|  |  |
| --- | --- |
| **Programme Name:** M.Sc. Data Science Semester IV  **Total Credits: 02** | **Course Name:** Blockchain Technologies for Data Science Practical  **Total Marks: 50**  **University assessment: 50** |

**BLOCKCHAIN TECHNOLOGIES PRACTICALS**

**PREREQUISITE:**

Basic programming knowledge and command-line skills, Understanding of blockchain concepts and decentralized systems, Familiarity with data management and machine learning basics, Knowledge of security concepts and IoT principles.

**COURSE OUTCOME:**

* Ability to set up blockchain development environments, write and deploy smart contracts, and manage Hyperledger Fabric networks.
* Proficiency in using IPFS for decentralized storage and integrating blockchain to ensure data integrity in machine learning models.
* Capability to implement blockchain-based data provenance systems and develop decentralized data marketplaces.
* Skills to deploy chain-code on Hyperledger Fabric, create secure blockchain-based voting systems, and integrate blockchain with IoT for secure data management.

|  |  |  |  |
| --- | --- | --- | --- |
| **Course Code** | | **Course Title** | **Credits** |
| PSDSP615a | | Blockchain Technologies for Data Science Practical | 02 |
| **Note: -** Node.js, npm, Truffle, Ganache, MetaMask, Solidity, Git Bash, Docker, Hyperledger Fabric binaries, IPFS, TensorFlow, PyTorch, | | | |
| 1 | Installation of Ethereum, Truffle, Ganache, and other tools | | |
| 2 | Writing and deploying basic smart contracts on Ethereum | | |
| 3 | Configuring and running a Hyperledger Fabric network | | |
| 4 | Storing and retrieving data using IPFS | | |
| 5 | Using blockchain to ensure data integrity in ML models | | |
| 6 | Implementing a blockchain-based data provenance system | | |
| 7 | Developing a decentralized marketplace for data exchange | | |
| 8 | Writing and deploying chain-code on Hyperledger Fabric | | |
| 9 | Implementing a secure voting system using blockchain | | |
| 10 | Combining blockchain with IoT for secure data management | | |

**INDEX**

|  |  |  |  |
| --- | --- | --- | --- |
| **S. No** | **PROGRAM** | **PAGE NO** | **DATE OF EXECUTION** |
| 1 | Installation of Ethereum, Truffle, Ganache, and other tools |  |  |
| 2 | Writing and deploying basic smart contracts on Ethereum |  |  |
| 3 | Configuring and running a Hyperledger Fabric network |  |  |
| 4 | Storing and retrieving data using IPFS |  |  |
| 5 | Using blockchain to ensure data integrity in ML models |  |  |
| 6 | Implementing a blockchain-based data provenance system |  |  |
| 7 | Developing a decentralized marketplace for data exchange |  |  |
| 8 | Writing and deploying chain-code on Hyperledger Fabric |  |  |
| 9 | Implementing a secure voting system using blockchain |  |  |
| 10 | Combining blockchain with IoT for secure data management |  |  |

**PROGRAM 1: INSTALLATION OF ETHEREUM, TRUFFLE, GANACHE AND OTHER TOOLS**

**OBJECTIVE**

The objective of this lab exercise is to set up a complete development environment for Ethereum blockchain application development. This includes installing the core tools and frameworks such as:

* Ethereum client (Ganache as a personal blockchain for testing)
* Truffle Suite (development framework for smart contracts)
* Node.js and npm (JavaScript runtime and package manager)
* MetaMask (browser-based crypto wallet and blockchain interaction tool)
* Git Bash (command-line shell for Windows users)
* Other dependencies as required.

**KEY CONCEPTS**

1. Ethereum: A decentralized blockchain platform supporting smart contracts. It enables the creation of decentralized applications (DApps).
2. Ganache: A personal, local Ethereum blockchain designed for development and testing. It simulates the blockchain environment locally and allows rapid testing without real ETH costs.
3. Truffle: A popular Ethereum development framework that provides tools for compiling, deploying, and testing smart contracts.
4. Node.js & npm: Node.js is a JavaScript runtime environment; npm is its package manager used to install dependencies including Truffle.
5. MetaMask: A browser extension wallet that allows users to manage Ethereum accounts and interact with decentralized applications.
6. Git Bash: Provides a Unix-like shell environment on Windows, useful for running blockchain commands.
7. Docker (optional): Can be used to containerize blockchain networks such as Hyperledger Fabric but is not strictly necessary for Ethereum setups.

**INSTALLATION STEPS OVERVIEW**

* Install Node.js and npm: Required to run JavaScript-based blockchain tools.
* Install Ganache: Can be downloaded as a desktop application or installed via npm as Ganache CLI.
* Install Truffle: Installed globally using npm.
* Install MetaMask: Browser extension (Chrome/Firefox).
* Install Git Bash: (Windows only) for command-line interface.
* Verify installations and set up sample project.

**CODE: INSTALLATION OF ETHEREUM, TRUFFLE, GANACHE, AND OTHER TOOLS**

Below are the commands and instructions you can run in your terminal (Git Bash or any terminal with Node.js/npm installed) to verify the installations and initialize a basic Truffle project:

**STEP 1: VERIFY NODE.JS AND NPM INSTALLATION**

* node -v
* npm -v

This prints the installed versions of Node.js and npm. If not installed, download and install from <https://nodejs.org/>.

**STEP 2: INSTALL GANACHE CLI GLOBALLY (OPTIONAL IF YOU PREFER CLI)**

* npm install -g ganache

Alternatively, download the Ganache desktop app from <https://trufflesuite.com/ganache/>.

**STEP 3: INSTALL TRUFFLE GLOBALLY**

* npm install -g truffle
* truffle version # Verifies the installation – generates the version details if installed properly

**STEP 4: INITIALIZE A NEW TRUFFLE PROJECT**

Create a new folder for your project and navigate into it:

* mkdir MyEthereumProject
* cd MyEthereumProject
* truffle init

This sets up the basic Truffle project structure

**STEP 5: RUN GANACHE**

If using Ganache CLI:

* ganache

If using Ganache GUI, launch the app manually.

Ganache runs a local Ethereum blockchain on <http://127.0.0.1:7545> by default.

**STEP 6: CONFIGURE TRUFFLE TO CONNECT TO GANACHE**

Edit truffle-config.js to add the following network configuration:

networks: {

development: {

host: "127.0.0.1",

port: 7545,

network\_id: "\*" // Match any network id

}

}

**STEP 7: INSTALL METAMASK BROWSER EXTENSION**

* Go to Chrome or Firefox web store and install MetaMask.
* Create or import an Ethereum account.
* Connect MetaMask to the local Ganache network by adding a custom RPC network:
  + Network Name: Ganache Local
  + RPC URL: <http://127.0.0.1:7545>
  + Chain ID: 1337 (or check Ganache for actual chain ID)

**CODE EXPLANATION:**

* node -v and npm -v verify that Node.js and npm are installed, which are prerequisites for running Truffle and Ganache.
* npm install -g truffle installs the Truffle framework globally, allowing you to run the truffle command anywhere.
* truffle init sets up a new Ethereum project skeleton with folders for contracts, migrations, and tests.
* Ganache acts as a local blockchain for development and testing so you don’t need to use public testnets or mainnet.
* The network configuration in truffle-config.js tells Truffle to deploy contracts on your local Ganache blockchain.
* MetaMask serves as your wallet and connection interface to the Ethereum blockchain, allowing you to interact with your deployed smart contracts from the browser.

