	SHRI SANT GAJANAN MAHARAJ COLLEGE OF ENGG, SHEGAON				**LABORATORY MANUAL**	
	PRACTICAL EXPERIMENT INSTRUCTION SHEET					
	EXPERIMENT TITLE: Studying Distributed Ledger Technology and its related sub-techniques, along with installing the LTS Version of Ubuntu O.S. on VirtualBox in Windows 10 for virtualization.					
EXPERIMENT NO. SSGMCE/WI/CSE/01/8KS06/01			ISSUE NO. : 00		ISSUE DATE : 01-10-2024	
REV. DATE :	REV. NO. : 00	DEPTT. : COMPUTER SCIENCE AND ENGINEERING				
LABORATORY : EMERGING TECHNOLOGY LAB VI (DLT)				SEMESTER : VIII		PAGE :1 OF 4

AIM: Studying Distributed Ledger Technology and its related sub-techniques, along with installing the LTS Version of Ubuntu O.S. on VirtualBox in Windows 10 for virtualization.

OBJECTIVE:

- Grasp the fundamental principles of Distributed Ledger Technology (DLT)
- Analyze the components of DLT, including blockchain, consensus mechanisms, smart contracts, and cryptography.
- Acquire foundational knowledge in virtualization and understand its utility in establishing isolated computing environments.
- Deploy VirtualBox on a Windows 10 platform to facilitate virtualization for operating systems and understand installation process of Ubuntu operating system within the VirtualBox environment.
- Implement the installation and configuration of tools associated with Distributed Ledger Technology on the Ubuntu virtual machine.

SCOPE:

The scope of this experiment is broad, covering both theoretical aspects of Distributed Ledger Technology and hands-on experience with virtualization, with the ultimate goal of preparing for practical applications in DLT development.

FACILITIES:

Computer System, Virtual Box, Ubuntu Setup.

THEORY/EXPERIMENT PROCEDURE:

This practical involves two main components:

1. Study of Distributed Ledger Technology (DLT) and Sub-Techniques:
2. Installation of Ubuntu OS (LTS Version) on VirtualBox in Windows 10**:**

Distributed Ledger Technology (DLT):

Distributed Ledger Technology (DLT) is a digital system designed to record asset transactions simultaneously across multiple locations. Unlike traditional databases, distributed ledgers lack a central data store or administrative function.

DLT is a broader term that encompasses various methods of recording and validating transactions across multiple locations or participants. "Blockchain" is a specific type of DLT, distinguished by its use of a chain of blocks to store transactions.

DLT emerged as a concept before blockchain gained prominence. The idea behind DLT is to create a decentralized and distributed system for secure and transparent transaction recording, challenging the traditional centralized databases. The term DLT was used to describe the broader landscape of technologies and protocols that enable distributed and decentralized ledgers whereas Blockchain, introduced through Bitcoin, became a popular and well-known implementation of DLT

Originally introduced by Bitcoin, DLT, commonly known as blockchain technology, has become a prominent term in the tech world due to its potential across industries. Simply put, DLT revolves around the concept of a "decentralized" network, challenging the conventional "centralized" mechanism. It is anticipated to have significant implications for sectors and entities that have traditionally relied on a trusted third party.

In essence, Distributed Ledger Technology (DLT) is a decentralized and distributed system that ensures secure and transparent transaction recording across multiple participants or nodes in a network. Unlike traditional centralized databases, where a single authority controls the database, DLT operates on a peer-to-peer network, enabling participants to share and synchronize a common record of information.

Below are some of the types of DLT

1. **Blockchain:** In this type of DLT, transactions are stored in the form chain of blocks and each block produces a unique hash that can be used as proof of valid transactions. Each node has a copy of the ledger which makes it more transparent.
2. **Directed Acyclic Graphs (DAG):** This uses a different data structure to organize the data that brings more consensus. In this type of DLT, validation of transactions mostly requires the majority of support from the nodes in the network. Every node on the network has to provide proof of transactions on the ledger and then can initiate transactions. In this nodes have to verify at least two of the previous transactions on the ledger to confirm their transaction.
3. **Hashgraph:** Here records are stored in the form of a directed acyclic graph. It uses a different consensus mechanism, using virtual voting as the form consensus mechanism for gaining network consensus. Hence nodes do not have to validate each transaction on the network.
4. **Holochain:** Holochain is termed as the next level of blockchain by some people because it is much more decentralized than blockchain. It is a type of DLT that simply proposes that each node will run on a chain of its own. Therefore nodes or miners have the freedom to operate autonomously. It basically moves to the agent-centric structure. Here agent means computer, node, miner, etc.
5. **Tempo or Radix:** Tempo uses the method of making a partition of the ledger this is termed sharding and then all the events that happened in the network are ordered properly. Basically, transactions are added to the ledger on basis of the order of events than the timestamp.

S**ub-techniques associated with DLT:**

- **Blockchain Technology:** Most well-known sub-technique, blockchain is a decentralized and distributed ledger that utilizes cryptographic techniques to secure transactions and create an immutable chain of blocks..
- **Consensus Mechanisms:** These are protocols used to achieve agreement among nodes in a DLT network regarding the validity of transactions. Common consensus mechanisms include Proof of Work (PoW), Proof of Stake (PoS), and Practical Byzantine Fault Tolerance (PBFT)
- **Smart Contracts:** Self-executing contracts with the terms of the agreement directly written into code. Smart contracts automatically enforce and execute the terms when predefined conditions are met, eliminating the need for intermediaries.
- **Permissioned and Permissionless Ledgers:** DLT systems can be categorized as either permissioned or permissionless. Permissioned ledgers restrict access to authorized participants, while permissionless ledgers allow anyone to participate in the network.
- **Hash Functions:** Cryptographic hash functions are used to ensure the integrity of data in each block. They generate a fixed-size output (hash) based on the input data, making it difficult to alter the data without changing the hash.
- **Cryptographic Signatures:** Digital signatures play a crucial role in ensuring the authenticity of

transactions. Participants use cryptographic signatures to prove ownership of their private keys and validate transactions.

- **Interoperability Protocols:** As DLT systems evolve, interoperability becomes essential for different networks to communicate and share information seamlessly. Interoperability protocols aim to establish standards for communication and data exchange between disparate DLT networks.
- **Off-Chain Scaling Solutions:** Techniques such as state channels and sidechains are employed to address scalability issues by moving some transactions off the main blockchain, reducing congestion and improving performance.

Installation of Ubuntu OS (LTS Version) on VirtualBox in Windows 10:

Step by step Procedure and explanation

Step 1: Download VirtualBox

Visit the official VirtualBox website: VirtualBox Downloads and Download the version of VirtualBox that is suitable for your Windows 10 system

Step 2: Install VirtualBox

Run the VirtualBox installer that you downloaded in Step 1 and follow the installation wizard, accepting the default settings.

Step 3: Download Ubuntu ISO

Visit the official Ubuntu website: Ubuntu Downloads and download the LTS (Long Term Support) version of Ubuntu.

Step 4: Create a New Virtual Machine in VirtualBox

Open VirtualBox, Click on "New" to create a new virtual machine and then enter a name for your virtual machine (e.g., "Ubuntu") after this elect "Linux" as the type and "Ubuntu" as the version and then click "Next."

Step 5: Assign Memory (RAM)

Choose the amount of RAM to allocate to your virtual machine. A minimum of 2048 MB (2 GB) is recommended for Ubuntu and then click "Next."

Step 6: Create a Virtual Hard Disk

Choose "Create a virtual hard disk now" and click "Create." Then select "VDI (VirtualBox Disk Image)" as the hard disk file type and choose whether to allocate disk space dynamically (recommended) or fixed size. After this allocate at least 25 GB of space for the virtual hard disk and then click "Create."

Step 7: Mount Ubuntu ISO

With your new virtual machine selected, click "Settings." Go to the "Storage" tab.In the "Controller: IDE" section, click on the empty disk icon under "Controller: IDE." On the right side, click on the disk icon next to "Optical Drive" and choose "Choose a disk file..."Select the Ubuntu ISO file you downloaded.

Step 8: Start the Virtual Machine

With your virtual machine selected, click "Start." The machine will boot from the Ubuntu ISO.
Follow the on-screen instructions to install Ubuntu.

Step 9: Install Ubuntu

Choose your language and click "Install Ubuntu." Follow the installation wizard, selecting options such as keyboard layout, installation type, and user account information. When prompted, choose to install third-party software for graphics and Wi-Fi hardware .Click "Install" and wait for the installation to complete.

Step 10: Complete Installation

Once the installation is complete, the virtual machine will prompt you to remove the installation medium (ISO file). Click "Enter" to restart the virtual machine. Ubuntu should now boot from the virtual hard disk

Results:

- Successful study of Distributed Ledger Technology and its sub-techniques.
- Installation of Ubuntu O.S. LTS version on VirtualBox in Windows 10 for virtualization.

CONCLUSION:

The experiment achieved its objectives, providing a comprehensive understanding of Distributed Ledger Technology and successfully setting up the virtualized Ubuntu environment on Windows 10 for further exploration and development.

VIVA QUESTIONS:

1) What is Distributed Ledger Technology (DLT) and how does it work?
2) What are some sub-techniques related to Distributed Ledger Technology?
3) Why is virtualization useful for experimenting with operating systems?
4) What is the process for installing Ubuntu LTS on VirtualBox in Windows 10?
5) What are the key benefits of using Ubuntu for blockchain development?

Prepared by : Dr. N. M. Kandoi	Approved by Head : Dr. J.M. Patil

	SHRI SANT GAJANAN MAHARAJ COLLEGE OF ENGG, SHEGAON	**LABORATORY MANUAL**
	PRACTICAL EXPERIMENT INSTRUCTION SHEET	
	EXPERIMENT TITLE: Investigating and setting up Docker and Docker Compose on the Ubuntu platform.	

EXPERIMENT NO. SSGMCE/WI/CSE/01/8KS06/02	ISSUE NO. : 00	ISSUE DATE : 01-10-2024
REV. DATE :	REV. NO. : 00	DEPTT. : COMPUTER SCIENCE AND ENGINEERING
LABORATORY : EMERGING TECHNOLOGY LAB VI (DLT)	SEMESTER : VIII	PAGE :1 OF 3

AIM: Investigating and setting up Docker and Docker Compose on the Ubuntu platform.

OBJECTIVE:

- To explore the functionality and capabilities of Docker and Docker Compose.
- To gain hands-on experience in the installation process of Docker and Docker Compose on the Ubuntu platform.
- To understand the compatibility and integration of Docker tools with Ubuntu.

SCOPE:

Explore Docker and Docker Compose installation on Ubuntu, gaining insights into containerization benefits and principles and establish practical foundation for deploying and managing applications on Ubuntu using Docker, aiming to leverage the advantages of containerized environments.

FACILITIES: Computer System, Virtual Box, Ubuntu Setup.

.

THEORY/EXPERIMENT PROCEDURE:

Docker:Docker is a containerization platform that enables the development, deployment, and running of applications in containers. Containers provide a consistent and isolated environment, ensuring that applications run reliably across various computing environments and it allow a developer to package up an application with all parts it needs, such as libraries and other dependencies, and ship it all out as one package.

How Docker Works:

- Developers create a Dockerfile describing the application and its dependencies.
- Docker builds the Dockerfile into an image.
- The image is then distributed or stored in a registry.
- Docker runs the image as a container on any environment that supports Docker.

Docker Composer: Docker Compose is a tool for defining and running multi-container Docker applications. It allows you to define an entire application stack, including services, networks, and volumes, in a single **docker-compose.yml** file. With one command, you can start all the services and containers defined in the file.

How Docker Compose Works:

- Developers define their multi-container application stack in a **docker-compose.yml** file.
- The **docker-compose** command is used to start entire application stack defined in the file.
- Docker Compose creates and starts containers for each service, sets up the specified networks, and mounts volumes as defined in the configuration.
- It provides a convenient way to manage the entire application lifecycle, including starting, stopping, and scaling services.

In Hyperledger Fabric(DLT) Docker and Docker Compose play crucial roles in simplifying deployment, ensuring consistency, and facilitating scalability:

- Hyperledger Fabric components, such as peers, orderers, and certificate authorities, are containerized using Docker. This provides a standardized and isolated environment for each component, making deployment consistent across different environments
- Docker containers isolate Hyperledger Fabric nodes, preventing conflicts between components and ensuring that each node operates independently. This enhances the security and stability of the Hyperledger Fabric network.
- Docker containers package all dependencies and configurations required for Hyperledger Fabric, making the network easily portable

Installation Process:
1.Docker Installation:
a. Update the package index: sudo apt update
b. Install prerequisites: sudo apt install apt-transport-https ca-certificates curl software-properties-common
c. Add the Docker GPG key: curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg
d. Set up the stable Docker repository: echo "deb [signed-by=/usr/share/keyrings/docker-archive-keyring.gpg] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
e. Update the package index again: sudo apt update
f. Install Docker: sudo apt install docker-ce docker-ce-cli containerd.io
g. Verify the installation: sudo docker run hello-world

2. Docker Compose Installation:
a. Download the Docker Compose binary: sudo curl -L "https://github.com/docker/compose/releases/latest/download/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
b. Apply executable permissions to the binary: sudo chmod +x /usr/local/bin/docker-compose
c. Create a symbolic link to make it accessible system-wide: sudo ln -s /usr/local/bin/docker-compose /usr/bin/docker-compose
d. Verify the installation: docker-compose --version
Now, Docker and Docker Compose should be successfully installed on Ubuntu platform, allowing you to leverage the benefits of containerization for your applications.
Results:

- Successful installation of Docker and Docker Compose on Ubuntu.
- Verification tests confirmed proper functionality.

