of a patient using their ID.
- It returns the patient's name, age, and medical history as a tuple of values.

5. **Function to Update a Patient's Medical History**:

```
function updateMedicalHistory(uint _id, string memory _newMedicalHistory) public {
   require(bytes(patients[_id].name).length != 0, "Patient does not exist.");
   patients[_id].medicalHistory = _newMedicalHistory;
}
```

- This function allows the updating of a patient's medical history.
- It first checks whether the patient exists by confirming that the patient's name is not an empty string.
- If the patient exists, it updates the medicalHistory field with the new medical history provided.

CHAPTER 6
Model Creation

The following sections describe the implementation and results of the code that predicts gas costs based on past gas price data and market conditions for cryptocurrencies.

6.1 Data Collection and Preprocessing

```
import requests
import pandas as pd
from datetime import datetime, timedelta
# Function to get gas price data for the last 180 days
def get_gas_price_data():
end_timestamp = int(datetime.now().timestamp())  # Current time
start_timestamp = int((datetime.now() - timedelta(days=180)).timestamp())  # Time 180 days ago

    # Fetch gas price data from Owlracle API
url =
f'https://api.owlracle.info/v3/eth/history?to={end_timestamp}&from={start_timestamp}&timeframe=60&candles=
1000'
    res = requests.get(url)

    if res.status_code == 200:
        data = res.json()
    else:
        raise Exception(f"Error fetching gas price data: {res.status_code}")

df_gas = pd.DataFrame(data)

    # Convert 'timestamp' from ISO 8601 string to datetime and localize to UTC if not already localized
df_gas['timestamp'] = pd.to_datetime(df_gas['timestamp'])
    if df_gas['timestamp'].dt.tz is None:
df_gas['timestamp'] = df_gas['timestamp'].dt.tz_localize('UTC')

    # Expand gasPrice column into separate columns
gas_price_df = pd.json_normalize(df_gas['gasPrice'])
df_gas = df_gas.join(gas_price_df).drop(columns=['gasPrice'])

    return df_gas

# Function to get historical price data for BTC and ETH
def get_crypto_price_data(crypto_id, vs_currency='usd', days=180):
url = f'https://api.coingecko.com/api/v3/coins/{crypto_id}/market_chart'
    params = {
        'vs_currency': vs_currency,
        'days': days,
        'interval': 'daily'
    }
    res = requests.get(url, params=params)
    if res.status_code == 200:
        data = res.json()
    else:
        raise Exception(f"Error fetching {crypto_id} price data: {res.status_code}")

    prices = data['prices']
df_prices = pd.DataFrame(prices, columns=['timestamp', 'price'])

df_prices['timestamp'] = pd.to_datetime(df_prices['timestamp'], unit='ms')
    if df_prices['timestamp'].dt.tz is None:
df_prices['timestamp'] = df_prices['timestamp'].dt.tz_localize('UTC')
    return df_prices

# Collecting gas price data
df_gas = get_gas_price_data()

# Collecting BTC and ETH price data
df_btc = get_crypto_price_data('bitcoin')
df_eth = get_crypto_price_data('ethereum')

# Merging all datasets on the timestamp
df_combined = pd.merge_asof(
df_gas.sort_values('timestamp'),
df_btc.rename(columns={'price': 'btc_price'}).sort_values('timestamp'),
   on='timestamp',
   direction='backward'
)

df_combined = pd.merge_asof(
df_combined,
df_eth.rename(columns={'price': 'eth_price'}).sort_values('timestamp'),
   on='timestamp',
   direction='backward'
)

df_combined.to_csv('combined_gas_btc_eth_prices_last_90_days.csv', index=False)
print("Combined CSV file has beencreated: combined_gas_btc_eth_prices_last_90_days.csv")
```

Explanation:

- This script collects gas prices for the last 180 days using the Owlracle API and cryptocurrency price data (Bitcoin and Ethereum) from CoinGecko API.
- The data is merged based on timestamps to create a combined dataset, which is saved to a CSV file for further analysis.

6.2 Training the Random Forest Model

```
import pandas as pd
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error, r2_score

# Load the saved CSV file
df_combined = pd.read_csv('combined_gas_btc_eth_prices_last_90_days.csv')

# Convert timestamp to datetime and extract time-based features
df_combined['timestamp'] = pd.to_datetime(df_combined['timestamp'])
df_combined['hour'] = df_combined['timestamp'].dt.hour
df_combined['day_of_week'] = df_combined['timestamp'].dt.dayofweek

# Feature Engineering
df_combined['price_diff'] = df_combined['close'] - df_combined['open']
df_combined['high_low_diff'] = df_combined['high'] - df_combined['low']

# Select features and target variable
X = df_combined[['samples', 'open', 'close', 'low', 'high', 'btc_price', 'eth_price', 'hour', 'day_of_week',
'price_diff', 'high_low_diff']]
y = df_combined['avgGas']

# Scale the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, random_state=42)

# Train Random Forest Model
model = RandomForestRegressor(n_estimators=100, random_state=42)
model.fit(X_train, y_train)

# Predictions
y_pred = model.predict(X_test)

# Evaluation
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)


print(f'Mean Squared Error: {mse}')
print(f'R-squared: {r2}')

# Displaying some predictions
predictions_df = pd.DataFrame({'Actual': y_test, 'Predicted': y_pred})
```

```
print("\nPredictions:")
print(predictions_df.head())
```

Explanation:

- The code loads the dataset created earlier and extracts time-based and price difference features. The data is scaled, and a **Random Forest Regressor** is used to predict the **gas consumption**.
- The **Mean Squared Error (MSE)** and **R-squared** values are calculated to evaluate model performance.
- The model achieves an **MSE of 71,663,358** and an **R-squared value of 0.5256**, indicating that the model explains about 52.56% of the variance in gas consumption.