**PROGRAM 2: WRITING AND DEPLOYING BASIC SMART CONTRACTS ON ETHEREUM**

**OBJECTIVE**

The goal of this lab exercise is to introduce you to writing basic smart contracts in Solidity, the most widely used programming language for Ethereum smart contracts and deploying these contracts to a blockchain environment (local Ganache blockchain). This hands-on experience will help you understand the fundamentals of smart contract development, deployment, and interaction.

**KEY CONCEPTS**

1. **What is a Smart Contract?**

A smart contract is a self-executing program with the terms of the agreement directly written into code. It runs on a blockchain, ensuring transparency, immutability, and decentralized execution without relying on intermediaries.

Smart contracts can:

* Store and manage digital assets.
* Automate workflows based on pre-defined rules.
* Provide decentralized applications (DApps) functionality.

1. **Why Ethereum and Solidity?**

Ethereum is the leading blockchain platform designed specifically to run smart contracts and decentralized applications.

* Solidity is a statically-typed, contract-oriented programming language designed to target the Ethereum Virtual Machine (EVM).
* It has syntax similar to JavaScript and C++, making it approachable for developers with experience in these languages.

1. **Basic Structure of a Solidity Smart Contract**

A Solidity contract includes:

* Pragma directive: Specifies the Solidity compiler version.
* Contract definition: Like a class in OOP.
* State variables: Storage variables on the blockchain.
* Functions: Define behaviors and interactions.
* Modifiers and Events (for access control and logging, respectively).

**EXAMPLE SKELETON:**

pragma solidity ^0.8.0;

contract SimpleStorage {

uint storedData;

function set(uint x) public {

storedData = x;

}

function get() public view returns (uint) {

return storedData;

}

}

1. **CONTRACT DEPLOYMENT**

* Deploying a smart contract means publishing the contract bytecode on the blockchain so it can be executed and interacted with by users.
* Deployment costs gas (Ethereum transaction fee), but on local Ganache, gas is simulated and free.
* Truffle provides scripts and commands to automate deployment.

1. **DEVELOPMENT LIFECYCLE USING TRUFFLE**

* Write: Create Solidity contract files (.sol) inside the contracts directory.
* Compile: Convert Solidity code into bytecode and ABI using truffle compile.
* Deploy: Deploy contracts to a blockchain network via migration scripts.
* Test/Interact: Use Truffle Console, scripts, or frontends to interact with deployed contracts.

1. **Migration Scripts**

* Migration scripts are JavaScript files that handle deployment logic.
* Located in the migrations folder, they define which contracts to deploy and in what order.
* Example: 2\_deploy\_contracts.js deploys your smart contracts.

1. **Using Ganache for Testing**

* Ganache simulates the blockchain locally with pre-funded accounts.
* You can test smart contract deployment and function calls without spending real ETH.

1. **Gas and Transactions**

* Every operation on the Ethereum blockchain costs gas.
* Gas limits ensure that contracts don't run indefinitely.
* On local testnets like Ganache, gas is simulated to avoid real costs but understanding gas is critical for production.

1. **Interacting with Deployed Contracts**

* After deployment, contracts have an address.
* You interact with the contract by calling its functions using tools like Truffle Console, web3.js, or ethers.js.
* Functions can be:
  + **Read-only** (view/pure) — no state change, no gas cost.
  + **State-changing** — update state, cost gas.

1. **Security Considerations (Basic Awareness)**

* Avoid common pitfalls like integer overflow, re-entrancy attacks.
* Solidity version ^0.8.x has built-in overflow checks.
* Always validate inputs and manage access control.

**CODE: SIMPLESTORAGE CONTRACT DEPLOYMENT**

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract SimpleStorage {

uint private storedData;

// Function to set a value

function set(uint x) public {

storedData = x;

}

// Function to get the stored value

function get() public view returns (uint) {

return storedData;

}

}

**STEP 2: COMPILE THE CONTRACT**

Run the following command inside your project directory:

* truffle compile

This compiles the contract and generates the ABI and bytecode in the build/contracts folder.

**STEP 3: CREATE MIGRATION SCRIPT**

Create a file 2\_deploy\_simple\_storage.js inside the migrations folder with the following:

const SimpleStorage = artifacts.require("SimpleStorage");

module.exports = function (deployer) {

deployer.deploy(SimpleStorage);

};

**STEP 4: DEPLOY THE CONTRACT**

Make sure Ganache is running locally. Deploy the contract to Ganache by running:

* truffle migrate --network development

This will deploy the contract on your local blockchain.

**STEP 5: INTERACT WITH THE CONTRACT USING TRUFFLE CONSOLE**

Open the console:

* truffle console --network development

Then interact with your contract:

// Get deployed instance

const instance = await SimpleStorage.deployed();

// Set a value

await instance.set(42);

// Get the stored value

const value = await instance.get();

console.log(value.toString()); // Should output '42'

**CODE EXPLANATION:**

* The SimpleStorage contract contains a private state variable storedData.
* The set function updates storedData with the input value.
* The get function returns the current value of storedData.
* The migration script 2\_deploy\_simple\_storage.js tells Truffle to deploy the SimpleStorage contract to the configured network.
* The Truffle console allows you to interact with the deployed contract instance asynchronously.
* When you call set(42), it creates a transaction on the blockchain that modifies the contract’s state.
* When you call get(), it reads the stored value without consuming gas (because it’s a view function).
* The output 42 confirms the contract works as intended.

**PROGRAM 3: CONFIGURING AND RUNNING A HYPERLEDGER FABRIC NETWORK**

**OBJECTIVE**

The objective of this lab exercise is to understand how to set up and configure a private permissioned blockchain network using Hyperledger Fabric — an enterprise-grade, modular blockchain framework. You will learn the architecture, key components, and steps to launch a Fabric network, enabling secure, scalable, and permissioned decentralized applications.

**KEY CONCEPTS**

1. **WHAT IS HYPERLEDGER FABRIC?**

Hyperledger Fabric is an open-source blockchain framework hosted by the Linux Foundation, designed for enterprise use cases requiring permissioned access, scalability, and modularity.

* Unlike public blockchains (Ethereum, Bitcoin), Fabric networks are permissioned — participants are known and vetted.
* Fabric supports pluggable consensus mechanisms, private channels, and rich smart contract functionality called chaincode.
* It is widely used in supply chain, finance, healthcare, and government sectors.

1. **KEY COMPONENTS OF HYPERLEDGER FABRIC:**

* Peers: Nodes that maintain the ledger and run chaincode (smart contracts).
* Orderers (Ordering Service): Nodes that order transactions into blocks to ensure consistency and finality.
* Membership Service Provider (MSP): Manages identities and certificates for authentication.
* Channels: Private subnets of communication between network members allowing confidential transactions.
* Chaincode: Fabric’s smart contracts, typically written in Go, Java, or JavaScript.

1. **FABRIC NETWORK ARCHITECTURE OVERVIEW:**

* A Fabric network consists of one or more organizations.
* Each org has one or more peers.
* Organizations interact via channels.
* The ordering service sequences transactions to guarantee consistency across peers.
* Transactions go through a lifecycle: Proposal → Endorsement → Ordering → Validation → Commit.

1. **PREREQUISITES FOR SETTING UP A FABRIC NETWORK:**

* Docker & Docker Compose: Used to containerize peers, orderers, and other Fabric components.
* Fabric binaries and samples: Pre-built tools and scripts to help bootstrap networks.
* Cryptographic material: Certificates and keys generated via Fabric CA or cryptogen tool for identity management.
* CLI tools: For interacting with the network.

1. **STEPS TO CONFIGURE AND RUN A BASIC FABRIC NETWORK:**

Step 1: Download Fabric samples and binaries from the official Hyperledger Fabric repository.

Step 2: Generate cryptographic material and certificates using cryptogen or Fabric CA.

Step 3: Create channel artifacts including genesis block and channel configuration transactions.