CONCLUSION:

Docker and Docker Compose provide reliable, consistent, and scalable solution for containerization on the Ubuntu platform, offering portability and efficient resource utilization.

VIVA QUESTIONS :

1) What is Docker, and what is its primary purpose?
2) How do you install Docker on an Ubuntu platform?
3) What is the role of Docker Compose in container management?
4) How do you create and run multiple containers using Docker Compose?
5) What are the advantages of using Docker for application deployment?

Prepared by : Dr. N. M. Kandoi	Approved by Head : Dr. J.M. Patil

	SHRI SANT GAJANAN MAHARAJ COLLEGE OF ENGG, SHEGAON				**LABORATORY MANUAL**
	PRACTICAL EXPERIMENT INSTRUCTION SHEET				
	EXPERIMENT TITLE: Installation of Prerequisites for Hyperledger Fabric: cURL, Node.js, NPM, Go, Git, Python, and Libtool, Followed by Hyperledger Fabric Network Configuration.				
EXPERIMENT NO. SSGMCE/WI/CSE/01/8KS06/03			ISSUE NO. : 00		ISSUE DATE : 01-10-2024
REV. DATE :	REV. NO. : 00	DEPTT. : COMPUTER SCIENCE AND ENGINEERING			
LABORATORY : EMERGING TECHNOLOGY LAB VI (DLT)				SEMESTER : VIII	PAGE :1 OF 3

AIM: Installation of Prerequisites for Hyperledger Fabric: cURL, Node.js, NPM, Go, Git, Python, and Libtool, Followed by Hyperledger Fabric Network Configuration.

OBJECTIVE:

- To install the necessary prerequisites for setting up a **Hyperledger Fabric** network and configure the network for blockchain development.
- Install prerequisite tools like cURL, Node.js, Git, Python, and Libtool etc. for Hyperledger Fabric.
- Configure a Hyperledger Fabric network on the local machine.
- Create a foundation for developing and deploying applications on the Hyperledger Fabric blockchain.

SCOPE:

Set up essential tools (cURL, Node.js, Git, Python, Libtool) for Hyperledger Fabric, configure a functional local network, and establish a foundation for developing and deploying applications on the Hyperledger Fabric blockchain.

FACILITIES: Computer system, Visual Studio Code (VS Code) and JavaScript software.

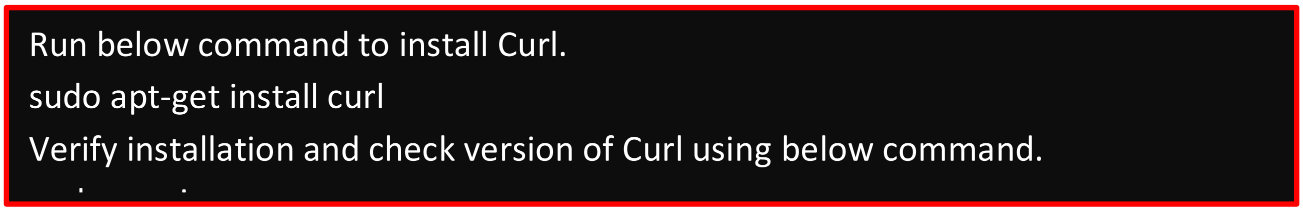
THEORY/EXPERIMENT PROCEDURE:

This experiment involves the installation of prerequisites for Hyperledger Fabric, a permissioned blockchain platform for building enterprise-level distributed ledger applications. The installation process typically includes setting up various tools and dependencies needed for development and running a Hyperledger Fabric network. Below is an explanation of each step:

1. cURL:

Purpose: cURL is a command-line tool used for making HTTP requests. It is commonly used to download files and resources from the internet.

Installation Process: Installation process for cURL involves downloading the appropriate binary or package for your operating system and then installing it.



```
Run below command to install Curl.
sudo apt-get install curl
Verify installation and check version of Curl using below command.
```

2. Node.js and NPM (Node Package Manager):

Purpose: Hyperledger Fabric utilizes Node.js as its runtime environment for executing JavaScript code. NPM is the package manager for Node.js, enabling easy management of software packages and dependencies.

Installation Process: Download the Node.js installer from the official website and run it to install both Node.js and NPM.

3. **Go (Golang):**

Purpose: Go is a programming language used for developing various components of Hyperledger Fabric. It is used for building and running peers, orderers, and other parts of the blockchain network.

Installation Process: Download the Go distribution suitable for your operating system, extract it, and set up the necessary environment variables like GOPATH and PATH.

4.**Git:**

Purpose: Git is a version control system used for tracking changes in source code during software development. It is essential for managing and obtaining the source code of Hyperledger Fabric components.

Installation Process: Download the Git installer for your operating system and follow the installation instructions.

5.Python:

Purpose: Python is a general-purpose programming language, and it might be used for various scripts and tools within the Hyperledger Fabric installation process.

Installation Process: Download the Python installer and install it on your system. Ensure that the Python executable is added to your system's PATH.

Libtool:

Purpose: Libtool is a utility for building shared libraries. It is used in the build process of software projects to handle library-related tasks.

Installation Process: The installation process may vary depending on the operating system. Typically, you can use your system's package manager to install Libtool.

Hyperledger Fabric Network Configuration:

Purpose: Once the prerequisites are installed, the next step is to configure the Hyperledger Fabric network. This involves defining the network's architecture, creating cryptographic materials, setting up peers, orderers, and channels.

Configuration Process: Follow the official documentation of Hyperledger Fabric to create configuration files, generate cryptographic materials using tools like Cryptogen and configtxgen, and set up the network based on your specific requirements.

i) **Clone the Hyperledger Fabric Samples Repository by following commands**

```
git clone https://github.com/hyperledger/fabric-samples.git
cd fabric-samples
```

ii) **Download Fabric Binary and Docker Image**s-

These images are required to set up and run the network.

```
curl -sSL https://bit.ly/2ysbOFE | bash -s
```

iii) **Start the Test Network using following commands –**

Navigate to the test network directory and launch the network.

```
cd test-network
./network.sh up createChannel -c mychannel -ca
```

This creates a Fabric network with a single channel called mychannel

Results:

1. All the prerequisites were successfully installed, ensuring that the development environment was ready for Hyperledger Fabric.
2. The Fabric binaries and Docker images were downloaded successfully, setting up a foundation for running the Fabric network.
3. The test network was successfully deployed, and a channel was created to simulate a blockchain network environment.

CONCLUSION:

This experiment involved installing all necessary tools and dependencies for working with Hyperledger Fabric. After setting up the environment, a basic Hyperledger Fabric network was successfully configured. This configuration forms the base for experimenting with different components of Hyperledger, such as chaincode development, consensus mechanisms, and transaction flows.

VIVA QUESTIONS:

1) What is the purpose of Hyperledger Fabric in blockchain networks?
2) Why is cURL required for setting up Hyperledger Fabric?
3) How do Node.js and NPM contribute to Hyperledger Fabric development?
4) What role does Go play in the Hyperledger Fabric environment?
5) How do you configure a Hyperledger Fabric network after installing the prerequisites?

Prepared by : Dr. N. M. Kandoi	Approved by Head : Dr. J.M. Patil

	SHRI SANT GAJANAN MAHARAJ COLLEGE OF ENGG, SHEGAON	**LABORATORY MANUAL**
	PRACTICAL EXPERIMENT INSTRUCTION SHEET	
	EXPERIMENT TITLE: Constructing the initial Hyperledger Fabric network and deploying the chaincode (smart contract) onto this network.	

EXPERIMENT NO. SSGMCE/WI/CSE/01/8KS06/04	ISSUE NO. : 00	ISSUE DATE : 01-10-2024
REV. DATE :	REV. NO. : 00	DEPTT. : COMPUTER SCIENCE AND ENGINEERING
LABORATORY : EMERGING TECHNOLOGY LAB VI (DLT)	SEMESTER : VIII	PAGE :1 OF 3

AIM: Constructing the initial Hyperledger Fabric network and deploying the chaincode (smart contract) onto this network.

OBJECTIVE:

- Understand how to build and set up the initial Hyperledger Fabric network, including peers, ordering service, and CA.
- Learn how to package, develop, and deploy chaincode (smart contract) on the network.
- Understand how to interact with the deployed chaincode to perform transactions and validate network functionality.

SCOPE:
This experiment involves building a basic Hyperledger Fabric network, deploying chaincode (smart contracts), and exploring its utility for automating business processes in a secure, decentralized manner.

FACILITIES: Computer System, Docker & Docker Compose, Hyperledger Fabric Binaries and Samples, Node.js & npm, Go Programming Language (optional), cURL, Fabric SDKs (Node.js, Java, or Go), Git.

THEORY/EXPERIMENT PROCEDURE:
Hyperledger Fabric Network:

Hyperledger Fabric is a modular framework that provides a foundation for developing blockchain applications with configurable network structures and consensus mechanisms. It uses a permissioned model, where participants are known to each other and have defined roles, unlike permissionless blockchains like Ethereum or Bitcoin.

Chaincode:

Chaincode is Fabric’s term for smart contracts. It is written in Go, Java, or JavaScript and defines the business logic for the interactions between the ledger and the network participants. Chaincode is installed on peers and invoked by transactions to update the ledger.

Components of a Fabric Network:

Peers: Nodes that store copies of the ledger and validate transactions.
Ordering Service: Orders transactions into blocks and propagates them to peers.
Certificate Authority (CA): Manages identities of participants.
Channels: Private communication channels between a subset of peers.
Chaincode (Smart Contract): Business logic for the network.

Implementation Process:

1. Setting up the Development Environment: To successfully complete this task, make sure your system has the following prerequisites installed: Docker, Docker Compose, Go, Node.js, and Hyperledger Fabric binaries. Having all these tools are essential.

```
# Download Hyperledger Fabric binaries and docker images
curl -sSL https://bit.ly/2ysbOFE | bash -s
```

2 Configuring the Network:

- Fabric networks are defined by configuration files that define the structure of the network. These include details about the peers, the orderer, channels, and the CA.
- Create a directory for the network configuration and generate the necessary artifacts (certificates, keys).

```
# Navigate to the fabric-samples directory
cd fabric-samples/test-network
# Start the Fabric network
./network.sh up
# Create a channel
./network.sh createChannel -c mychannel
```

3. Writing and Packaging the Chaincode:

- Write a chaincode (smart contract) in your preferred language (Go, Node.js, or Java). For this example, we'll use Node.js

```
'use strict';
const { Contract } = require('fabric-contract-api');
class MyAssetContract extends Contract {
async createAsset(ctx, id, value) {
const asset = { id, value };
await ctx.stub.putState(id, Buffer.from(JSON.stringify(asset)));
return asset;
}
async readAsset(ctx, id) {
const assetJSON = await ctx.stub.getState(id);
if (!assetJSON || assetJSON.length === 0) {
throw new Error(`Asset ${id} does not exist`);
}
return JSON.parse(assetJSON.toString());
}
}
```

Package the chaincode into a .tar.gz file.

```
# Navigate to the chaincode directory
cd fabric-samples/chaincode
# Package the chaincode
```

```
tar -czf mychaincode.tar.gz mychaincode/
```

4. Deploying the Chaincode:

- Install and approve the chaincode on the peers, then commit it to the channel.

```
# Install the chaincode on peer0.org1
./network.sh deployCC -ccn mychaincode -ccp ./chaincode/mychaincode -ccl javascript
```

5. Interacting with the Chaincode:

- Use the CLI or an SDK to interact with the deployed chaincode. You can invoke the createAsset or readAsset functions, for example.

```
 # Invoke the chaincode to create an asset
 docker exec cli peer chaincode invoke -o localhost:7050 \
--ordererTLSHostnameOverride orderer.example.com \
--tls --cafile $ORDERER_CA \
-C mychannel -n mychaincode \
-c '{"function":"createAsset","Args":["asset1", "100"]}'
```

Query the asset:

```
docker exec cli peer chaincode query -C mychannel -n mychaincode -c
'{"Args":["readAsset", "asset1"]}'
```

Results:

Once the Hyperledger Fabric network is successfully set up, and the chaincode is deployed, the following results can be observed:

1. Network Initialization: The Fabric network with peers, an orderer, and a channel should be operational.
2. Chaincode Deployment: The chaincode should be successfully deployed on the peer nodes.
3. Asset Creation and Querying: The creation of an asset through the chaincode should be successful, and querying the asset should return its details from the ledger.

CONSLUSION:

In this experiment, we successfully constructed a Hyperledger Fabric network and deployed a chaincode to it. We demonstrated the ability to create and query assets on the blockchain. The network setup highlights the modularity and permissioned nature of Hyperledger Fabric, making it a robust platform for developing enterprise blockchain solutions.