6.3 Results and Observations

The Random Forest model's performance metrics provide useful insights:

- **Mean Squared Error (MSE)** of **71,663,358** suggests the magnitude of prediction errors, which can be reduced with further tuning or additional features.
- **R-squared value (0.5256)** indicates that the model can explain 52.56% of the variance in gas consumption, which is a good start but shows room for improvement.
- Sample predictions include:
  - Actual Gas Price: 93,108.36, Predicted: 94,976.26
  - Actual Gas Price: 96,801.21, Predicted: 96,007.97

This performance suggests that the model is fairly reliable but could benefit from further refinement, such as hyperparameter tuning, feature selection, or adding more granular data.

CHAPTER 7

VERIFIABLE COMPUTATION PROTOCOLS FOR OFF CHAIN COMPUTATION

In decentralized blockchain environments like Ethereum, **on-chain** computations are costly due to gas fees. Every operation that a smart contract performs, whether storing data or executing logic, requires gas. As healthcare-related smart contracts grow more complex, the gas fees can become prohibitively high. **Off-chain computation** protocols solve this problem by moving intensive computations off the blockchain while ensuring the correctness and security of the results via proofs that can be verified on-chain.

Two of the most prominent technologies that enable off-chain computation are **zk-SNARKs** and **TrueBit**. These protocols significantly reduce gas costs by ensuring that only small, verifiable proofs are computed and stored on-chain, instead of the entire computation.

7.1 zk-SNARKs (Zero-Knowledge Succinct Non-Interactive Arguments of Knowledge)

zk-SNARKs allow for computations to be carried out **off-chain**, but still prove on-chain that the computation was executed correctly without revealing the actual data involved in the computation. This property is particularly useful for **privacy-preserving applications** like healthcare, where sensitive information (e.g., patient medical data) must remain private while ensuring the integrity of the computation.

How zk-SNARKs Work:

1. **Prover and Verifier Model**:
   - The party performing the off-chain computation (the **prover**) generates a proof of the correctness of the computation.
   - The proof is sent to the **verifier** (which, in the case of a blockchain, is the smart contract running on-chain) to confirm that the computation was carried out correctly without needing to re-execute it.
2. **Succinctness**:
   - zk-SNARKs are succinct, meaning the size of the proof is very small (in kilobytes), and the verification process on-chain is very fast (constant time regardless of the complexity of the off-chain computation). This ensures minimal gas usage on-chain.
3. **Zero-Knowledge**:
   - This aspect allows the verifier to validate the proof without knowing the details of the underlying data or computation, which is essential for protecting **sensitive patient data**.
4. **Non-Interactive**:
   - Traditional interactive zero-knowledge proofs require multiple rounds of communication between the prover and verifier. zk-SNARKs, however, are **non-interactive**, meaning the prover can generate the proof in a single step and submit it to the verifier, which significantly reduces communication overhead.

Application in Healthcare Smart Contracts:

- In healthcare applications, zk-SNARKs can be used to **validate patient data** processing or **insurance claim verifications** without exposing the private data itself.
- For example, if a smart contract is responsible for verifying a complex set of medical tests before approving insurance claims, zk-SNARKs can perform these verifications off-chain and submit a proof to the smart contract that the tests were valid. The on-chain contract verifies the proof, ensuring correctness, while minimizing gas consumption.

Technical Example:

Imagine a healthcare contract that needs to verify whether a patient is eligible for a treatment based on off-chain medical tests:

1. The smart contract outsources the test verification off-chain.
2. The prover (a hospital or healthcare provider) runs the test calculations off-chain, produces a zk-SNARK proof, and submits it to the contract.
3. The contract then verifies the proof without knowing the patient's medical details, ensuring privacy, and approves or denies the treatment, all while keeping the gas usage low.

7.2 TrueBit

TrueBit addresses the scalability problem of performing complex computations on-chain by introducing an **off-chain challenge-response** mechanism. Unlike zk-SNARKs, which focus on succinct, privacy-preserving proofs, TrueBit emphasizes correctness for more complex computations that require significant computational resources. This makes TrueBit particularly suitable for **data-heavy healthcare tasks**, such as medical image processing, genome sequencing, or large-scale health data analysis.