Step 4: Launch network components (peers, orderers, CA, CLI) using Docker Compose files.

Step 5: Create and join a channel where organizations communicate.

Step 6: Install and instantiate chaincode on peers.

Step 7: Invoke and query chaincode functions to interact with the ledger.

1. **CHANNELS AND PRIVACY:**

* Channels enable private communication among a subset of network members.
* Each channel maintains its own ledger.
* This feature supports privacy and confidentiality in enterprise contexts.

1. **CHAINCODE (SMART CONTRACTS) IN FABRIC:**

* Chaincode defines the business logic and rules of transactions.
* Fabric endorses transactions based on chaincode execution results.
* Supports languages: Go (most common), Java, JavaScript (Node.js).

1. **ORDERER TYPES AND CONSENSUS:**

* Fabric supports various consensus mechanisms:
  + Solo (development/testing only)
  + Kafka (deprecated)
  + Raft (recommended for production)

1. **WHY HYPERLEDGER FABRIC FOR DATA SCIENCE?**

* Permissioned network suitable for handling sensitive data.
* Supports data provenance, immutability, and auditability.
* Modular architecture allows integration with existing enterprise systems and data pipelines.

1. **TOOLS AND UTILITIES:**

* Fabric CA: Certificate Authority for issuing digital identities.
* Fabric SDKs: For integrating Fabric network with external applications in various languages.
* CLI tools: For administrative tasks and chaincode lifecycle management.

**CODE: CONFIGURING AND RUNNING A HYPERLEDGER FABRIC NETWORK**

**PREREQUISITES:**

* Docker and Docker Compose installed and running
* cURL installed
* Git installed
* At least 8 GB RAM recommended

**STEP 1: DOWNLOAD FABRIC SAMPLES AND BINARIES**

Open your terminal and run:

* curl -sSL https://bit.ly/2ysbOFE | bash -s -- 2.5.1 1.5.2

This script downloads Fabric binaries, Docker images, and sample files. Versions: Fabric 2.5.1, Fabric CA 1.5.2 (adjust if needed).

**STEP 2: NAVIGATE TO FABRIC SAMPLES DIRECTORY**

* cd fabric-samples/test-network

This sample directory contains scripts and configs to launch a basic network.

**STEP 3: START THE TEST NETWORK**

Run the following command to launch the network with two organizations and a channel:

*./network.sh up createChannel -c mychannel -ca*

* up starts the network
* createChannel creates a channel named mychannel
* -ca enables the Certificate Authorities for identity management

**STEP 4: DEPLOY CHAINCODE (SAMPLE)**

Deploy the sample chaincode (basic asset transfer) with this command:

*./network.sh deployCC -ccn basic -ccp ../asset-transfer-basic/chaincode-go -ccl go*

* -ccn basic: chaincode name
* -ccp: path to chaincode source
* -ccl: chaincode language

**STEP 5: INTERACT WITH THE NETWORK USING CLI**

Use the peer CLI container to invoke or query chaincode:

**Query example:**

docker exec -it cli bash

# Query all assets

peer chaincode query -C mychannel -n basic -c '{"Args":["GetAllAssets"]}'

**STEP 6: TEAR DOWN NETWORK AFTER USE**

When done, shut down the network with:

./network.sh down

**EXPLANATION:**

* **Step 1:** The script downloads everything needed for Fabric setup including Docker images.
* **Step 3:** The network.sh script automates starting orderers, peers, and certificate authorities inside Docker containers, creating a channel for communication.
* **Step 4:** Deploys chaincode (smart contract) to peers in the network.
* **Step 5:** Uses Fabric CLI inside a container to invoke chaincode functions, allowing you to test transactions.
* **Step 6:** Stops and cleans up all running containers to free resources.

This approach uses the official Fabric sample test network for rapid setup and testing, which is ideal for learning and prototyping.

**PROGRAM 4: STORING AND RETRIEVING DATA USING IPFS**

**OBJECTIVE**

This lab exercise aims to familiarize you with the InterPlanetary File System (IPFS) — a decentralized peer-to-peer storage protocol. You will learn how to store files on IPFS, retrieve them using content-based addressing, and understand its integration potential with blockchain for secure, immutable, and distributed data storage.

**KEY CONCEPTS:**

1. **WHAT IS IPFS?**

IPFS is a distributed file system protocol designed to make the web faster, safer, and more open. Instead of addressing data by location (like HTTP URLs), IPFS addresses data by its content hash.

* It forms a decentralized network where peers store and share content.
* Files are split, hashed, and stored across nodes.
* Content is immutable and verifiable via its cryptographic hash.

1. **CONTENT ADDRESSING VS LOCATION ADDRESSING**

* HTTP (Location-based): Data is fetched from a specific server address (e.g., https://example.com/file.txt).
* IPFS (Content-based): Data is fetched using a content identifier (CID), a unique hash derived from the file's content (e.g., Qm...).

This means if the content changes, the CID changes, ensuring integrity and version control.

1. **HOW IPFS WORKS:**

* When a file is added to IPFS, it is broken into blocks.
* Each block is hashed using a cryptographic hash function.
* The hash tree (Merkle DAG) links all blocks, producing a unique CID.
* The file is distributed across IPFS nodes.
* Users request content by CID; IPFS locates and retrieves it from any node holding it.

1. **IPFS NODES AND NETWORK**

* Each user runs an IPFS node.
* Nodes connect to peers in a distributed hash table (DHT).
* Files can be pinned to ensure they stay available on your node.
* Popular files propagate widely.

1. **IPFS USE CASES IN BLOCKCHAIN**

* Store large or off-chain data (e.g., datasets, media) securely.
* Blockchain stores the CID for reference; actual data is stored on IPFS.
* Enhances scalability by keeping the blockchain lightweight.
* Ensures data integrity — any tampering changes the CID, revealing corruption.

1. **INSTALLING AND USING IPFS**

* Go-IPFS is the reference implementation.
* IPFS can be run locally or accessed via public gateways.
* Files can be added via command line or API calls.

1. **BASIC IPFS COMMANDS**

* ipfs init — Initialize local IPFS node.
* ipfs daemon — Start IPFS node to connect to the network.
* ipfs add <file> — Add file to IPFS, returning CID.
* ipfs cat <CID> — Retrieve content by CID.
* ipfs pin add <CID> — Pin content locally.

1. **INTEGRATING IPFS WITH APPLICATIONS**

* IPFS APIs enable programmatic file upload and retrieval.
* Commonly used with decentralized applications (DApps) alongside blockchain.
* Tools like js-ipfs and ipfs-http-client allow integration with JavaScript.

1. **SECURITY AND PRIVACY**

* Content on IPFS is public and immutable.
* Sensitive data should be encrypted before adding.
* Access control must be implemented at application level.

1. **LIMITATIONS AND CHALLENGES**

* Data availability depends on nodes pinning content.
* No native incentive layer like blockchain tokens (though Filecoin is related).
* Gateway availability and latency may vary.

**CODE: STORING AND RETRIEVING DATA USING IPFS**

**STEP 1: INSTALL IPFS**

* Download the latest Go-IPFS release for your OS from:  
  <https://dist.ipfs.io/#go-ipfs>
* Extract the archive and move the binary to your system’s PATH (e.g., /usr/local/bin on Linux/macOS or add to Environment Variables on Windows).

**STEP 2: INITIALIZE IPFS NODE**

Open your terminal and run:

*ipfs init*

This initializes a new IPFS repository in your home directory.

Start the node and connect to the IPFS network:

*ipfs daemon*

This will keep running, connecting your node to peers.