VIVA QUESTIONS:

1) What are the key components of a Hyperledger Fabric network?
2) What is chaincode in Hyperledger Fabric, and what is its purpose?
3) How do you create and configure a Hyperledger Fabric network?
4) What steps are involved in deploying chaincode onto a Fabric network?
5) How is a transaction executed once the chaincode is deployed?

Prepared by : Dr. N. M. Kandoi	Approved by Head : Dr. J.M. Patil

	SHRI SANT GAJANAN MAHARAJ COLLEGE OF ENGG, SHEGAON	**LABORATORY MANUAL**
	PRACTICAL EXPERIMENT INSTRUCTION SHEET	
	EXPERIMENT TITLE: Exploration and installation of the MultiChain Blockchain platform, creating a stream, and demonstrating its utility for general data storage and retrieval.	

EXPERIMENT NO. SSGMCE/WI/CSE/01/8KS06/05	ISSUE NO. : 00	ISSUE DATE : 01-10-2024
REV. DATE :	REV. NO. : 00	DEPTT. : COMPUTER SCIENCE AND ENGINEERING
LABORATORY : EMERGING TECHNOLOGY LAB VI (DLT)	SEMESTER : VIII	PAGE :1 OF 3

AIM: Exploration and installation of the MultiChain Blockchain platform, creating a stream, and demonstrating its utility for general data storage and retrieval.

OBJECTIVE:

- Explore and install the MultiChain blockchain platform.
- Create a stream within the MultiChain platform.
- Demonstrate the use of a stream for general data storage and retrieval.
- Highlight the practical application of blockchain technology for storing and retrieving data beyond cryptocurrencies.

SCOPE:
The scope covers installing MultiChain, creating a stream, and demonstrating how it can be used for secure, decentralized data storage and retrieval across nodes, highlighting MultiChain's utility beyond cryptocurrency transactions.

FACILITIES: Computer system, MultiChain 2.0 or later version, compatible with Linux (Ubuntu), macOS, or Windows. Optional tools include Python for interacting with MultiChain scripts, Postman for API testing, Blockchain Explorer to monitor streams and transactions, and Docker for containerized deployment.

THEORY/EXPERIMENT PROCEDURE:

MultiChain Overview

MultiChain is a permissioned blockchain platform designed for private blockchain networks. It extends the Bitcoin protocol with additional features such as native assets, data streams, permissions management, and enhanced scalability. MultiChain's streams feature is particularly useful for structured and unstructured data storage, allowing secure, tamper-proof storage of data on the blockchain.

Streams in MultiChain

Streams in MultiChain are used to store key-value pairs or unstructured data in a decentralized manner. Users can publish, subscribe to, and retrieve data from these streams, making them ideal for applications like secure document storage, audit trails, and data sharing.

Key concepts:

Streams: Channels for storing and retrieving data**.**

Publish: Adding data to the stream**.**

Subscribe: Accessing and retrieving data from the stream**.**

Implementation

Step 1: Installation of MultiChain

1. **Download MultiChain**: Download the latest version from the official website:
   wget https://www.multichain.com/download/multichain-2.0.tar.gz
2. **Extract this tar file and Install multichain2.0**
   ```
   tar -xvzf multichain-2.0.tar.gz
   cd multichain-2.0
   sudo mv multichaind multichain-cli multichain-util /usr/local/bin
   ```

Step 2: Creating a Blockchain

1. **Initialize a New Blockchain:** Create a new blockchain named mychain**.**
   ```
   multichain-util create mychain
   ```
2. **Launch the Blockchain**: Start the blockchain in daemon mode:
   ```
   multichaind mychain -daemon
   ```

Step 3: Creating a Stream

1. **Create a Stream:** Once the blockchain is running, create a stream called data_stream.
   ```
   multichain-cli mychain create stream data_stream true
   multichain-cli mychain subscribe data_stream
   ```
2. **Publish Data:** Publish key-value pairs to the stream**:**
   ```
   multichain-cli mychain publish data_stream "key1" "data1"
   multichain-cli mychain publish data_stream "key2" "data2"
   ```

Step 4: Retrieving Data from the Stream

1. **Retrieve Data:Query the stream to retrieve stored data:**
   ```
   multichain-cli mychain liststreamitems data_stream
   ```

Step 5: Optional - Using Python or Postman

Interact with the blockchain via Python using the requests library.
Use Postman to test API interactions for publishing and retrieving data.

Result

After completing all the steps above, MultiChain will be successfully installed, a stream will be created, and its functionality will be demonstrated by storing and retrieving data. When querying the stream, the output might look like this:

```
[
 {
  "key": "key1",
  "data": "data1"
 },
 {
  "key": "key2",
  "data": "data2"
 } ]
```

CONCLUSION:

The experiment successfully demonstrates the MultiChain blockchain platform's ability to handle general-purpose data storage and retrieval using streams. MultiChain’s data streams allow for decentralized, tamper-proof, and efficient storage of data, making it applicable in various sectors beyond cryptocurrency, such as document storage, logging systems, and secure data sharing.

VIVA QUESTIONS:

1) What is the purpose of creating a stream in MultiChain?
2) How do you install the MultiChain platform on your system?
3) What types of data can be stored in a MultiChain stream?
4) How is data retrieved from a stream in MultiChain?
5) What are the advantages of using MultiChain for data storage and retrieval?

Prepared by : Dr. N. M. Kandoi	Approved by Head : Dr. J.M. Patil

	SHRI SANT GAJANAN MAHARAJ COLLEGE OF ENGG, SHEGAON				**LABORATORY MANUAL**	
	PRACTICAL EXPERIMENT INSTRUCTION SHEET					
	EXPERIMENT TITLE: Demonstrating the secure sharing of PDF documents among MultiChain blockchain nodes with exclusive permissions.					
EXPERIMENT NO. SSGMCE/WI/CSE/01/8KS06/06			ISSUE NO. : 00		ISSUE DATE : 01-10-2024	
REV. DATE :		REV. NO. : 00	DEPTT. : COMPUTER SCIENCE AND ENGINEERING			
LABORATORY : EMERGING TECHNOLOGY LAB VI (DLT)				SEMESTER : VIII		PAGE :1 OF 3

AIM: Demonstrating the secure sharing of PDF documents among MultiChain blockchain nodes with exclusive permissions.

OBJECTIVE:

- Demonstrate secure sharing of PDF documents among MultiChain blockchain nodes.
- Set up exclusive permissions for nodes to ensure that only authorized participants can access the documents.
- Showcase how MultiChain can be used for controlled, permissioned data sharing

SCOPE:

This experiment aims to showcase how MultiChain can be utilized for securely sharing documents, such as PDFs, among nodes with permissioned access, demonstrating blockchain's role in secure data sharing.

FACILITIES: Computer system, MultiChain 2.0 or later version, compatible with Linux (Ubuntu), macOS, or Windows Optional tools include Python for interacting with MultiChain scripts, Postman for API testing, Blockchain Explorer to monitor streams and transactions, and Docker for containerized deployment.

THEORY/EXPERIMENT PROCEDURE:

MultiChain Overview

MultiChain is a private blockchain platform that supports permissioned data sharing and storage. Unlike public blockchains, MultiChain allows administrators to control who can participate in the network and what actions they can perform. This permissioning makes MultiChain an ideal platform for securely sharing sensitive data, such as PDF documents, among trusted parties.

Permissions in MultiChain:

MultiChain provides granular permission controls, including the ability to:

- **Connect**: Allow or deny nodes from connecting to the blockchain.
- **Send/Receive**: Control which nodes can send and receive transactions.
- **Create/Issue**: Determine which nodes can create streams and issue assets.
- **Admin**: Grant administrative control for managing permissions on the network.

By leveraging these permissions, we can ensure that only authorized nodes can upload, share, or retrieve documents on the blockchain.

Implementation

Step 1: Installation of MultiChain

Download and Install MultiChain: Download the latest version of MultiChain and install it on all participating nodes (e.g., Node A, Node B, Node C).

```
wget https://www.multichain.com/download/multichain-2.0.tar.gz
tar -xvzf multichain-2.0.tar.gz
cd multichain-2.0
sudo mv multichaind multichain-cli multichain-util /usr/local/bin
```

Step 2: Creating a Permissioned Blockchain

Create the Blockchain: On Node A, create the permissioned blockchain.

```
multichain-util create pdf_chain
multichaind pdf_chain -daemon
```

Grant Permissions to Nodes:

Assign connect, send, receive, and other required permissions to Node B and Node C.

perl

```
multichain-cli pdf_chain grant <nodeB_address> connect,send,receive
multichain-cli pdf_chain grant <nodeC_address> connect,send,receive
```

Step 3: Creating a Stream for PDF Document Sharing

1. **Create a Document Stream:** On Node A, create a stream to store and share PDF documents.

   ```
   multichain-cli pdf_chain create stream pdf_docs true
   multichain-cli pdf_chain subscribe pdf_docs
   ```

2. **Grant Exclusive Permissions:**

Grant exclusive permissions to Node B and Node C to publish and retrieve documents in the pdf_docs stream**.**

```
multichain-cli pdf_chain grant <nodeB_address> write,publish
multichain-cli pdf_chain grant <nodeC_address> read,subscribe
```

Step 4: Publishing and Retrieving PDF Documents

1. **Publish a PDF Document:** On Node B, publish a PDF document to the blockchain**.**

   ```
   multichain-cli pdf_chain publish pdf_docs "pdf_key1" "hex_data_of_pdf"
   ```

   Convert the PDF file to hexadecimal format before publishing.

   ```
   xxd -p <filename.pdf> | tr -d '\n'
   ```

2. **Retrieve the PDF Document:**

   On Node C, retrieve the published PDF document.

   ```
   multichain-cli pdf_chain liststreamitems pdf_docs
   ```

   Convert the hexadecimal data back to a PDF file.

   ```
   echo "<hex_data>" | xxd -r -p > retrieved_document.pdf
   ```

Step 5: Optional - Using Python for Automation

Use Python scripts to automate the conversion, publishing, and retrieval of PDF documents using MultiChain's JSON-RPC API.

Result

The experiment successfully demonstrates the secure sharing of PDF documents among MultiChain nodes with exclusive permissions. Node B is able to publish PDF documents to the blockchain, while Node C retrieves them. The permission system ensures that unauthorized nodes cannot access or publish documents, preserving the privacy and integrity of the data.

Example output when retrieving stream data:

```
 [
  {
    "key": "pdf_key1",
    "data": "hex_data_of_pdf"
  }
]
```

After converting the hexadecimal data back to a PDF file, Node C will have access to the original PDF document.

CONCLUSION:

This experiment showcases how MultiChain's permissioned blockchain network can be used to securely share PDF documents among authorized participants. By utilizing the platform's stream and permissioning features, we can ensure that only specific nodes have the right to publish or retrieve sensitive documents. This method offers a decentralized, tamper-proof, and secure solution for document sharing across distributed systems.

VIVA QUESTIONS:

1) How does MultiChain ensure secure sharing of documents between nodes?
2) What are streams in MultiChain, and how are they used for document sharing?
3) How do you set permissions for nodes in MultiChain?
4) What is the benefit of using a blockchain for secure document sharing?
5) How are PDF documents stored and retrieved in MultiChain?

Prepared by : Dr. N. M. Kandoi	Approved by Head : Dr. J.M. Patil

	SHRI SANT GAJANAN MAHARAJ COLLEGE OF ENGG, SHEGAON	**LABORATORY MANUAL**
	PRACTICAL EXPERIMENT INSTRUCTION SHEET	
	EXPERIMENT TITLE: Exploring asset transfer using MultiChain Blockchain and studying the installation process of Hydra Blockchain.	

EXPERIMENT NO. SSGMCE/WI/CSE/01/8KS06/07	ISSUE NO. : 00	ISSUE DATE : 01-10-2024
REV. DATE :	REV. NO. : 00	DEPTT. : COMPUTER SCIENCE AND ENGINEERING
LABORATORY : EMERGING TECHNOLOGY LAB VI (DLT)	SEMESTER : VIII	PAGE :1 OF 3

AIM: Exploring asset transfer using MultiChain Blockchain and studying the installation process of Hydra Blockchain.

OBJECTIVE:

- Explore the process of asset creation and transfer using the MultiChain blockchain platform.
- Demonstrate how assets can be transferred between nodes on a permissioned blockchain.
- Study the installation process of the Hydra Blockchain platform and understand its core features.

SCOPE:

The scope includes exploring the asset creation and transfer functionality in MultiChain and studying the installation process of the Hydra Blockchain, providing insights into managing digital assets across nodes.

FACILITIES: Computer system, MultiChain 2.0 or later for asset transfer and Hydra node binaries for installation and exploration, compatible with Linux (Ubuntu), macOS, or Windows, Optional tools include Python for scripting and automating asset transfers, Postman for testing the MultiChain API, and Docker for containerized deployment of both MultiChain and Hydra nodes.

THEORY/EXPERIMENT PROCEDURE:

MultiChain is a permissioned blockchain platform that allows the creation, issuance, and transfer of assets in a secure, decentralized manner. Assets in MultiChain can represent anything of value, including tokens, ownership rights, or real-world items. The permissioned nature of MultiChain ensures that only authorized nodes can create, transfer, or receive assets.