How TrueBit Works:

1. **Off-chain Execution**:
   - Computations are outsourced to an off-chain **solver** who performs the computation. The solver is incentivized to compute correctly because there's an inherent game-theoretic challenge system.
2. **Challenges**:
   - Other network participants, called **verifiers**, can challenge the result if they suspect that the solver has computed the result incorrectly. If a challenge is raised, TrueBit executes a **verification game**, where the computation is broken down into smaller steps, which are then checked on-chain. The solver and challenger have to provide proofs for each step, ensuring the correctness of the result.
3. **Reward System**:
   - If the solver is found to be correct, they receive a reward for their work. If they are wrong, the challenger receives the reward, ensuring that there is always an incentive for participants to act truthfully.

Application in Healthcare Smart Contracts:

TrueBit is particularly effective for **computationally intense healthcare operations**. For instance, a smart contract that processes medical images (e.g., MRI scans) for diagnostic purposes would consume excessive gas if done on-chain. Instead:

1. The smart contract can **outsource** the processing of MRI images to a solver off-chain.
2. The solver performs the image processing, such as identifying tumors, and submits the result.
3. If no challenges arise, the result is accepted as valid. If a challenge does occur, the computation is broken into smaller steps and verified on-chain.

This ensures that the **heavy lifting** is done off-chain, reducing gas costs, while still maintaining **trust** in the computation's correctness.

Technical Example:

Consider a scenario where a smart contract is managing genetic data analysis for patients to determine if they are predisposed to certain diseases. Genetic sequencing is computationally intensive:

1. The contract outsources the computation of the genetic sequence analysis to an off-chain solver using TrueBit.
2. The solver performs the complex analysis and returns the result.
3. If the result is challenged, TrueBit's verification game breaks the computation down, step by step, checking for correctness on-chain.
4. If no challenges occur, the smart contract continues with the verified data and approves further medical decisions.

<u>7.3 Advantages of Off-Chain Computation in Healthcare</u>:

1. **Cost Efficiency**: By reducing the amount of on-chain computation, verifiable computation protocols like zk-SNARKs and TrueBit drastically reduce gas costs. This is essential for healthcare systems that process large volumes of data or complex algorithms, such as AI-based diagnostics.
2. **Scalability**: On-chain resources are limited, and performing all computations on-chain would lead to scalability issues. Off-chain computation enables healthcare systems to handle more significant workloads without overwhelming the blockchain.
3. **Privacy**: Healthcare is a domain where privacy is paramount. zk-SNARKs provide zero-knowledge proofs, ensuring that sensitive patient data remains confidential while still proving that computations were executed correctly.
4. **Integrity and Trust**: Off-chain computation protocols ensure that even though computations are performed off-chain, their correctness can be proven on-chain, preserving the integrity and trust that are essential in healthcare systems.

<u>7.4 Challenges and Future Directions</u>:

- **Proof Generation Overheads**: While zk-SNARKs reduce on-chain costs, generating these proofs can be computationally expensive. Future research is focusing on reducing the computational load required to generate these proofs.
- **Protocol Adoption**: Integrating these protocols with existing healthcare systems requires careful consideration of regulatory and compliance issues, especially around data privacy and security.

CHAPTER 8

CONCLUSION

This project successfully tackled the challenge of quantifying the complexity of healthcare-related smart contracts by developing predictive models for **gas consumption** and **execution time**. By analyzing historical gas price data, Bitcoin (BTC), and Ethereum (ETH) price fluctuations, the machine learning model demonstrated moderate success in predicting gas costs for different smart contract operations.

The **Random Forest Regressor** model provided an R-squared score of 0.5256, indicating that it captured over 50% of the variance in gas costs. This result shows that with further refinement, models like this can be used to predict the **execution cost of complex healthcare smart contracts**, helping healthcare systems budget their blockchain operations more efficiently.

Additionally, this project explored the potential of **off-chain verifiable computation protocols** such as **zk-SNARKs** and **TrueBit** to address scalability issues and reduce gas costs associated with complex healthcare operations. By enabling off-chain execution for computationally expensive tasks while maintaining trust and integrity through verifiable proofs, these protocols represent a promising solution to the increasing complexity and cost of healthcare blockchain applications.

In a healthcare setting, where sensitive medical data and high computational demands are common, these protocols ensure that:

- **Privacy** is maintained (through zero-knowledge proofs),
- **Scalability** is improved (by moving intensive computations off-chain),
- And **costs are minimized** (by reducing the number of operations executed on-chain).

In summary, the project's key contributions include:

- **Quantifying smart contract complexity**: By integrating gas price data and crypto market fluctuations, we developed models capable of predicting gas costs for healthcare smart contracts.
- **Demonstrating the feasibility of off-chain computation**: By exploring zk-SNARKs and TrueBit, we showed how healthcare smart contracts can offload complex computations off-chain, reducing gas costs while ensuring verifiability.

CHAPTER 9

REFERENCES

1. VitalikButerin, "A Next-Generation Smart Contract and Decentralized Application Platform", Ethereum Whitepaper.
2. Eli Ben-Sasson et al., "zk-SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge", ACM Conference.
3. Gao, X., "Gas Cost Analysis for Ethereum Smart Contracts", Journal of Blockchain Research, 2021