**STEP 4: ADD A FILE TO IPFS**

Open another terminal window and run:

*ipfs add example.txt*

* Replace example.txt with the path to your file.
* The command outputs a **CID**, for example:  
  added QmXoypizjW3WknFiJnKLwHCnL72vedxjQkDDP1mXWo6uco

**STEP 5: RETRIEVE A FILE FROM IPFS USING CID**

Fetch the file content using:

ipfs cat QmXoypizjW3WknFiJnKLwHCnL72vedxjQkDDP1mXWo6uco > output.txt

This downloads the content of the file identified by the CID and saves it as output.txt.

**STEP 6: PIN THE FILE LOCALLY (OPTIONAL)**

To ensure the file stays available on your node:

ipfs pin add QmXoypizjW3WknFiJnKLwHCnL72vedxjQkDDP1mXWo6uco

**PROGRAMMATIC EXAMPLE: USING IPFS HTTP CLIENT (JAVASCRIPT)**

import { create } from 'ipfs-http-client';

const ipfs = create({ url: 'http://localhost:5001/api/v0' });

async function addAndRetrieve() {

const { cid } = await ipfs.add('Hello IPFS!');

console.log('Added file CID:', cid.toString());

const chunks = [];

for await (const chunk of ipfs.cat(cid)) {

chunks.push(chunk);

}

const content = Buffer.concat(chunks).toString();

console.log('Retrieved content:', content);

}

addAndRetrieve();

**CODE EXPLANATION:**

* ipfs init sets up your local IPFS node’s repository.
* ipfs daemon launches your node to participate in the IPFS network.
* ipfs add uploads a file, returning a CID which uniquely identifies the content.
* ipfs cat fetches content from the network using the CID.
* Pinning keeps the content persistently available on your node.
* The JavaScript example shows how to add and retrieve data programmatically via IPFS HTTP API, enabling integration in apps.

**PROGRAM 5: USING BLOCKCHAIN TO ENSURE DATA INTEGRITY IN MACHINE LEARNING MODELS**

**OBJECTIVE:**

This lab exercise focuses on how blockchain technology can be leveraged to ensure data integrity, provenance, and auditability in machine learning (ML) pipelines. You will understand how blockchain’s immutable ledger can provide trustworthy data management crucial for building reliable, transparent, and tamper-proof ML models.

**KEY CONCEPTS:**

1. **Data Integrity in Machine Learning**

* ML models heavily depend on the quality and integrity of training data.
* Data tampering, corruption, or unauthorized modifications can lead to biased, inaccurate, or malicious models.
* Verifying data provenance and ensuring data immutability is critical for trustworthy AI.

1. **Blockchain as a Trust Layer**

* Blockchains are decentralized, immutable ledgers where each transaction is cryptographically linked and verified.
* Once data is recorded on a blockchain, it cannot be altered without network consensus.
* This property makes blockchain ideal for recording **proofs of data authenticity** and **provenance metadata**.

1. **How Blockchain Ensures Data Integrity**

* Data or its hash can be recorded on the blockchain, creating a permanent, tamper-evident audit trail.
* Each data submission is timestamped and linked to its source.
* Subsequent ML model training steps can reference these blockchain entries to verify input data validity.

1. **Workflow Integration**

Typical workflow to ensure ML data integrity using blockchain:

* Data Collection: Raw data is collected from trusted sources.
* Hashing Data: Generate cryptographic hashes (fingerprints) of datasets or data batches.
* Blockchain Recording: Store the hashes, timestamps, and metadata as transactions on the blockchain.
* Model Training: ML pipelines reference these hashes to confirm the training data has not changed.
* Audit and Verification: Any party can verify data integrity by comparing dataset hashes against the blockchain records.

1. **Advantages**

* Immutability: Blockchain records are unalterable.
* Transparency: All stakeholders can access the data provenance.
* Decentralization: No single point of failure or control.
* Auditability: Enables compliance with data regulations and ethical AI standards.

1. **Technical Considerations**

* Storing large datasets directly on-chain is infeasible; instead, **store hashes on-chain** and actual data off-chain (e.g., IPFS or secure databases).
* Use smart contracts to automate validation, access control, or alerts on data inconsistencies.
* Integrate blockchain verification steps into ML pipelines to ensure only verified data is used.

1. **Blockchain Platforms and Tools**

* Ethereum or Hyperledger Fabric can be used depending on the use case.
* Smart contracts automate data registration and verification.
* Oracles may be used to feed off-chain data or verification status to the blockchain.

1. **Example Use Cases**

* Healthcare ML models ensuring patient data integrity.
* Financial fraud detection models with verifiable transaction data.
* Supply chain demand forecasting with tamper-proof sensor data.

1. **Challenges**

* Latency and throughput limits of blockchains.
* Balancing privacy and transparency (sensitive data should not be exposed on-chain).
* Integration complexity between ML workflows and blockchain infrastructure.

1. **Future Trends**

* Combining blockchain with federated learning for secure, distributed ML.
* Use of zero-knowledge proofs for privacy-preserving data verification.
* Integration with trusted execution environments (TEEs) for enhanced security.

**CODE: USING BLOCKCHAIN TO ENSURE DATA INTEGRITY IN MACHINE LEARNING MODELS**

**STEP 1: SOLIDITY SMART CONTRACT TO STORE DATA HASHES**

Create a file DataIntegrity.sol in the contracts folder:

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract DataIntegrity {

// Event emitted when a new data hash is registered

event DataHashRegistered(address indexed sender, bytes32 dataHash, uint256 timestamp);

// Mapping of data hashes to timestamp of registration

mapping(bytes32 => uint256) public dataHashes;

// Register a new data hash

function registerDataHash(bytes32 \_dataHash) public {

require(dataHashes[\_dataHash] == 0, "Hash already registered");

dataHashes[\_dataHash] = block.timestamp;

emit DataHashRegistered(msg.sender, \_dataHash, block.timestamp);

}

// Verify if a data hash is registered

function verifyDataHash(bytes32 \_dataHash) public view returns (bool) {

return dataHashes[\_dataHash] != 0;

}

// Get timestamp of when the data hash was registered

function getTimestamp(bytes32 \_dataHash) public view returns (uint256) {

require(dataHashes[\_dataHash] != 0, "Hash not registered");

return dataHashes[\_dataHash];

}

}

**STEP 2: DEPLOY THE CONTRACT**

Create migration script 2\_deploy\_data\_integrity.js:

const DataIntegrity = artifacts.require("DataIntegrity");

module.exports = function (deployer) {

deployer.deploy(DataIntegrity);

};

Deploy using:

truffle migrate --network development

**STEP 3: NODE.JS SCRIPT TO HASH DATA AND REGISTER ON BLOCKCHAIN**

INSTALL DEPENDENCIES:

npm install crypto web3

Create registerDataHash.js:

const fs = require('fs');

const crypto = require('crypto');

const Web3 = require('web3');

const contractABI = require('./build/contracts/DataIntegrity.json').abi;

const contractAddress = 'YOUR\_DEPLOYED\_CONTRACT\_ADDRESS'; // Replace with deployed contract address

async function main() {

// Read dataset file

const data = fs.readFileSync('dataset.csv');

// Compute SHA-256 hash of the dataset

const hash = crypto.createHash('sha256').update(data).digest('hex');

console.log('Data SHA-256 Hash:', hash);

// Connect to local blockchain (Ganache)

const web3 = new Web3('http://127.0.0.1:7545');

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

const account = accounts[0];

// Instantiate contract

const contract = new web3.eth.Contract(contractABI, contractAddress);

// Register hash on blockchain

const receipt = await contract.methods.registerDataHash('0x' + hash).send({ from: account, gas: 300000 });

console.log('Transaction receipt:', receipt.transactionHash);

}

main().catch(console.error);

**STEP 4: VERIFYING DATA INTEGRITY**

To verify a data file’s integrity later:

* Compute its SHA-256 hash.
* Query the smart contract’s verifyDataHash method to check registration.
* Compare timestamp or presence for validation.