Asset Creation and Transfer

In MultiChain, assets can be created on the blockchain, transferred between nodes, and tracked transparently. Each asset is represented by a unique name and can be sent between participants. The transactions are recorded on the blockchain, ensuring tamper-proof records.

Hydra Blockchain Overview

Hydra is an innovative decentralized blockchain network with a hybrid consensus model, combining Proof of Stake (PoS) with a permissioned model designed to address scalability, security, and efficiency issues commonly faced by existing blockchain networks. It is built with a focus on providing low-cost transactions, high throughput, and a sustainable economic model that supports decentralized applications (dApps) and services.

It is a versatile and scalable blockchain platform designed for efficiency, security, and decentralized governance. Its low transaction fees, PoS consensus, and cross-chain compatibility make it a strong candidate for applications ranging from DeFi to gaming and beyond, while its eco-friendly design positions it as a forward-thinking solution in the blockchain space.

Use Cases for Hydra Blockchain

- **Decentralized Finance (DeFi)**: Hydra's low fees and scalability make it ideal for DeFi platforms, where fast and cost-efficient transactions are crucial.
- **dApps Development**: The EVM compatibility allows developers to build decentralized applications on Hydra.
- **Gaming & NFTs**: Hydra's high throughput and low transaction costs are well-suited for gaming and non-fungible token (NFT) platforms, where numerous transactions occur simultaneously.
- **Cross-Chain Solutions**: Hydra's interoperability enables seamless communication between various blockchain ecosystems, supporting cross-chain dApps and services.

Implementation

Part 1: Asset Transfer Using MultiChain

Step 1: Installing MultiChain:

Install MultiChain by following the steps outlined in experiments 5 and 6

Step 2: Creating and Launching a Blockchain

Create the Blockchain: multichain-util create asset_chain

Launch the Blockchain: multichaind asset_chain -daemon

Step 3: Creating an Asset

Grant Permission to Create Assets: Grant permissions to the node to issue assets:

```
multichain-cli asset_chain grant <address> issue
```

Issue an Asset: Create an asset named "TokenABC" and issue 1000 units to a node:

```
multichain-cli asset_chain issue <address> TokenABC 1000
```

Step 4: Transferring the Asset

Transfer Asset: Transfer some assets to another node:

```
multichain-cli asset_chain sendasset <recipient_address> TokenABC 100
```

Check Balances: Check the balance of the recipient node to confirm the asset transfer:

```
multichain-cli asset_chain getaddressbalances <recipient_address>
```

Part 2: Installing Hydra Blockchain

Step 1: Downloading and Installing Hydra

Download Hydra Node: Download Hydra binaries from the official Hydra website.

```
wget https://github.com/Hydra-Chain/Hydra/releases/download/<version>/hydra-linux.tar.gz
```

Extract and Install: Extract the downloaded file and move binaries to a suitable directory:

```
tar -xvzf hydra-linux.tar.gz
sudo mv hydrad hydra-cli /usr/local/bin
```

Step 2: Starting Hydra Node

Initialize a Hydra Node: Create a new data directory and initialize the Hydra node:

```
hydrad --datadir=/path/to/data/ init
```

Start the Hydra Node: Run the Hydra node to join the network:

```
hydrad --datadir=/path/to/data/ start
```

Step 3: Exploring Hydra Blockchain

Connect to the Hydra Network:

Use hydra-cli to check the status of the node and explore network parameters:

```
hydra-cli getinfo
```

Create and Explore Accounts: Create new accounts, and use the Hydra node to explore account creation, staking, and basic transaction functionalities.

Result

MultiChain Asset Transfer: After the successful creation and transfer of the asset "TokenABC," you will see that the asset has been issued and transferred between the nodes on the permissioned blockchain. The asset balance of the recipient node will reflect the correct transferred amount (100 units of TokenABC).

Sample result of checking balances:

```
[
  {
    "name": "TokenABC",
    "qty": 100,
    "assets": true
  }
]
```

Hydra Installation: After installing and running a Hydra node, you will have successfully connected to the Hydra network. You can now explore network parameters, participate in staking, and understand its hybrid consensus mechanism.

CONCLUSION:

This experiment successfully demonstrated the process of asset creation and transfer on a MultiChain blockchain. Assets can be securely issued and transferred between authorized nodes, showing MultiChain's capabilities for handling digital value in a permissioned blockchain environment. Additionally, the experiment covered the installation of Hydra Blockchain, providing insights into setting up and exploring a decentralized network using Hydra's hybrid consensus model. Both platforms showcase the flexibility and potential of blockchain technology for various use cases, from asset transfers to decentralized applications.

VIVA QUESTIONS:

1) What is the purpose of creating assets in MultiChain?
2) How do you transfer an asset between nodes in MultiChain?
3) What is the function of the genesis block in Hydra Blockchain?
4) How do you install MultiChain on a Linux system?
5) What is the role of Hydra Blockchain in a decentralized network?**.**

Prepared by : Dr. N. M. Kandoi	Approved by Head : Dr. J.M. Patil

	SHRI SANT GAJANAN MAHARAJ COLLEGE OF ENGG, SHEGAON	**LABORATORY MANUAL**
	PRACTICAL EXPERIMENT INSTRUCTION SHEET	
	EXPERIMENT TITLE: Creating a Private Blockchain with the Geth Client and Initiating Ether Mining on the Network.	

EXPERIMENT NO. SSGMCE/WI/CSE/01/8KS06/08	ISSUE NO. : 00	ISSUE DATE : 01-10-2024
REV. DATE :	REV. NO. : 00	DEPTT. : COMPUTER SCIENCE AND ENGINEERING
LABORATORY : EMERGING TECHNOLOGY LAB VI (DLT)	SEMESTER : VIII	PAGE :1 OF 4

AIM: Creating a Private Blockchain with the Geth Client and Initiating Ether Mining on the Network.

OBJECTIVE:

- To set up a private Ethereum blockchain using the Geth (Go Ethereum) client.
- To Initialize and configure the genesis block for the private blockchain.
- To understand the process of Launch and manage Ether mining on the private network.
- Demonstrate the process of generating and distributing Ether through mining.

SCOPE:

This experiment focuses on setting up a private Ethereum blockchain using the Geth client, initiating Ether mining, and understanding the process of transaction validation and rewards generation in a controlled environment**.**

FACILITIES: Computer System, Geth (Go Ethereum) client for running Ethereum nodes and networks, compatible with Linux (Ubuntu), or Windows, Optional tools include Ethereum Wallet for managing Ether balances, Python for automating tasks like transactions or mining scripts, Postman for testing the Ethereum RPC API, and Docker for containerized deployment of Geth nodes.

THEORY/EXPERIMENT PROCEDURE:

Ethereum Blockchain Overview
Ethereum is a decentralized, open-source blockchain platform that enables the creation and execution of smart contracts. Ether (ETH) is the native cryptocurrency of the Ethereum blockchain, and it can be generated by mining, which involves solving complex mathematical puzzles to validate and confirm transactions.

Private Ethereum Blockchain
Private Ethereum blockchain is an isolated network where nodes only interact with each other, making it suitable for testing and experimentation. Unlike public Ethereum networks, private blockchains can be customized with their own consensus mechanisms; difficulty levels, and blocks rewards.

Geth (Go Ethereum)
Geth is one of the most popular Ethereum clients, implemented in Go. It allows users to set up private Ethereum networks, mine Ether, and deploy smart contracts. Geth offers various tools for configuring nodes, managing accounts, and mining on both public and private networks.

Ether Mining
Mining is the process of validating transactions and securing the blockchain by solving cryptographic puzzles. Miners are rewarded with Ether when they successfully mine a new block.

In a private blockchain, mining can be controlled to simulate real-world Ethereum mining.

Implementation

Step 1: Installing Geth

Download and Install Geth: Download the Geth binaries for your operating system from the official Ethereum website or install via package managers:

For Linux (Ubuntu):

```
sudo apt-get install software-properties-common
sudo add-apt-repository -y ppa:ethereum/ethereum
sudo apt-get update
sudo apt-get install ethereum
```

Verify Installation: Confirm the installation by running**:** geth version

Step 2: Setting Up a Private Ethereum Blockchain

Create a Genesis File: The genesis file defines the parameters for the private blockchain, such as the difficulty, gas limit, and initial Ether balance. Create a genesis.json file with the following structure:

```
{
  "config": {
    "chainId": 1234,
    "homesteadBlock": 0,
    "eip155Block": 0,
    "eip158Block": 0
  },
  "difficulty": "400",
  "gasLimit": "8000000",
  "alloc": {
    "0xYourAccountAddress": { "balance": "1000000000000000000000" }
  }
}
```

Initialize the Blockchain: Use the Geth client to initialize the blockchain with the custom genesis file:

```
geth --datadir ./privatechain init ./genesis.json
```

Create Accounts: Create new accounts for mining and transacting**:**

```
geth --datadir ./privatechain account new
```

Step 3: Starting the Private Blockchain

Launch the Private Blockchain: Start the private blockchain using the following command**:**

```
geth --datadir ./privatechain --networkid 1234 --http --http.port 8545 --http.addr 127.0.0.1 --http.api personal,eth,net,web3,miner --nodiscover --allow-insecure-unlock
```

The --networkid is used to differentiate the private blockchain from public networks.

The --nodiscover flag ensures that the private chain does not try to connect to other nodes.

Access the Geth Console: Open the Geth console to interact with the blockchain:

geth attach http://127.0.0.1:8545

Step 4: Initiating Ether Mining

Start Mining: In the Geth console, start mining by entering:

miner.start()

Mining will begin, and you will see the process of validating blocks and generating Ether.

Check Account Balances: Monitor the mining rewards by checking the account balance:

eth.getBalance(eth.coinbase)

Stop Mining: Once satisfied with the mining process, you can stop it with:

miner.stop()

Step 5: Transferring Ether between Accounts

Send Ether: Transfer some Ether from the coinbase account (the miner’s account) to another account:

eth.sendTransaction({from: eth.coinbase, to: "0xRecipientAddress", value: web3.toWei(1, "ether")})

Check Transaction Status: Check the status of the transaction using:

eth.getTransaction("transaction_hash")

Result

The experiment results in the successful setup of a private Ethereum blockchain using Geth, where mining Ether is demonstrated. After mining a few blocks, Ether is generated and transferred between accounts. The mining process validates the blocks, and the rewards are reflected in the account balances.

For example, the initial balance of the miner account will increase with each mined block:

```
> eth.getBalance(eth.coinbase)
"50000000000000000000" // after mining some blocks
```

Ether can also be transferred between nodes, and each transaction is recorded on the private blockchain.

CONCLUSION:

This experiment successfully demonstrated the process of creating a private Ethereum blockchain using Geth and initiating Ether mining on the network. Through the genesis block setup and Geth configuration, we established a functional private network where Ether mining was simulated, and rewards were distributed. This setup allows for experimentation and learning without using the public Ethereum network, making it a valuable tool for blockchain developers and researchers.

VIVA QUESTIONS:

1) What is the purpose of the genesis block in setting up a private blockchain, and what parameters can be customized in the genesis file?
2) Explain the process of Ether mining on a private Ethereum blockchain. How does it differ from mining on the public Ethereum network?
3) What role does the networkid play in differentiating a private blockchain from the public Ethereum network?
4) How can you check the balance of an account and monitor transactions in the Geth console during the mining process?
5) What are the key advantages of using a private Ethereum blockchain for development and testing compared to the public Ethereum network?

Prepared by : Dr. N. M. Kandoi	Approved by Head : Dr. J.M. Patil

	SHRI SANT GAJANAN MAHARAJ COLLEGE OF ENGG, SHEGAON	**LABORATORY MANUAL**
	PRACTICAL EXPERIMENT INSTRUCTION SHEET	
	EXPERIMENT TITLE: 1.Implementation and Analysis of SHA-256 Hash Functions: Characteristics and Output Patterns 2. Determining Plaintext from a SHA-256 Hash Using Brute Force and Dictionary Methods	

EXPERIMENT NO. SSGMCE/WI/CSE/01/8KS06/09	ISSUE NO. : 00	ISSUE DATE : 01-01-2025
REV. DATE :	REV. NO. : 01	DEPTT. : COMPUTER SCIENCE AND ENGINEERING
LABORATORY : EMERGING TECHNOLOGY LAB VI (DLT)	SEMESTER : VIII	PAGE :1 OF 6

AIM: 1.Implementation and Analysis of SHA-256 Hash Functions: Characteristics and Output Patterns

OBJECTIVE:

- **Implement and observe** the SHA-256 cryptographic hash function in Node.js, analyzing its fixed-length output and sensitivity to input variations.
- **Test the function** with diverse input types (e.g., strings, empty input, large input) to explore characteristics like the avalanche effect and hash length consistency.
- **Understand the significance** of SHA-256 in cryptography, particularly its role in ensuring data integrity and security.

SCOPE:

- **Understanding SHA-256**: Analyze how SHA-256 generates a fixed-length hash from inputs of varying size and content.
- **Hash Characteristics**: Explore case sensitivity, hash length consistency, and changes in output with small input variations.
- **Practical Application**: Gain insights into the use of SHA-256 in cryptographic and security applications such as digital signatures, data integrity, and blockchain

FACILITIES: Computer System, VS Code, Java Script.