**CODE EXPLANATION:**

* The Solidity contract stores hashes of datasets as bytes32 keys mapped to timestamps.
* The registerDataHash function adds a new hash with the current block timestamp, preventing duplicates.
* The Node.js script hashes the data file off-chain using SHA-256 and then records this hash on-chain.
* Later, the same hash can be verified to ensure the dataset has not been modified since registration.
* This workflow integrates blockchain immutability with ML data integrity guarantees.

**PROGRAM 6: IMPLEMENTING A BLOCKCHAIN-BASED DATA PROVENANCE SYSTEM**

**OBJECTIVE:**

This lab exercise aims to develop an understanding of how blockchain technology can be used to implement a data provenance system—a system that tracks the origin, history, and lifecycle of data throughout its existence. Provenance ensures trust, accountability, and transparency in data management, which is essential for sensitive or regulated domains.

**Key Concepts:**

1. **What is Data Provenance?**

* Data provenance refers to the detailed history of data, including where it originated, how it was created, processed, modified, and by whom.
* It helps establish data authenticity, lineage, and audit trails.
* Provenance is critical for verifying data quality, compliance, and reproducibility.

1. **Challenges in Traditional Provenance Systems**

* Centralized logging and tracking systems are vulnerable to tampering.
* Lack of transparency and trust among multiple stakeholders.
* Difficulty in auditing data modifications retrospectively.

1. **Why Use Blockchain for Data Provenance?**

* **Immutability:** Records once written cannot be altered or deleted, ensuring trustworthy provenance.
* **Decentralization:** Multiple parties can share and verify provenance data without relying on a single trusted authority.
* **Transparency:** Blockchain provides an auditable trail accessible to authorized stakeholders.
* **Smart Contracts:** Automate the capture, validation, and querying of provenance information.

1. **Provenance Data Model on Blockchain**

* Each data event (creation, modification, access) is recorded as a transaction.
* Provenance records include metadata such as:
  + Data identifier (e.g., hash or URI)
  + Timestamp
  + Actor (user or system identity)
  + Operation type (create, update, delete)
  + Context or description

1. **Implementing Provenance with Blockchain**

* Define a smart contract to record provenance events.
* Store critical metadata on-chain; large data or files stored off-chain with references.
* Allow querying of provenance history for any data item.
* Use access controls to protect sensitive provenance details.

1. **Integration with Data Pipelines**

* Instrument data ingestion and processing steps to emit provenance events.
* Automate blockchain recording within the workflow.
* Use blockchain as a source of truth for data audits and compliance checks.

1. **Use Cases**

* Scientific research data reproducibility.
* Healthcare records lifecycle management.
* Supply chain product origin and processing history.
* Data marketplaces verifying provenance before transactions.

1. **Privacy and Scalability Considerations**

* Sensitive data must not be stored directly on-chain; use encryption or off-chain storage with on-chain references.
* Use permissioned blockchains like Hyperledger Fabric for enterprise privacy.
* Implement efficient querying mechanisms to handle provenance data growth.

1. **Tools and Technologies**

* Blockchain platforms: Ethereum (public), Hyperledger Fabric (permissioned).
* Off-chain storage: IPFS, cloud storage with hashes stored on-chain.
* APIs and SDKs to connect data systems with blockchain.

**CODE: IMPLEMENTING A BLOCKCHAIN-BASED DATA PROVENANCE SYSTEM**

**STEP 1: SOLIDITY SMART CONTRACT FOR DATA PROVENANCE**

Create a file DataProvenance.sol in the contracts folder:

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract DataProvenance {

struct ProvenanceEvent {

address actor;

uint256 timestamp;

string operation; // e.g., "CREATE", "UPDATE", "DELETE"

string description;

}

// Mapping from data identifier hash to array of provenance events

mapping(bytes32 => ProvenanceEvent[]) private provenanceRecords;

// Event emitted when a provenance event is recorded

event ProvenanceRecorded(bytes32 indexed dataId, address indexed actor, string operation, uint256 timestamp);

// Record a new provenance event for a data item

function recordEvent(bytes32 dataId, string memory operation, string memory description) public {

ProvenanceEvent memory newEvent = ProvenanceEvent({

actor: msg.sender,

timestamp: block.timestamp,

operation: operation,

description: description

});

provenanceRecords[dataId].push(newEvent);

emit ProvenanceRecorded(dataId, msg.sender, operation, block.timestamp);

}

// Get provenance event count for a data item

function getEventCount(bytes32 dataId) public view returns (uint256) {

return provenanceRecords[dataId].length;

}

// Retrieve provenance event by index for a data item

function getProvenanceEvent(bytes32 dataId, uint256 index) public view returns (

address actor,

uint256 timestamp,

string memory operation,

string memory description

) {

require(index < provenanceRecords[dataId].length, "Index out of bounds");

ProvenanceEvent storage evt = provenanceRecords[dataId][index];

return (evt.actor, evt.timestamp, evt.operation, evt.description);

}

}

**STEP 2: MIGRATION SCRIPT**

Create 2\_deploy\_data\_provenance.js in migrations:

const DataProvenance = artifacts.require("DataProvenance");

module.exports = function (deployer) {

deployer.deploy(DataProvenance);

};

Deploy with:

truffle migrate --network development

**STEP 3: EXAMPLE INTERACTION USING TRUFFLE CONSOLE**

const instance = await DataProvenance.deployed();

const dataId = web3.utils.sha3("dataset\_v1.csv");

// Record a creation event

await instance.recordEvent(dataId, "CREATE", "Initial upload of dataset v1");

// Record an update event

await instance.recordEvent(dataId, "UPDATE", "Cleaned missing values");

// Get number of events

const count = await instance.getEventCount(dataId);

console.log("Total provenance events:", count.toString());

// Retrieve and display all events

for (let i = 0; i < count; i++) {

const evt = await instance.getProvenanceEvent(dataId, i);

console.log(`Event ${i}: Actor=${evt.actor}, Operation=${evt.operation}, Timestamp=${new Date(evt.timestamp \* 1000).toISOString()}, Description=${evt.description}`);

}

**CODE EXPLANATION:**

* DataProvenance contract records multiple provenance events per data item, identified by a hash (dataId).
* Each event stores the actor’s address, timestamp, operation type, and a descriptive note.
* Events are stored in arrays mapped to the data identifier, enabling full lifecycle tracking.
* Functions allow recording new events and retrieving event history by index.
* Example interaction shows how to record and query provenance events in a real blockchain environment.
* This smart contract provides an immutable, auditable provenance ledger accessible to all network participants.

**PROGRAM 7: DEVELOPING A DECENTRALIZED MARKETPLACE FOR DATA EXCHANGE**

**OBJECTIVE:**

This lab exercise aims to explore the design and implementation of a decentralized marketplace for data exchange using blockchain technology. You will learn how blockchain enables secure, transparent, and trustless buying, selling, and sharing of datasets without relying on centralized intermediaries.

**KEY CONCEPTS:**

1. **What is a Decentralized Data Marketplace?**

* A platform where data providers and consumers transact directly over a blockchain network.
* Removes middlemen, reducing costs and censorship risks.
* Participants retain control over their data and transactions are transparent and verifiable.

1. **Why Blockchain for Data Marketplaces?**

* Trustless Transactions: Smart contracts enforce agreements automatically.
* Transparency: All transactions and ownership changes are recorded on an immutable ledger.
* Data Provenance: Buyers can verify data authenticity and history.
* Tokenization: Digital tokens can represent data assets or payment means.
* Access Control: Cryptographic techniques protect data access rights.