THEORY/EXPERIMENT PROCEDURE:

SHA-256 (Secure Hash Algorithm 256-bit) is a cryptographic hash function designed to provide data integrity by converting arbitrary input into a fixed 256-bit (32-byte) hash value. Characteristics of SHA-256 include:

1. **Deterministic Output**: The same input always generates the same hash.
2. **Fixed-Length Output**: Regardless of input size, the output is always 256 bits.
3. **Avalanche Effect**: A small change in input results in a significantly different hash.
4. **Pre-Image Resistance**: It is computationally infeasible to reverse-engineer the original input from the hash.
5. **Collision Resistance**: It is extremely unlikely for two different inputs to produce the same hash.

Let's implement and observe its characteristics and output patterns using **Node.js**.

Step 1: Project Set-up

1. Create a folder named sha256-demo: mkdir sha256-demo, cd sha256-demo
2. Initialize the project with Node.js: npm init -y
3. Install the required crypto package (already built into Node.js, so no need to install separately).

Step 2: Write and run the SHA-256 Implementation

1. Create a index.js file and run it using node index.js

```
//index.js
const crypto = require('crypto');
// Function to generate SHA-256 hash
function generateSHA256Hash(input) {
   return crypto.createHash('sha256').update(input).digest('hex');
}
// Test cases to observe SHA-256 characteristics
const inputs = [
   "Hello, World!",              // Basic string
   "hello, world!",              // Case-sensitive input
   "",                       // Empty input
   "A".repeat(20),             // Large input
   "The quick brown fox jumps over the lazy dog", // Famous pangram
   "The quick brown fox jumps over the lazy dog." // Slight variation
];
// Display results
inputs.forEach((input, index) => {
   const hash = generateSHA256Hash(input);
   console.log(`Input ${index + 1}: "${input}"`);
   console.log(`SHA-256 Hash: ${hash}`);
   console.log(`Hash Length: ${hash.length} characters`);
   console.log("-".repeat(50));
       });
```

RESULTt: Upon running the implementation, the following observations were made:

1. The hash output for each input was a fixed 64-character hexadecimal string, demonstrating the fixed-length nature of SHA-256.
2. Case sensitivity was evident as inputs like "Hello, World!" and "hello, world!" produced different hashes.
3. An empty string still generated a unique hash, emphasizing that every input, including null inputs, has a unique representation.
4. The avalanche effect was observed as minor changes in input (e.g., adding a period to the pangram) resulted in completely different hashes.
5. Consistent results across all test cases verified the deterministic behavior of the function.

EXPECTED OOTPUT: For the test cases provided, the output will show

The hash of each input and Observations about input sensitivity and hash properties.

```
PS E:\Session-24-25\8th-DLT\Lab\SHA-256\sha256-demo> node index.js
Input 1: "Hello, World!"
SHA-256 Hash: dffd6021bb2bd5b0af676290809ec3a53191dd81c7f70a4b28688a362182986f
Hash Length: 64 characters
--------------------------------------------------
Input 2: "hello, world!"
SHA-256 Hash: 68e656b251e67e8358bef8483ab0d51c6619f3e7a1a9f0e75838d41ff368f728
Hash Length: 64 characters
--------------------------------------------------
Input 3: ""
SHA-256 Hash: e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855
Hash Length: 64 characters
--------------------------------------------------
Input 4: "AAAAAAAAAAAAAAAAAAAA"
SHA-256 Hash: edfcaac579024f574adbcaa3c13e4fd2b7f1797826afe679f2144af2cb5c062d
Hash Length: 64 characters
--------------------------------------------------
Input 5: "The quick brown fox jumps over the lazy dog"
SHA-256 Hash: d7a8fbb307d7809469ca9abcb0082e4f8d5651e46d3cdb762d02d0bf37c9e592
Hash Length: 64 characters
--------------------------------------------------
Input 6: "The quick brown fox jumps over the lazy dog."
SHA-256 Hash: ef537f25c895bfa782526529a9b63d97aa631564d5d789c2b765448c8635fb6c
        Hash Length: 64 characters
```

OBSERVATIONS:

- **Deterministic Output**: The same input always produces the same hash.
- **Case Sensitivity**: Changing case of characters (e.g., Hello, World! vs. hello, world!) changes the hash completely.
- **Empty Input**: Even an empty input produces a fixed-length hash.
- **Input Size**: Hash length remains 256 bits (64 characters in hexadecimal) regardless of the input size.
- **Small Change, Big Difference**:
  Small variations in input (e.g., adding a period) result in drastically different hashes.
- **Uniform Distribution**: Hashes appear random and do not exhibit any predictable patterns

Conclusion:

Implementation of SHA-256 in Node.js successfully demonstrated essential characteristics of hash function:

1. A fixed-length output regardless of input size.
2. Sensitivity to input variations, including case and minor textual changes.
3. Robustness in handling diverse input scenarios, confirming the deterministic and secure properties of SHA-256.

The experiment reinforces SHA-256's significance in ensuring data integrity and security in cryptographic applications.

AIM: **2**. Determining Plaintext from a SHA-256 Hash. Using Brute Force and Dictionary Methods

OBJECTIVE:

- **Determine** the original plaintext from a given SHA-256 hash using brute force and dictionary-based methods.
- **Evaluate** the effectiveness and efficiency of brute force and dictionary methods in recovering plaintext.
- **Understand** the limitations of these methods in real-world applications and the strength of SHA-256 as a cryptographic function.

SCOPE:

- **Brute Force Method**: Explore the process of systematically trying every possible combination of characters until the correct plaintext is found.
- **Dictionary Method**: Use a precompiled list of commonly used words or phrases to match against the hash.
- **Comparison**: Compare the efficiency and feasibility of brute force versus dictionary methods in recovering plaintext from SHA-256 hashes.
- **Cryptographic Understanding**: Understand the challenges involved in hash reversal and the limitations of these methods in real-world applications.

FACILITIES: Computer System, VS Code, Python.

THEORY/EXPERIMENT PROCEDURE:

SHA-256 is a cryptographic hash function that produces a fixed 256-bit output for any input of arbitrary length. It is designed to be a one-way function, meaning that it is computationally infeasible to reverse the process and obtain the original plaintext. However, methods like brute force and dictionary attacks attempt to reverse the hash by:

1. **Brute Force**: Trying all possible combinations of characters until the hash matches the target.
2. **Dictionary Attack**: Using a predefined list of common words or phrases to check against the hash, assuming the plaintext is a common word or phrase.

Both methods rely on the assumption that the plaintext might be easily guessable, which is often not the case with strong cryptographic hashe

Let's implement both Brute Force and Dictionary Methods

```
//brute.js
const crypto = require('crypto');
const target = "nmk";
console.log(sha256(target));
// SHA-256 hash function
function sha256(input) {
   return crypto.createHash('sha256').update(input).digest('hex');
}
// Target hash for "hello"
const targetHash = '87a8ec19bb5c61cf468147ee90636ee9cd02615c7429ec99701484607c02de23';  //
SHA-256 of "hello"
// Character set and max length
const charset = 'abcdefghijklmnopqrstuvwxyz';
const maxLength = 5;
```

```
// Brute force attack function
function bruteForceAttack() {
   let found = false;
   // Try all combinations of characters up to maxLength
   for (let length = 1; length <= maxLength; length++) {
      found = tryCombinations(charset, length, '', found);
      if (found) break;
   }  }
// Recursive function to generate and check combinations
function tryCombinations(charset, length, prefix, found) {
   if (length === 0) {
      const hash = sha256(prefix);
      if (hash === targetHash) {
         console.log(`Match found! Input: ${prefix}`);
         return true;  // Found the match, return true
      }
   } else {
      for (let i = 0; i < charset.length; i++) {
         found = tryCombinations(charset, length - 1, prefix + charset[i], found);
         if (found) break;  // If found, exit the loop
      }
   }
   return found;  // Return the status (found or not)
}
// Start brute force attack
bruteForceAttack();

// dictionary.js

const crypto = require('crypto');

// Target hash
const targetHash = '87a8ec19bb5c61cf468147ee90636ee9cd02615c7429ec99701484607c02de23';
// Wordlist (can be expanded with more words)
const wordlist = ['hello', 'world', 'test', 'password', '12345','nmk'];
// Function to generate SHA-256 hash
function sha256(input) {
   return crypto.createHash('sha256').update(input).digest('hex');
}
// Dictionary attack function
function dictionaryAttack() {
   let found = false;
   // Try each word in the wordlist
   for (let word of wordlist) {
      const hash = sha256(word);
      if (hash === targetHash) {
         console.log(`Match found! Input: ${word}`);
         found = true;
         break;
      }
   }
   if (!found) {
        console.log("No match found.");
    }
}
// Start dictionary attack
dictionaryAttack();
```

RESULT:

Upon performing the brute force and dictionary methods, the following observations were made:

1. **Brute Force**: The brute force method was computationally expensive and time-consuming, requiring significant processing power for even short plaintexts.
2. **Dictionary Attack**: The dictionary method was faster but limited to the words or phrases included in the dictionary. If the plaintext was not in the dictionary, the method failed to recover the original plaintext.
3. **Hash Comparison**: In cases where the plaintext matched an entry in the dictionary, the correct hash was found quickly. However, for more complex or random inputs, both methods were ineffective.

CONCLUSION:

The experiment demonstrated the challenges of reversing a SHA-256 hash. While the dictionary method proved effective for common words or phrases, it was not useful for more complex or random plaintexts. The brute force method, though theoretically capable of determining the plaintext, was impractical due to its high computational cost. This reinforces the strength of SHA-256 in securing data, as both methods are inefficient for strong, unpredictable inputs.

VIVA QUESTIONS:

1) What is SHA-256 and how does it fit into the family of SHA hash functions?
2) How does SHA-256 handle different input sizes?
3) What is the role of the initial hash values in SHA-256?
4) How would you approach solving a SHA-256 hash using brute force?
5) Can you explain what a dictionary attack is and how it differs from brute force?

Prepared by : Dr. N. M. Kandoi	Approved by Head : Dr. J.M. Patil

	SHRI SANT GAJANAN MAHARAJ COLLEGE OF ENGG, SHEGAON	**LABORATORY MANUAL**
	PRACTICAL EXPERIMENT INSTRUCTION SHEET	
	EXPERIMENT TITLE: Implementing Proof of Work with Dynamic Difficulty Levels in a Blockchain System Write a JavaScript program that performs the following tasks: 1. **Finds a Valid Nonce**: Identifies a valid nonce and its corresponding hash for a block that satisfies difficulty levels ranging from 1 to 10. 2. **Meets Difficulty Requirements**: Ensures the hash starts with a number of leading zeros equal to the difficulty level. **Computational Time Measurement**: Calculates and displays the time taken to find the valid nonce for each difficulty level.	

EXPERIMENT NO. SSGMCE/WI/CSE/01/8KS06/10	ISSUE NO. : 00	ISSUE DATE : 01-01-2025
REV. DATE :	REV. NO. : 01	DEPTT. : COMPUTER SCIENCE AND ENGINEERING
LABORATORY : EMERGING TECHNOLOGY LAB VI (DLT)	SEMESTER : VIII	PAGE :1 OF 5

AIM: Implementing Proof of Work with Dynamic Difficulty Levels in a Blockchain System
Write a JavaScript program that performs the following tasks:

1. **Finds a Valid Nonce**: Identifies a valid nonce and its corresponding hash for a block that satisfies difficulty levels ranging from 1 to 10.
2. **Meets Difficulty Requirements**: Ensures the hash starts with a number of leading zeros equal to the difficulty level.
3. **Computational Time Measurement**: Calculates and displays the time taken to find the valid nonce for each difficulty level.

OBJECTIVE:

- To simulate the POW algorithm with dynamic difficulty levels in a blockchain.
- To identify a valid nonce and its corresponding hash for a block that satisfies difficulty levels ranging from 1 to 6.
- To ensure the hash starts with the required number of leading zeros corresponding to the difficulty level.
- To measure and display the computational time taken to find the valid nonce for each difficulty level.

SCOPE:

- The program will implement a basic POW algorithm where the difficulty level determines how many leading zeros the block hash must have.
- Difficulty levels will range from 1 to 10, with the program dynamically adjusting the number of leading zeros required for the hash.
- Program will be written in JavaScript and will simulate the PoW process for a single block.
- The computational time for each difficulty level will be recorded and displayed.

FACILITIES: Computer System, VS Code, Java Script.

THEORY/EXPERIMENT PROCEDURE:

- **Proof of Work (PoW):** A consensus algorithm used in blockchain networks, where miners must solve computationally difficult puzzles to add new blocks to the blockchain.
- **Nonce:** A number used once to modify the block's hash and ensure that it meets the difficulty requirement.
- **Difficulty Level:** The number of leading zeros in the block hash. The higher the difficulty, the more zeros required, making the computation harder.
- **Hash Function:** A cryptographic function that generates a fixed-size output (hash) from an input (data). In POW, hash must satisfy difficulty requirement (e.g., a certain number of leading zeros).

  This experiment simulates Proof of Work (PoW) for a blockchain system. It dynamically adjusts difficulty levels from 1 to 6 and finds a valid nonce such that the block's hash meets the difficulty criteria.

Explanation:

1. Importing the Crypto Module

```
const crypto = require('crypto');
```

The crypto module is a built-in Node.js library used to perform cryptographic functions.

Here, it is used to generate SHA-256 hashes.

2. Defining Block Data

```
const blockData = "Block #1: Alice sends 5 BTC to Bob";
```

This is the data of the block. It represents a simple transaction where Alice sends 5 BTC to Bob.

3. Hash Function

```
function computeHash(data) {
return crypto.createHash('sha256').update(data).digest('hex');
}
```

computeHash(data) generates a SHA-256 hash of the input data.

The update(data) function feeds the data into the hashing algorithm.

The digest('hex') converts the hash output into a hexadecimal string.

Example:

For computeHash("Hello"), the output will be:

```
185f8db32271fe25f561a6fc938b2e264306ec304eda518007d1764826381969
```

4. Proof of Work Function

```
function proofOfWork(blockData, difficulty) {
  const target = "0".repeat(difficulty) + "f".repeat(64 - difficulty);
  let nonce = 0;
 const startTime = Date.now(); // Start time for computation
 while (true) {
 // Concatenate the block data and nonce
  const data = blockData + nonce;
  // Compute the hash
  const hash = computeHash(data);
 // Check if the hash meets the difficulty requirement
 if (hash.startsWith("0".repeat(difficulty))) {
   const endTime = Date.now(); // End time for computation
   return { nonce, hash, timeTaken: (endTime - startTime) / 1000 }; // Time in seconds
  }
  // Increment the nonce and try again
  nonce++;
  }
}
```

Explanation of proofOfWork function:

Defining the Target Hash Condition

```
const target = "0".repeat(difficulty) + "f".repeat(64 - difficulty);
```

The difficulty level determines how many leading zeros the hash must have.

The target string is created with:

"0".repeat(difficulty): Generates difficulty number of leading zeros.

"f".repeat(64 - difficulty): Fills the rest of the 64-character hash with "f" (hex representation of the highest possible value).

Initializing the Nonce and Timer

```
let nonce = 0;
const startTime = Date.now();
```

The nonce is initialized to 0 and will be incremented until a valid hash is found.

startTime records when the mining process begins.

Brute Force Loop (Mining Process)

```
while (true) {
   const data = blockData + nonce;
 const hash = computeHash(data);
 if (hash.startsWith("0".repeat(difficulty))) {
   const endTime = Date.now();
     return { nonce, hash, timeTaken: (endTime - startTime) / 1000 };
   }
   nonce++;
}
```

The program keeps incrementing the nonce and calculating the hash.

It checks whether the hash starts with the required number of leading zeros.

If the condition is met, the function returns: valid nonce, valid hash, and time taken to find the nonce

5. Running Proof of Work for Difficulty Levels 1 to 6

```
for (let difficulty = 1; difficulty <= 10; difficulty++) {
   console.log(`\nRunning Proof of Work for Difficulty Level: ${difficulty}`);
  const result = proofOfWork(blockData, difficulty);
 console.log(`Block Data: ${blockData}`);
   console.log(`Difficulty Level: ${difficulty}`);
  console.log(`Valid Nonce: ${result.nonce}`);
  console.log(`Valid Hash: ${result.hash}`);
 console.log(`Time Taken: ${result.timeTaken} seconds`);
}
```

The program loops over difficulty levels from 1 to 6

Calls proofOfWork(blockData, difficulty), passing the block data and difficulty.

Displays:

Block Data , Difficulty Level , Valid Nonce, Valid Hash, Time Taken

Example Execution Output

```
Running Proof of Work for Difficulty Level: 1
Block Data: Block #1: Alice sends 5 BTC to Bob
Difficulty Level: 1
Valid Nonce: 2
Valid Hash: 0a4b35e66e34364b7d2732e5b3ddcfb3c4e1c0c1e2a6d8b9e45a37b657e89000
Time Taken: 0.002 seconds
Running Proof of Work for Difficulty Level: 2
Block Data: Block #1: Alice sends 5 BTC to Bob
Difficulty Level: 2
Valid Nonce: 17
Valid Hash: 005a9db4a22bba512d8db3e837b9de6f2a64ffbe760e18cd317f7c84a8264532
Time Taken: 0.012 seconds
...
Running Proof of Work for Difficulty Level: 10
Block Data: Block #1: Alice sends 5 BTC to Bob
Difficulty Level: 10
Valid Nonce: 5402931
Valid Hash: 00000000003f6d7ad5d8e41c69327b40bfa573e6126c4b3b5eb91ad2b7c4fa8c
Time Taken: 12.45 seconds
```

Higher difficulty requires more leading zeros, leading to higher computation time.

Difficulty 1 takes milliseconds, while difficulty 10 takes several seconds.

Nonce Finding:

The program brute forces through nonce values to find a valid hash.

Hash Computation: Uses SHA-256 to generate hashes.

Dynamic Difficulty Adjustment: Difficulty increases, number of leading zeros required in hash increases.

Computational Cost: Higher difficulty levels require exponentially more time to compute.

Real-World Application: Bitcoin Mining

This program mimics Bitcoin's mining process, where miners compete to find a valid nonce.

The Bitcoin network adjusts difficulty based on network speed.

Security in Blockchain: PoW ensures security by making it computationally expensive to alter blockchain.

RESULT:

- The program will successfully identify a valid nonce for each difficulty level (1 to 10).
- For each difficulty level, the program will display the corresponding hash and nonce.
- The time taken to find the valid nonce for each difficulty level will be calculated and displayed, showing how the computational time increases as the difficulty level increases.

CONCLUSION:

- The experiment demonstrates the relationship between difficulty level and computational time in the Proof of Work algorithm.
- As the difficulty level increases, the time required to find a valid nonce also increases, highlighting the computational cost of securing blockchain networks.

The program successfully simulates the dynamic nature of difficulty adjustment in blockchain mining

VIVA QUESTIONS:

1) How do you generate the nonce in your program?
2) How does the program measure the time taken to find a valid nonce?
3) What does it mean for a hash to have a certain number of leading zeros?
4) What challenges did you face in measuring computational time, and how did you address them?
5) What happens if there is an issue with the hash functions or time measurement?

Prepared by : Dr. N. M. Kandoi	Approved by Head : Dr. J.M. Patil

	SHRI SANT GAJANAN MAHARAJ COLLEGE OF ENGG, SHEGAON	**LABORATORY MANUAL**
	PRACTICAL EXPERIMENT INSTRUCTION SHEET	
	EXPERIMENT TITLE: Implement and demonstrate the concept of **key splitting** and **secure sharing** using **Shamir's Secret Sharing (SSS)** algorithm.	

EXPERIMENT NO. SSGMCE/WI/CSE/01/8KS06/11	ISSUE NO. : 00	ISSUE DATE : 01-01-2025
REV. DATE :	REV. NO. : 01	DEPTT. : COMPUTER SCIENCE AND ENGINEERING
LABORATORY : EMERGING TECHNOLOGY LAB VI (DLT)	SEMESTER : VIII	PAGE :1 OF 5

AIM: Implement and demonstrate the concept of **key splitting** and **secure sharing** using **Shamir's Secret Sharing (SSS)** algorithm.

OBJECTIVE:

- **To understand** the concept of key splitting and secure sharing using Shamir's Secret Sharing (SSS) algorithm, including its theoretical foundations and cryptographic significance.
- **To Apply** the Shamir's Secret Sharing algorithm in JavaScript by implementing key splitting, generating secret shares, and reconstructing the original key using a threshold-based approach.
- **To Evaluate** the security, efficiency, and real-world applications of secret sharing in cryptographic systems, comparing it with other key management techniques.

SCOPE:

- **Implementation:** Develop JavaScript program for key splitting and reconstruction using SSS.
- **Security:** Ensure secure key distribution with threshold-based access control.
- **Application:** Apply secret sharing in secure storage, key management, and authentication.

FACILITIES: Computer System, VS Code, JavaScript (Node.js), Optional (for cryptographic operations, if necessary)

THEORY/EXPERIMENT PROCEDURE:

Shamir's Secret Sharing is a cryptographic method used to divide a secret (e.g., a private key) into multiple parts, called shares, such that, a minimum number of shares (threshold) are required to reconstruct the secret and fewer than the threshold number of shares reveals no information about the secret. This threshold is defined by the parameter kkk, which determines how many shares are needed to reconstruct the secret. The algorithm is based on polynomial interpolation over a finite field, where the secret is the constant term of the polynomial. The shares are generated by evaluating the polynomial at different points.

The core idea is that if the number of shares is less than the threshold kkk, the secret cannot be reconstructed, ensuring security. This technique is widely used in secure key management systems, such as in distributed systems, cryptographic protocols, and multi-party computations.

This scheme is useful in scenarios requiring high security, such as managing cryptographic keys in distributed systems, where no single party should have complete control over the secret.

Purpose of Splitting and Reconstructing Secrets

The purpose of splitting and reconstructing secrets, such as cryptographic keys, is to ensure **security, redundancy, and fault tolerance** in managing sensitive information. By dividing a secret into multiple shares and requiring only a subset to reconstruct it, this method mitigates the risks associated with single points of failure or unauthorized access.

Real-Life Examples (Shortened)

1. **Bank Vault Access** – A vault's access code is split among managers (e.g., 3 out of 5 needed) to prevent unauthorized access.
2. **Cryptocurrency Wallets** – A private key is shared among trusted individuals, requiring a minimum number of shares to recover funds.
3. **Nuclear Launch Codes** – Launch codes are divided among officials, requiring collective authorization for security.
4. **Corporate Decision-Making** – Sensitive data access is shared among board members, needing a quorum to reconstruct encryption keys.
5. **Disaster Recovery** – Encryption keys for backups are split and stored in different locations to ensure recovery if some is lost.

Key Benefits

1. **Security** – No single person has full access, reducing theft or misuse.
2. **Redundancy** – Shares are distributed to ensure recovery even if some are lost.
3. **Fault Tolerance** – The secret can be reconstructed even if some shares are inaccessible.
4. **Collaborative Decision-Making** – Multiple parties must contribute, ensuring accountability.
5. **Scalability** – Share distribution can be customized for security needs.
6. **Privacy** – Partial leaks do not reveal the secret.

Key Concepts

- **Splitting** – A secret is divided into **n** shares using a polynomial.
- **Reconstruction** – At least **t** shares are needed to recover the secret via polynomial interpolation.

Algorithm:

1. **Secret Initialization:**
   Choose a secret that will be shared. This secret will be the constant term of a polynomial.
2. **Polynomial Generation:**
   Generate a random polynomial of degree k−1k-1k−1, where the constant term is the secret, and the other coefficients are random integers. The degree of the polynomial determines the threshold kkk.
3. **Share Generation:**
   Evaluate the polynomial at nnn distinct points to create nnn shares. Each share consists of a

pair of values (x,y)(x, y)(x,y), where xxx is the position and yyy is the value of the polynomial at that position.

4. **Secret Reconstruction:**
   Given any kkk shares, reconstruct the polynomial using **Lagrange interpolation**. The secret is then extracted from the constant term of the reconstructed polynomial.

Implementation Steps:

1. **Setup Environment:**
   Use JavaScript (Node.js) for the implementation of the algorithm. Optionally, install necessary dependencies for cryptographic operations.
2. **Key Splitting:**
   Implement the function to generate shares based on Shamir's Secret Sharing. The function should take the secret, total number of shares, and the threshold kkk as inputs.
3. **Key Reconstruction:**
   Implement the reconstruction function using Lagrange interpolation to combine the shares and recover the original secret.