1. **Marketplace Components:**

* Data Listings: Metadata describing datasets for sale (size, type, quality).
* Smart Contracts: Facilitate listing, bidding, payment, and delivery.
* Payment Mechanisms: Often use cryptocurrencies or tokens.
* Access Management: Controls who can download or use data post-purchase.
* Reputation Systems: Evaluate trustworthiness of sellers and buyers.

1. **Architecture Overview:**

* On-chain Components:
  + Smart contracts handling listings, bids, payments, and dispute resolution.
  + Token contracts for payments or incentives.
* Off-chain Components:
  + Actual data storage on decentralized storage (e.g., IPFS).
  + User interfaces (web/mobile apps).
  + Data encryption and key management.

1. **Workflow Example:**

* Seller uploads dataset to IPFS and pins it.
* Seller creates a listing on blockchain marketplace contract with dataset CID and price.
* Buyer browses listings, chooses dataset, and sends payment via smart contract.
* Upon payment confirmation, buyer receives decryption key or access rights.
* Smart contract releases funds to seller.
* Both parties may rate the transaction to build reputation.

1. **Smart Contract Functions:**

* listData(dataCID, price, metadata)
* buyData(listingId)
* confirmDelivery(listingId)
* withdrawFunds()
* rateTransaction(listingId, rating)

1. **Challenges:**

* Ensuring data privacy while sharing on a public blockchain.
* Managing off-chain data storage reliability.
* Handling disputes and refunds fairly.
* Scalability for large user bases and datasets.
* Regulatory and compliance considerations.

1. **Technologies and Standards:**

* Ethereum or other smart contract platforms.
* IPFS/Filecoin for decentralized storage.
* ERC-721 or ERC-1155 tokens for data asset representation (NFTs).
* Decentralized identity (DID) solutions for user authentication.

1. **Use Cases:**

* Scientific data sharing marketplaces.
* IoT sensor data exchanges.
* Healthcare data sharing with patient consent.
* Financial data subscription services.

1. **Future Directions:**

* Integration with AI for data quality scoring.
* Automated smart contract-based licensing.
* Cross-chain interoperability for asset portability.

**CODE: DEVELOPING A DECENTRALIZED MARKETPLACE FOR DATA EXCHANGE**

**Step 1: Solidity Smart Contract**

Create DataMarketplace.sol in the contracts folder:

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract DataMarketplace {

struct Listing {

address payable seller;

string dataCID; // IPFS CID of the dataset

uint256 price; // Price in wei

bool sold;

}

Listing[] public listings;

event DataListed(uint256 listingId, address indexed seller, string dataCID, uint256 price);

event DataPurchased(uint256 listingId, address indexed buyer, uint256 price);

// List a new dataset for sale

function listData(string memory \_dataCID, uint256 \_price) public {

require(\_price > 0, "Price must be positive");

listings.push(Listing(payable(msg.sender), \_dataCID, \_price, false));

emit DataListed(listings.length - 1, msg.sender, \_dataCID, \_price);

}

// Purchase a listed dataset by sending exact price

function buyData(uint256 listingId) public payable {

require(listingId < listings.length, "Invalid listing");

Listing storage listing = listings[listingId];

require(!listing.sold, "Already sold");

require(msg.value == listing.price, "Incorrect payment");

listing.sold = true;

listing.seller.transfer(msg.value);

emit DataPurchased(listingId, msg.sender, msg.value);

}

// Get total number of listings

function getListingCount() public view returns (uint256) {

return listings.length;

}

// Get listing details

function getListing(uint256 listingId) public view returns (

address seller,

string memory dataCID,

uint256 price,

bool sold

) {

require(listingId < listings.length, "Invalid listing");

Listing storage listing = listings[listingId];

return (listing.seller, listing.dataCID, listing.price, listing.sold);

}

}

**Step 2: Migration Script**

Create 2\_deploy\_data\_marketplace.js:

const DataMarketplace = artifacts.require("DataMarketplace");

module.exports = function (deployer) {

deployer.deploy(DataMarketplace);

};

Deploy using:

truffle migrate --network development

**Step 3: Example Usage via Truffle Console**

const marketplace = await DataMarketplace.deployed();

// Seller lists a dataset

await marketplace.listData("QmExampleCID12345", web3.utils.toWei("0.1", "ether"), { from: accounts[0] });

// Buyer purchases the dataset

await marketplace.buyData(0, { from: accounts[1], value: web3.utils.toWei("0.1", "ether") });

// Check listing details

const listing = await marketplace.getListing(0);

console.log(listing);

**CODE EXPLANATION:**

* The contract maintains an array of Listing structs representing datasets for sale.
* Sellers list data by providing an IPFS CID and a price.
* Buyers purchase by sending the exact amount of Ether; payment is immediately transferred to the seller.
* Events notify off-chain apps of listing and purchase actions.
* This simple contract does not handle access control or encrypted data delivery; those are handled off-chain or in extended implementations.

**PROGRAM 8: WRITING AND DEPLOYING CHAINCODE ON HYPERLEDGER FABRIC**

**OBJECTIVE:**

This lab exercise aims to teach you how to write, package, install, and instantiate **chaincode** (smart contracts) on a Hyperledger Fabric network. Chaincode defines the business logic and transaction rules that run on the Fabric peers, enabling decentralized applications with trusted workflows.

**KEY CONCEPTS:**

1. **What is Chaincode?**

* Chaincode is Fabric’s equivalent of smart contracts.
* It is a program, typically written in Go, Java, or JavaScript (Node.js), executed by peers to validate and update the ledger state.
* Chaincode handles transaction proposals and endorses them according to predefined logic.

1. **Chaincode Lifecycle**

Fabric v2.x introduces a new lifecycle process:

* **Packaging:** Chaincode is packaged into a tarball.
* **Installation:** Installed on peers.
* **Approval:** Each organization approves the chaincode definition.
* **Commitment:** The chaincode is committed to the channel.
* **Instantiation:** The chaincode is ready for use.

This process ensures decentralized governance and versioning.

1. **Chaincode Structure**

A typical chaincode has:

* **Init function:** Initialization logic called during instantiation.
* **Invoke function:** Handles different transaction functions.
* **State interactions:** Uses the ledger APIs to read/write key-value pairs.
* **Error handling and validation.**

1. **Writing Chaincode (Go Example)**

* Use the Fabric Chaincode Shim API to interact with the ledger.
* Define functions for create, update, delete, query operations.
* Manage composite keys for complex data.

1. **Packaging Chaincode**

* Chaincode source code is packaged into a .tar.gz format.
* Metadata specifies name, version, language, and endorsement policy.

1. **Installing Chaincode**

* Chaincode package is installed on each peer using CLI or SDK.
* Installation verifies code integrity and prepares it for endorsement.

1. **Approving Chaincode**

* Each org must approve the chaincode definition specifying version, sequence, endorsement policy, and collections config.

1. **Committing Chaincode**

* Once all necessary approvals are obtained, the chaincode is committed to the channel.
* This activates the chaincode for transactions.

1. **Invoking Chaincode**

* Clients submit transaction proposals to invoke chaincode functions.
* Endorsing peers simulate and sign proposals.
* Orderer packages endorsed transactions into blocks.
* Peers validate and commit transactions to the ledger.

1. **Testing and Debugging**

* Use Fabric test networks or development environments.
* Logs and lifecycle commands help debug deployment and execution.

1. **Use Cases**

* Asset transfer and tracking.
* Supply chain provenance.
* Healthcare data management.
* Decentralized finance.

1. **Tools**

* Fabric samples repo includes example chaincode.
* Fabric CLI, SDKs, and APIs facilitate interaction.
* VSCode extensions support chaincode development.

**CODE PART: BASIC GO CHAINCODE EXAMPLE AND DEPLOYMENT STEPS**

**Step 1: Write Basic Chaincode in Go**

Create a new folder asset-transfer-basic and inside it, create chaincode.go:

package main

import (

"encoding/json"

"fmt"

"github.com/hyperledger/fabric-contract-api-go/contractapi"

)

// AssetTransferContract defines the Smart Contract structure

type AssetTransferContract struct {

contractapi.Contract

}

// Asset represents a simple asset with ID and Value

type Asset struct {

ID string `json:"ID"`

Value string `json:"Value"`

}

// InitLedger initializes ledger with some assets

func (c \*AssetTransferContract) InitLedger(ctx contractapi.TransactionContextInterface) error {

assets := []Asset{

{ID: "asset1", Value: "100"},

{ID: "asset2", Value: "200"},

}

for \_, asset := range assets {

assetJSON, err := json.Marshal(asset)

if err != nil {

return err

}

err = ctx.GetStub().PutState(asset.ID, assetJSON)

if err != nil {

return fmt.Errorf("failed to put asset %s to world state: %v", asset.ID, err)

}

}

return nil

}

// CreateAsset adds a new asset to the ledger

func (c \*AssetTransferContract) CreateAsset(ctx contractapi.TransactionContextInterface, id string, value string) error {

exists, err := c.AssetExists(ctx, id)

if err != nil {

return err

}

if exists {

return fmt.Errorf("asset %s already exists", id)

}

asset := Asset{

ID: id,

Value: value,

}

assetJSON, err := json.Marshal(asset)

if err != nil {

return err

}

return ctx.GetStub().PutState(id, assetJSON)

}

// ReadAsset returns the asset stored in the ledger with given id

func (c \*AssetTransferContract) ReadAsset(ctx contractapi.TransactionContextInterface, id string) (\*Asset, error) {

assetJSON, err := ctx.GetStub().GetState(id)

if err != nil {

return nil, fmt.Errorf("failed to read asset %s from world state: %v", id, err)

}

if assetJSON == nil {

return nil, fmt.Errorf("asset %s does not exist", id)

}

var asset Asset

err = json.Unmarshal(assetJSON, &asset)

if err != nil {

return nil, err

}

return &asset, nil

}

// AssetExists returns true when asset with given ID exists in world state

func (c \*AssetTransferContract) AssetExists(ctx contractapi.TransactionContextInterface, id string) (bool, error) {

assetJSON, err := ctx.GetStub().GetState(id)

if err != nil {

return false, err

}

return assetJSON != nil, nil

}

func main() {

contract := new(AssetTransferContract)

chaincode, err := contractapi.NewChaincode(contract)

if err != nil {

panic(fmt.Sprintf("Error creating chaincode: %v", err))

}

if err := chaincode.Start(); err != nil {

panic(fmt.Sprintf("Error starting chaincode: %v", err))

}

}

**Step 2: Package and Deploy Chaincode**

Use Fabric tools and CLI to:

* Package chaincode:

peer lifecycle chaincode package asset\_transfer.tar.gz --path ./asset-transfer-basic --lang golang --label asset\_transfer\_1

* Install on peers, approve, commit (requires Fabric CLI and proper environment variables set).

For a test network, you can use scripts from Fabric samples (test-network folder) to deploy the sample chaincode.

**Step 3: Invoke and Query Chaincode**

Use Fabric CLI commands or SDKs to:

* Initialize ledger:

peer chaincode invoke -C mychannel -n asset\_transfer -c '{"function":"InitLedger","Args":[]}'

* Create asset:

peer chaincode invoke -C mychannel -n asset\_transfer -c '{"function":"CreateAsset","Args":["asset3","300"]}'

* Read asset:

peer chaincode query -C mychannel -n asset\_transfer -c '{"function":"ReadAsset","Args":["asset3"]}'

**CODE EXPLANATION:**

* The Go chaincode defines an AssetTransferContract with basic CRUD functions for assets.
* InitLedger seeds the ledger with initial data.
* Chaincode interacts with the ledger state through PutState and GetState.
* The Fabric chaincode shim API handles transaction contexts and world state.
* Packaging and lifecycle commands deploy the chaincode to the Fabric network.
* Invokes modify ledger state; queries read data without modification.

**PROGRAM 9: IMPLEMENTING A SECURE VOTING SYSTEM USING BLOCKCHAIN**

**OBJECTIVE:**

This lab exercise focuses on designing and implementing a **secure, transparent, and tamper-proof voting system** using blockchain technology. The aim is to leverage blockchain’s immutable ledger and cryptographic properties to ensure election integrity, voter privacy, and trust in the electoral process.

**KEY CONCEPTS:**

1. **Why Use Blockchain for Voting?**

* **Transparency:** All votes are recorded on a public, immutable ledger accessible to stakeholders.
* **Immutability:** Once recorded, votes cannot be altered or deleted, preventing fraud.
* **Decentralization:** Removes the need for a central trusted authority.
* **Verifiability:** Voters can independently verify their votes.
* **Security:** Cryptographic techniques ensure voter anonymity and data integrity.

1. **Key Requirements of a Secure Voting System**

* **Eligibility:** Only authorized voters can cast votes.
* **Privacy:** Votes are confidential; voter identities must be protected.
* **Integrity:** Votes must be recorded exactly as cast.
* **Uniqueness:** Each voter can vote only once.
* **Auditability:** The system provides a verifiable audit trail without revealing voter identities.

1. **Blockchain Voting Architecture**

* **Voter Registration:** Voter identities and credentials are registered, typically off-chain with cryptographic proofs stored on-chain.
* **Casting Votes:** Voters submit encrypted or anonymized votes as blockchain transactions.
* **Vote Storage:** Votes stored immutably in the blockchain ledger.
* **Vote Counting:** Transparent tallying can be automated via smart contracts.
* **Result Verification:** Publicly verifiable results without compromising privacy.

1. **Smart Contract Role**

* Enforce voting rules (eligibility, one vote per voter).
* Record votes securely.
* Manage election lifecycle (start, end, tally).
* Publish results on-chain.

1. **Privacy and Anonymity Techniques**

* **Zero-Knowledge Proofs (ZKP):** Prove voter eligibility without revealing identity.
* **Ring Signatures / Mixnets:** Obfuscate the link between voter and vote.
* **Homomorphic Encryption:** Allows tallying encrypted votes without decryption.

1. **Challenges**

* Balancing transparency and privacy.
* Scalability to handle large voter bases.
* Prevention of coercion and vote-buying.
* User-friendly interfaces and accessibility.

1. **Platforms and Tools**

* Ethereum and other smart contract platforms.
* Hyperledger Fabric for permissioned voting networks.
* Cryptographic libraries supporting ZKP and encryption.

1. **Real-World Examples**

* Estonia’s e-voting system leveraging blockchain.
* West Virginia’s pilot mobile voting app.
* Research projects combining blockchain with advanced cryptography.

1. **Testing and Validation**

* Rigorous security audits.
* Simulations and testnets.
* Independent verifiers and observers.