Testing:
Run tests by creating a secret, generating shares, and then selecting different combinations of shares to test the reconstruction process.

```
const crypto = require('crypto');
const sss = require('shamirs-secret-sharing');
// Polyfill for globalThis.crypto.getRandomValues
globalThis.crypto = {
  getRandomValues: (buffer) => crypto.randomFillSync(buffer),
};
console.log('Start');
// Define the private key
const privateKeyHex = 'e331b6d69882b4c4c3fb6c3eabf9f5a3e6a0e3c3a6b3d9c9e2b2a3d2c3e1f9a1';
console.log('Private Key:', privateKeyHex);
// Convert private key to a Buffer
const privateKeyBuffer = Buffer.from(privateKeyHex, 'hex');
console.log('Private Key Buffer (before split):', privateKeyBuffer);
// Split the private key into shares
const totalShares = 5;
const threshold = 3;
const shares = sss.split(privateKeyBuffer, { shares: totalShares, threshold });
console.log('Shares Array:', shares);
 // Simulate combining shares to recover the private key
try {
  const recoveredPrivateKeyBuffer = sss.combine(shares.slice(0, threshold)); // Use only the
threshold number of shares
  console.log('Recovered Private Key Buffer:', recoveredPrivateKeyBuffer);
  console.log('Recovered Private Key:', recoveredPrivateKeyBuffer.toString('hex'));
} catch (error) {
  console.error('Error combining shares:', error.message);
}
```

Expected Outcome:

```
Private Key: e331b6d69882b4c4c3fb6c3eabf9f5a3e6a0e3c3a6b3d9c9e2b2a3d2c3e1f9a1
Private Key Buffer (before split): <Buffer e3 31 b6 d6 98 82 b4 c4 c3 fb 6c 3e ab f9 f5 a3 e6 a0 e3
c3 a6 b3 d9 c9 e2 b2 a3 d2 c3 e1 f9 a1>
Shares Array: [
  _Buffer(82) [Uint8Array] [
      8,   1,  96, 170,  70, 191, 186, 146, 105, 202, 189,  12,
    171, 220, 154,  75, 112, 150, 196, 184,  75,  35, 235, 166,
    218, 114, 178,  89,  30, 214,  69,  56,  91,  74, 233,   5,
    106, 119, 126, 166, 154, 165, 138, 253,  22,  17,   9,   6,
     56,  50, 128, 135,   3,  32,  57, 188,   5,  17, 136, 139,
     17, 202,  86,  11,  97, 220,  16,  50, 150,  92, 110, 216,
     96, 171,  38,  15, 171, 169, 197, 170, 104,  55
  ],
  _Buffer(82) [Uint8Array] [
     8,   2, 194, 101,  53, 220, 141, 252, 143,   6,  91, 118,
    41,  41, 230, 145,  27, 213, 189,  35, 192,  58,  41,  53,
    10,  20, 128, 223, 161, 215,  48, 152, 105, 231, 187, 240,
     7,  98, 116, 235,  95, 109,  26, 246,  54,  77, 175, 232,
    76, 148, 218, 106,  91, 188,  39,  84, 237, 169, 141, 255,
    36,  62, 208,  48,  93, 128,  35, 173, 161,  16, 221,  41,
    53, 129,  87, 210,  64,  53,  88, 246, 171, 130
  ],
  _Buffer(82) [Uint8Array] [
      8,   3, 162, 207, 115,  99,  55, 110, 230, 204, 230, 122,
    130, 245, 124, 218, 107,  66, 121,  58, 139, 224, 194, 114,
    208, 165,  50,  84, 191, 162, 117,  18,  50,  79,  82,  60,
    109, 204,  10, 254, 197, 110, 144, 200,  32, 191, 166,  78,
    116,  64,  90,  78,  88, 105,  30,  17, 232,  19,   5,  74,
     53, 152, 134, 192,  60, 159,  51,  91,  55, 248, 179, 115,
     85, 178, 113,  11, 235,  42, 157, 109, 195,  86
  ],
  _Buffer(82) [Uint8Array] [
      8,   4, 145, 122, 180,  99, 176, 214, 106,  10,  70,  73,
    199,  88, 202,  35, 253,  15, 199,  66, 216, 215, 253,  33,
    162, 234, 222, 208,  17,   9, 178, 106, 137, 213, 166, 113,
    101, 125, 242, 214, 123,  62, 120, 144,   4, 227, 141,  84,
    104,  57, 146,  18, 223, 184,   7,  34, 124,  30,  61,  67,
     80, 196,  80, 168, 252, 172,  74,  76,  37,  88, 163, 181,
    153,  75, 194,  72, 172,  99, 171,  53, 186,  86
  ],
  _Buffer(82) [Uint8Array] [
      8,   5, 241, 208, 242, 220,  10,  68,   3, 192, 251,  69,
    108, 132,  80, 104, 141, 152,   3,  91, 147,  13,  22, 102,
    120,  91, 108,  91,  15, 124, 247, 224, 210, 125,  79, 189,
     15, 211, 140, 195, 225,  61, 242, 174,  18,  17, 132, 242,
     80, 237,  18,  54, 220, 109,  62, 103, 121, 164, 181, 246,
     65,  98,   6,  88, 157, 179,  90, 186, 179, 176, 205, 239,
    249, 120, 228, 145,   7, 124, 110, 174, 210, 130
  ]
]
Recovered Private Key Buffer: _Buffer(32) [Uint8Array] [
  227,  49, 182, 214, 152, 130, 180,
  196, 195, 251, 108,  62, 171, 249,
  245, 163, 230, 160, 227, 195, 166,
  179, 217, 201, 226, 178, 163, 210,
  195, 225, 249, 161
]
Recovered Private Key: e331b6d69882b4c4c3fb6c3eabf9f5a3e6a0e3c3a6b3d9c9e2b2a3d2c3e1f9a1
```

- The secret should be successfully split into shares and securely reconstructed when the minimum threshold of shares is used.
- The secret cannot be reconstructed with fewer than the required number of shares, demonstrating the security of the scheme.

RESULT:

1. The private key was successfully split into 5 shares with a threshold of 3.
2. Using any 3 of the 5 shares, the private key was accurately reconstructed.
3. The reconstructed private key matched the original, confirming the integrity of the process.

CONCLUSION:

The experiment demonstrates that Shamir's Secret Sharing effectively splits and reconstructs a cryptographic private key. This ensures secure distribution of sensitive information, as fewer than the threshold number of shares cannot compromise the secret. The method is highly applicable in distributed systems and cryptographic key management.

VIVA QUESTIONS:

1) Explain the purpose and working of the SSS algorithm?
2) What is the significance of the threshold in Shamir’s Secret Sharing?
3) What are the main components of the Shamir’s Secret Sharing algorithm?
4) What is a "dealer" in Shamir's Secret Sharing scheme?
5) How can Shamir’s Secret Sharing be integrated into a real-world application?

Prepared by : Dr. N. M. Kandoi	Approved by Head : Dr. J.M. Patil

	SHRI SANT GAJANAN MAHARAJ COLLEGE OF ENGG, SHEGAON	**LABORATORY MANUAL**
	PRACTICAL EXPERIMENT INSTRUCTION SHEET	
	EXPERIMENT TITLE: Implementation and Validation of Digital Signatures Using Asymmetric Cryptography with Public and Private Key Pairs.	

EXPERIMENT NO. SSGMCE/WI/CSE/01/8KS06/12	ISSUE NO. : 00	ISSUE DATE : 01-01-2025
REV. DATE :	REV. NO. : 01	DEPTT. : COMPUTER SCIENCE AND ENGINEERING
LABORATORY : EMERGING TECHNOLOGY LAB VI (DLT)	SEMESTER : VIII	PAGE :1 OF 5

AIM: Implementation and Validation of Digital Signatures Using Asymmetric Cryptography with Public and Private Key Pairs.

OBJECTIVE:
- To generate an RSA key pair (public and private key).
- To create and verify a digital signature for a given message.
- To analyze the effect of message tampering corrupted signatures, and using different public keys in the verification process.
- To understand the role of cryptographic hashing and RSA in ensuring data integrity and authentication.

SCOPE:
- This experiment demonstrates digital signature creation and verification using the RSA algorithm.
- It covers cryptographic key generation, hashing, and digital signature verification.
- It applies to real-world security mechanisms such as secure email communication, digital certificates, and blockchain transactions.

FACILITIES: Computer System, VS Code, JavaScript (Node.js)

THEORY/EXPERIMENT PROCEDURE:

A **digital signature** is a cryptographic technique used to ensure data authenticity and integrity. It is generated using a private key and can be verified using a corresponding public key. The process involves:

1. **Key Pair Generation** – The RSA algorithm generates a pair of keys:
   - **Public Key**: Used for verification.
   - **Private Key**: Used for signing the message.
2. **Signing the Message** – The sender hashes the message and encrypts the hash using their private key, generating a unique **digital signature**.
3. **Verification** – The receiver decrypts the signature using the sender’s public key and compares the resulting hash with the computed hash of the received message. If they match, the signature is valid.
4. **Tampering Detection** – If the message or signature is altered, the hashes will not match, and verification will fail.

```
const crypto = require('crypto');
// Step 1: Generate Key Pair
const { publicKey, privateKey } = crypto.generateKeyPairSync('rsa', {
  modulusLength: 2048,
  publicKeyEncoding: { type: 'spki', format: 'pem' },
  privateKeyEncoding: { type: 'pkcs8', format: 'pem' },
});
console.log('Public Key:', publicKey);
console.log('Private Key:', privateKey);
// Step 2: Create a Digital Signature (Valid Case)
const message = 'This is a confidential message.';
const signer = crypto.createSign('sha256');
signer.update(message);
signer.end();
const signature = signer.sign(privateKey, 'hex');
console.log('\nOriginal Message:', message);
console.log('Digital Signature:', signature);
// Step 3: Verify the Digital Signature (Valid Case)
const verifier = crypto.createVerify('sha256');
verifier.update(message);
verifier.end();
const isValid = verifier.verify(publicKey, signature, 'hex');
console.log('Is the signature valid? (Expected: true):', isValid);
// Step 4: Verify with Tampered Message (Invalid Case)
const tamperedMessage = 'This is a modified message.';
const verifierTampered = crypto.createVerify('sha256');
verifierTampered.update(tamperedMessage);
verifierTampered.end();
const isTamperedValid = verifierTampered.verify(publicKey, signature, 'hex');
console.log('\nTampered Message:', tamperedMessage);
console.log('Is the signature valid for tampered message? (Expected: false):', isTamperedValid);
// Step 5: Verify with Corrupted Signature (Invalid Case)
const corruptedSignature = signature.slice(0, -1) + 'x';
const verifierCorrupted = crypto.createVerify('sha256');
verifierCorrupted.update(message);
verifierCorrupted.end();
const isCorruptedValid = verifierCorrupted.verify(publicKey, corruptedSignature, 'hex');
console.log('\nCorrupted Signature:', corruptedSignature);
console.log('Is the signature valid for corrupted signature? (Expected: false):', isCorruptedValid);
// Step 6: Verify with Different Public Key (Invalid Case)
```

```
const { publicKey: differentPublicKey } = crypto.generateKeyPairSync('rsa', {
  modulusLength: 2048,
  publicKeyEncoding: { type: 'spki', format: 'pem' },
  privateKeyEncoding: { type: 'pkcs8', format: 'pem' },
});
const verifierDifferentKey = crypto.createVerify('sha256');
verifierDifferentKey.update(message);
verifierDifferentKey.end();
const isDifferentKeyValid = verifierDifferentKey.verify(differentPublicKey, signature, 'hex');
console.log('\nVerification with Different Public Key:');
console.log('Is the signature valid with different public key? (Expected: false):',
isDifferentKeyValid);
```

Explanation of above code

Step 1: Generate Key Pair

```
const { publicKey, privateKey } = crypto.generateKeyPairSync('rsa', { ... });
```

Above line generate Public-private key pair using the RSA algorithm.

Public key is used to verify signature and private key to sign the data.

Purpose: Ensure asymmetric encryption where only the private key holder can create valid signatures, and anyone with the public key can verify them.

Step 2: Create a Digital Signature

```
const message = 'This is a confidential message.';
const signer = crypto.createSign('sha256');
signer.update(message);
signer.end();
const signature = signer.sign(privateKey, 'hex');
```

The original message is hashed using the SHA-256 algorithm.

The hashed output is encrypted with the private key to create a unique digital signature.

Purpose: Ensure message authenticity. The signature binds the message to the private key.

Step 3: Verify the Digital Signature (Valid Case)

```
const verifier = crypto.createVerify('sha256');
verifier.update(message);
verifier.end();
const isValid = verifier.verify(publicKey, signature, 'hex');
```

The same hashing algorithm (SHA-256) is used to hash the message again.

The signature is decrypted using the public key and compared with the freshly computed hash.

Purpose: Ensure the message and signature are untampered and the signature matches the original private key.

Result: true.

Step 4: Verify with Tampered Message (Invalid Case)

```
const tamperedMessage = 'This is a modified message.';
const isTamperedValid = verifierTampered.verify(publicKey, signature, 'hex');
```

The modified message is hashed, resulting in a different hash.

When compared to the decrypted signature, the hashes don't match.

Purpose: Demonstrates that any modification to the original message invalidates the signature.

Result: false.

Step 5: Verify with Corrupted Signature (Invalid Case)

```
const corruptedSignature = signature.slice(0, -1) + 'x';
const isCorruptedValid = verifierCorrupted.verify(publicKey, corruptedSignature, 'hex');
```

A small alteration is made to the digital signature.

The verification process fails as the corrupted signature cannot be decrypted to match the original hash.

Purpose: Shows that even minor alteration to the signature render it invalid.

Result: false.

Step 6: Verify with Different Public Key (Invalid Case)

```
const { publicKey: differentPublicKey } = crypto.generateKeyPairSync('rsa', { ... });
const isDifferentKeyValid = verifierDifferentKey.verify(differentPublicKey, signature, 'hex');
```

A new key pair is generated.

The public key from this pair is used for verification, but it doesn’t match the private key that signed the original message.

Purpose: Demonstrates that the signature is unique to a specific public-private key pair.

Real-Life Example:

- **E-Governance & Digital Certificates**: Digital signatures are widely used in government portals to authenticate documents (e.g., Aadhaar verification in India).
- **Secure Email Communication**: Emails signed with digital signatures ensure they are not altered during transmission.
- **Cryptocurrency Transactions**: Bitcoin and other cryptocurrencies use digital signatures to validate transactions and prevent forgery.
- **Software Distribution**: Developers sign software updates to confirm their authenticity and protect against tampered versions.

Application:

- **Data Authentication**: Ensures that the message comes from the intended sender.
- **Data Integrity**: Protects against unauthorized modifications.
- **Non-Repudiation**: The sender cannot deny sending the signed message.
- **Cybersecurity**: Used in SSL/TLS protocols for secure web browsing.

RESULT:

1. The experiment successfully generates an RSA key pair.
2. A digital signature is created and validated against the original message.
3. When the message is altered, the verification fails.
4. If a corrupted signature is used, verification fails.
5. A different public key cannot verify a signature, ensuring security.

CONCLUSION:

- Digital signatures provide a secure mechanism for verifying data authenticity and integrity.
- Any change in the message, signature, or public key cause’s verification failure, ensuring security.
- Experiment demonstrates importance of cryptographic security in digital communications and transactions.

VIVA QUESTIONS:

1) Explain the process of creating a digital signature using a private key.
2) What cryptographic standards (e.g., PKCS, X.509) support the use of digital signatures?
3) In the context of blockchain technology, how are digital signatures used to ensure transaction security?
4) Can you explain a real-world example of a digital signature implementation (e.g., in banking or government digital services)?

Prepared by : Dr. N. M. Kandoi	Approved by Head : Dr. J.M. Patil

	SHRI SANT GAJANAN MAHARAJ COLLEGE OF ENGG, SHEGAON	**LABORATORY MANUAL**
	PRACTICAL EXPERIMENT INSTRUCTION SHEET	
	EXPERIMENT TITLE: Develop a Bitcoin Script Simulator with Lock Time Verification for Enhancing Transaction Security and Blockchain Integrity.	

EXPERIMENT NO. SSGMCE/WI/CSE/01/8KS06/13	ISSUE NO. : 00	ISSUE DATE : 01-01-2025
REV. DATE :	REV. NO. : 01	DEPTT. : COMPUTER SCIENCE AND ENGINEERING
LABORATORY : EMERGING TECHNOLOGY LAB VI (DLT)	SEMESTER : VIII	PAGE :1 OF 6

AIM: Develop a Bitcoin Script Simulator with Lock Time Verification for Enhancing Transaction Security and Blockchain Integrity.

OBJECTIVE:
- Simulate Bitcoin Script execution with **precise lock time verification**.
- Implement **script processing** for unlocking and locking scripts.
- Demonstrate **conditional execution** based on timestamp constraints.
- Implement **basic script operations** (OP_CHECKLOCKTIMEVERIFY, OP_HASH160, OP_EQUALVERIFY, OP_CHECKSIG).

SCOPE:
- Covers **Bitcoin Script basics**, focusing on transaction validation logic.
- Simulates **time-locked transactions** where funds are accessible only after a set time.
- Uses cryptographic hash functions (SHA-256, RIPEMD-160) for **public key verification**.
- Implements **signature validation simulation** for transaction authorization.

FACILITIES: Computer System, VS Code, JavaScript (Node.js)

THEORY/EXPERIMENT PROCEDURE:

1. Bitcoin Script and Lock Time Verification

Bitcoin transactions use **locking and unlocking scripts** to determine how funds can be spent. This experiment simulates a Bitcoin Script execution process with **lock time constraints**:
- **Locking Script (ScriptPubKey)**: Specifies spending conditions.
- **Unlocking Script (ScriptSig)**: Provides proof to meet the conditions.

2. Key Script Operations

- **OP_CHECKLOCKTIMEVERIFY (CLTV)**: Prevents execution before a specified timestamp.
- **OP_HASH160**: Hashes a public key (SHA-256 → RIPEMD-160).
- **OP_EQUALVERIFY**: Ensures the provided hash matches the expected one.
- **OP_CHECKSIG**: Validates the digital signature.

3. Execution Flow

1. The **unlocking script** is processed, pushing signature and public key onto the stack.
2. The **locking script** runs:
    - Checks if currentTimestamp meets the lock time.
    - Performs **hash verification** and **signature validation**.
    - Approves or rejects the transaction based on conditions.

This Experiment simulates:

- The **processing of unlocking and locking scripts** in a Bitcoin transaction.
- The use of specific opcodes like OP_CHECKLOCKTIMEVERIFY (CLTV), OP_HASH160, and OP_CHECKSIG.
- A **timestamp-based lock mechanism** (OP_CHECKLOCKTIMEVERIFY) to ensure funds cannot be spent before a specific time.

Bitcoin script consists of two main components:

- **Unlocking Script (ScriptSig)**: This script provides the necessary conditions to unlock the transaction output.
- **Locking Script (ScriptPubKey)**: This script contains the conditions that must be met to spend the output.

This experiment demonstrates how these scripts are executed, with particular focus on:

1. **Lock Time Check (OP_CHECKLOCKTIMEVERIFY)**: This opcode ensures that a transaction cannot be spent before a specific lock time.
2. **Digital Signature Verification (OP_CHECKSIG)**: This opcode verifies that the provided signature matches the expected public key.
3. **Stack Operations**: Operations like OP_DUP (duplicate top stack element), OP_DROP (remove top stack element), and others are used to manipulate the script execution stack.

The cryptographic operations used here, such as hashing via SHA-256 and RIPEMD-160, are central to Bitcoin's security and transaction validation.

```
const crypto = require('crypto');
// Simulated Bitcoin Script execution with precise lock time check and timestamp display
function executeScript(unlocking, locking, currentTimestamp, lockTime) {
  const stack = [];
  console.log("\n--- Executing Script ---");
  // Process unlocking script (ScriptSig)
  console.log("Unlocking Script Processing:");
  for (const element of unlocking) {
    if (Buffer.isBuffer(element)) {
      stack.push(element);
    } else {
      const bufferElement = Buffer.from(element, "hex");
      console.log(`Pushing to stack: ${bufferElement.toString('hex')}`);
      stack.push(bufferElement);
    }
  }
```

```
console.log("Stack after unlocking script:", stack.map(item => item.toString('hex')));
  // Process locking script (ScriptPubKey)
  console.log("Locking Script Processing:");
  for (const element of locking) {
    if (element === "OP_CHECKLOCKTIMEVERIFY") {
      const lockTimeBuffer = Buffer.alloc(4);
      lockTimeBuffer.writeUInt32LE(lockTime, 0);
      stack.push(lockTimeBuffer);
      const lockTimeHex = lockTimeBuffer.readUInt32LE(0); // Parse as a 32-bit unsigned integer
(correct format)
      console.log(`Lock Time: ${lockTimeHex}`);
      console.log(`Current Timestamp: ${currentTimestamp}`);
      if (currentTimestamp < lockTimeHex) {
        console.log("Script failed: Lock time has not been reached.");
        return false;
      } else {
        console.log("Current TimeStamp equal or grater than Lock Time");
        console.log("So Script executed successfully.");
        return true;
      }
    } else if (element === "OP_DROP") {
      stack.pop(); // Remove the top element from the stack
    } else if (element === "OP_DUP") {
      stack.push(stack[stack.length - 1]); // Duplicate the top of the stack
    } else if (element === "OP_HASH160") {
      const data = stack.pop();
      const hash = hash160(data);
      console.log(`OP_HASH160: Hash of ${data.toString("hex")} -> ${hash.toString("hex")}`);
      stack.push(hash); // Hash the public key
    } else if (element === "OP_EQUALVERIFY") {
      const hash1 = stack.pop();
      const hash2 = stack.pop();
      console.log(`OP_EQUALVERIFY: Comparing ${hash1.toString("hex")} and
${hash2.toString("hex")}`);
      if (!hash1.equals(hash2)) {
        console.log("Script failed: Hashes do not match");
        return false;
      }
} else if (element === "OP_CHECKSIG") {
      // Ensure the public key and signature are on the stack in correct order
      const publicKey = stack.pop(); // Public Key first
      const signature = stack.pop(); // Signature second
      if (!signature || !publicKey) {
        console.error("Error: Missing signature or public key.");
        return false;
      }
      console.log(`Public Key: ${publicKey.toString("hex")}`);
      console.log(`Signature: ${signature.toString("hex")}`);
      // Simulate the signature check (use a crypto library to do actual validation in real use)
      if (signature.toString("hex") === publicKey.toString("hex")) { // Simplified signature
verification for this example
        console.log("Simulated Signature Verification: Signature matches Public Key.");
      } else {
        console.log("Error: Signature does not match the public key.");
        return false;
      }
    } else {
      stack.push(Buffer.from(element, "hex"));
    }
  }
  return stack.length === 0;
}
// Helper function for hash160 (RIPEMD-160(SHA-256))
function hash160(buffer) {
  const sha256 = crypto.createHash("sha256").update(buffer).digest();
  const ripemd160 = crypto.createHash("ripemd160").update(sha256).digest();
  return ripemd160;
}
// Example usage of the script
const unlockingScript = [
  '46616b655369676e6174757265313233343435', // Signature (in hex format)
  '46616b655369676e6174757265313233343435' // Public Key (in hex format)
];
const lockingScript = [
  'OP_CHECKLOCKTIMEVERIFY',
  'OP_DUP',
  'OP_HASH160',
  '5d3f5c4ac31d070e24a4161031d707ccda3b7815', // Public Key Hash (in hex format)
  'OP_EQUALVERIFY',
  'OP_CHECKSIG'
];
let currentTimestamp = 1737705612; // Initial current timestamp (Unix time)
const lockTime = 1737705616; // Defined lock time (Unix time)
let hasExecutedAfterLockTime = false; // Flag to track if the loop has executed after lockTime
while (true) {
  console.log("---------------------------");
  console.log(`Current Timestamp: ${currentTimestamp}`);
  console.log(`Lock Time: ${lockTime}`);
  const result = executeScript(unlockingScript, lockingScript, currentTimestamp, lockTime);
  // Check the result based on currentTimestamp
  if (currentTimestamp < lockTime) {
    if (!result) {
      // Only print the "Lock time has not been reached" message if the script failed and it's before lock
time
     // console.log("Script failed: Lock time has not been reached.");
    }
  } else {
    if (!hasExecutedAfterLockTime) {
      //console.log("Script executed after lock time.");
      hasExecutedAfterLockTime = true; // Mark that we've executed after lockTime
      break; // Exit the loop after the extra iteration
    }   }
  // Increment currentTimestamp by 10 seconds for the next iteration
  currentTimestamp += 10;
}
```

Program Details

Imports Crypto Module

Used for **hashing functions** (SHA-256 and RIPEMD-160) to verify public keys.

Executes Bitcoin Script

Processes **Unlocking Script (ScriptSig)** → Pushes **signature & public key** onto the stack

Processes **Locking Script (ScriptPubKey)** → Runs **CLTV, hash verification, and signature check**

Check Lock Time (CLTV)

If currentTimestamp **is less than** lockTime, the script **fails**.

If currentTimestamp **meets or exceeds** lockTime, the script **executes successfully**.

Simulates Other Script Operations

- **OP_DUP** → duplicates the public key.
- **OP_HASH160** → Hashes the public key for verification.
- **OP_EQUALVERIFY** → Ensures hash matches the expected value.
- **OP_CHECKSIG** → validates the signature against the public key.

Simulates Execution over Time

- Runs in a loop, **incrementing time** every 10 seconds.
- **Before lock time → fails**, **after lock time → succeeds** and exits.

Sample Output:

```
Current Timestamp: 1737705612
Lock Time: 1737705616
--- Executing Script ---
Unlocking Script Processing:
```

```
Pushing to stack: 46616b655369676e61747572653132333435
Pushing to stack: 46616b655369676e61747572653132333435
Stack after unlocking script: [
  '46616b655369676e61747572653132333435',
  '46616b655369676e61747572653132333435']
Locking Script Processing:
Lock Time: 1737705616
Current Timestamp: 1737705612
Script failed: Lock time has not been reached.
--------------------------
Current Timestamp: 1737705622
Lock Time: 1737705616
--- Executing Script ---
Unlocking Script Processing:
Pushing to stack: 46616b655369676e61747572653132333435
Pushing to stack: 46616b655369676e61747572653132333435
Stack after unlocking script: [
  '46616b655369676e61747572653132333435',
  '46616b655369676e61747572653132333435']
```

Locking Script Processing:
Lock Time: 1737705616
Current Timestamp: 1737705622
Current TimeStamp equal or greater than Lock Time
So Script executed successfully.

RESULT:

Before Lock Time: The script **fails** with "Lock time has not been reached."

After Lock Time: The script **executes successfully**, validating the signature and hash.

CONCLUSION:

- **CLTV (**Check Lock Time Verify) **ensures time-based security**: Transactions cannot be executed before a predefined time.
- **Bitcoin Script enforces access rules**: Funds are only unlocked under strict cryptographic conditions.
- **Digital signatures validate ownership**: Ensuring that only authorized parties can spend funds.
- **Practical application**: Used in **time-locked contracts**, multi-signature wallets, and security measures in Bitcoin transactions.

VIVA QUESTIONS:

1) What is Bitcoin Script, and how does it contribute to Bitcoin transactions?
2) What is the purpose of Lock Time in Bitcoin transactions?
3) Can you explain the difference between nLockTime and CheckLockTimeVerify (CLTV) in Bitcoin?

Prepared by : Dr. N. M. Kandoi	Approved by Head : Dr. J.M. Patil