**CODE PART: SIMPLE VOTING SMART CONTRACT EXAMPLE**

**Step 1: Solidity Smart Contract**

Create Voting.sol in the contracts folder:

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract Voting {

struct Candidate {

uint id;

string name;

uint voteCount;

}

mapping(address => bool) public voters;

mapping(uint => Candidate) public candidates;

uint public candidatesCount;

address public admin;

event VoteCast(address indexed voter, uint indexed candidateId);

modifier onlyAdmin() {

require(msg.sender == admin, "Only admin can perform this action");

\_;

}

constructor() {

admin = msg.sender;

}

// Add candidates by admin

function addCandidate(string memory \_name) public onlyAdmin {

candidatesCount++;

candidates[candidatesCount] = Candidate(candidatesCount, \_name, 0);

}

// Cast vote for a candidate

function vote(uint \_candidateId) public {

require(!voters[msg.sender], "You have already voted");

require(\_candidateId > 0 && \_candidateId <= candidatesCount, "Invalid candidate");

voters[msg.sender] = true;

candidates[\_candidateId].voteCount++;

emit VoteCast(msg.sender, \_candidateId);

}

// Get vote count of a candidate

function getVoteCount(uint \_candidateId) public view returns (uint) {

require(\_candidateId > 0 && \_candidateId <= candidatesCount, "Invalid candidate");

return candidates[\_candidateId].voteCount;

}

}

**Step 2: Migration Script**

Create 2\_deploy\_voting.js:

const Voting = artifacts.require("Voting");

module.exports = function (deployer) {

deployer.deploy(Voting);

};

Deploy with:

truffle migrate --network development

**Step 3: Example Usage via Truffle Console**

const voting = await Voting.deployed();

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

// Admin adds candidates

await voting.addCandidate("Alice", { from: accounts[0] });

await voting.addCandidate("Bob", { from: accounts[0] });

// Voters cast votes

await voting.vote(1, { from: accounts[1] });

await voting.vote(2, { from: accounts[2] });

// Check vote counts

const votesAlice = await voting.getVoteCount(1);

const votesBob = await voting.getVoteCount(2);

console.log(`Alice has ${votesAlice} votes`);

console.log(`Bob has ${votesBob} votes`);

**CODE EXPLANATION:**

* The contract maintains a list of candidates with vote counts.
* The admin (deployer) can add candidates before voting begins.
* Each Ethereum address can vote once, enforced by the voters mapping.
* Votes increment the selected candidate’s count.
* The contract emits an event on each vote for transparency.
* This simple model lacks anonymity and advanced privacy but demonstrates core voting logic.

**PROGRAM 10: COMBINING BLOCKCHAIN WITH IOT FOR SECURE DATA MANAGEMENT**

**OBJECTIVE:**

This lab exercise aims to explore how blockchain technology can be integrated with Internet of Things (IoT) systems to ensure secure, tamper-proof, and transparent management of IoT data. You will understand the challenges of IoT data security and how blockchain’s decentralized ledger can provide solutions.

**KEY CONCEPTS:**

1. **IoT Data Challenges**

* Massive volumes of data generated by distributed IoT devices.
* Data integrity risks due to device vulnerabilities and potential tampering.
* Centralized IoT architectures face single points of failure and trust issues.
* Need for reliable audit trails and provenance for IoT sensor data.

1. **Why Blockchain for IoT Data Management?**

* Decentralization: Eliminates single points of failure.
* Immutability: Ensures data recorded is tamper-proof.
* Transparency and Auditability: All transactions logged and verifiable.
* Trustless Environment: Devices can interact without centralized trust.
* Automated Agreements: Smart contracts enable automatic actions on predefined conditions.

1. **Architecture for Blockchain-IoT Integration**

* IoT Devices: Sensors, actuators collecting and sending data.
* Edge Gateways: Aggregate and preprocess data; interface with blockchain.
* Blockchain Network: Stores hashes or summaries of IoT data and controls access.
* Smart Contracts: Automate data validation, alerting, and workflows.
* Off-chain Storage: Due to data volume, raw data often stored off-chain (e.g., IPFS), with blockchain storing references.

1. **Data Flow**

* IoT device captures data → data sent to gateway → data hashed and/or encrypted → hash and metadata sent to blockchain → data stored off-chain → smart contracts triggered for rules (e.g., alerts).

1. **Security Benefits**

* Detect tampering by comparing on-chain hash with off-chain data.
* Authenticate devices via blockchain-based identity management.
* Enable secure firmware updates via blockchain verification.

1. **Consensus and Scalability Considerations**

* Lightweight consensus mechanisms or permissioned blockchains suit IoT constraints.
* Edge computing reduces latency and bandwidth needs.
* Layer-2 solutions and sidechains can help scale.

1. **Use Cases**

* Supply chain monitoring with sensor data provenance.
* Smart grids and energy management.
* Healthcare wearables securely sharing data.
* Environmental monitoring with trusted data logging.

1. **Challenges**

* Resource constraints on IoT devices.
* Latency and throughput of blockchain.
* Privacy concerns for sensitive IoT data.
* Complex integration and standardization.

1. **Technologies**

* Ethereum, Hyperledger Fabric, IOTA (specialized for IoT).
* IPFS and other decentralized storage.
* Blockchain SDKs tailored for IoT.

**CODE PART: BASIC EXAMPLE OF IOT DATA HASH REGISTRATION ON ETHEREUM BLOCKCHAIN**

**Step 1: Solidity Smart Contract to Register IoT Data Hashes**

Create IoTDataRegistry.sol in the contracts folder:

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract IoTDataRegistry {

struct DataRecord {

bytes32 dataHash;

uint256 timestamp;

address device;

}

DataRecord[] public records;

event DataRegistered(bytes32 indexed dataHash, uint256 timestamp, address indexed device);

// Register IoT data hash on blockchain

function registerData(bytes32 \_dataHash) public {

DataRecord memory record = DataRecord({

dataHash: \_dataHash,

timestamp: block.timestamp,

device: msg.sender

});

records.push(record);

emit DataRegistered(\_dataHash, block.timestamp, msg.sender);

}

// Get total records count

function getRecordsCount() public view returns (uint256) {

return records.length;

}

// Get a data record by index

function getDataRecord(uint256 index) public view returns (bytes32, uint256, address) {

require(index < records.length, "Index out of bounds");

DataRecord storage record = records[index];

return (record.dataHash, record.timestamp, record.device);

}

}

**Step 2: Migration Script**

Create 2\_deploy\_iot\_data\_registry.js:

const IoTDataRegistry = artifacts.require("IoTDataRegistry");

module.exports = function (deployer) {

deployer.deploy(IoTDataRegistry);

};

Deploy with:

truffle migrate --network development

**Step 3: Node.js Script to Simulate IoT Device Registering Data Hash**

Install dependencies:

npm install web3 crypto

Create registerIoTData.js:

const Web3 = require('web3');

const crypto = require('crypto');

const fs = require('fs');

const contractABI = require('./build/contracts/IoTDataRegistry.json').abi;

const contractAddress = 'YOUR\_DEPLOYED\_CONTRACT\_ADDRESS'; // replace with actual

async function registerIoTData(filePath) {

// Read IoT data file

const data = fs.readFileSync(filePath);

// Compute SHA-256 hash of data

const hash = crypto.createHash('sha256').update(data).digest('hex');

console.log('Computed data hash:', hash);

// Connect to local Ethereum node

const web3 = new Web3('http://127.0.0.1:7545');

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

const deviceAccount = accounts[0];

const contract = new web3.eth.Contract(contractABI, contractAddress);

// Register hash on blockchain

const receipt = await contract.methods.registerData('0x' + hash).send({ from: deviceAccount, gas: 300000 });

console.log('Data registered in transaction:', receipt.transactionHash);

}

registerIoTData('sensor\_data.json').catch(console.error);

**CODE EXPLANATION:**

* The IoTDataRegistry smart contract records hashes of IoT data submissions with timestamps and device addresses.
* Each IoT device (simulated by Ethereum accounts) registers data hashes immutably on-chain.
* The Node.js script simulates an IoT device reading sensor data, hashing it, and registering the hash on Ethereum.
* This provides a tamper-proof, verifiable record of IoT data collection events.
* Actual sensor data can be stored off-chain (e.g., IPFS), referenced securely via the blockchain hash.