a) A simple client class that generates the private and public keys by using the built in Python RSA algorithm and test it

b) A transaction class to send and receive money and test it

Code#

# import libraries

import hashlib

import random

import string

import json

import binascii

import numpy as np

import pandas as pd

import pylab as pl

import logging

import datetime

import collections

pip install pycryptodome

# following imports are required by PKI

import Crypto

import Crypto.Random

from Crypto.Hash import SHA

from Crypto.PublicKey import RSA

from Crypto.Signature import PKCS1\_v1\_5

import binascii

class Client:

def \_\_init\_\_(self):

random = Crypto.Random.new().read

self.\_private\_key = RSA.generate(1024, random)

self.\_public\_key = self.\_private\_key.publickey()

self.\_signer = PKCS1\_v1\_5.new(self.\_private\_key)

@property

def identity(self):

return binascii.hexlify(self.\_public\_key.exportKey(format='DER')).decode('ascii')

class Transaction:

def \_\_init\_\_(self, sender, recipient, value):

self.sender = sender

self.recipient = recipient

self.value = value

self.time = datetime.datetime.now()

def to\_dict(self):

if self.sender == "Genesis":

identity = "Genesis"

else:

identity = self.sender.identity

return collections.OrderedDict({

'sender': identity,

'recipient': self.recipient,

'value': self.value,

'time' : self.time})

def sign\_transaction(self):

private\_key = self.sender.\_private\_key

signer = PKCS1\_v1\_5.new(private\_key)

h = SHA.new(str(self.to\_dict()).encode('utf8'))

return binascii.hexlify(signer.sign(h)).decode('ascii')

Dinesh = Client()

Ramesh = Client()

t = Transaction(Dinesh,Ramesh.identity,5.0)

signature = t.sign\_transaction()

print (signature)

2. a) Create multiple transactions and display them.

b) Create a blockchain, a genesis block and execute it.

Code#

def display\_transaction(transaction):

#for transaction in transactions:

dict = transaction.to\_dict()

print ("sender: " + dict['sender'])

print ('-----')

print ("recipient: " + dict['recipient'])

print ('-----')

print ("value: " + str(dict['value']))

print ('-----')

print ("time: " + str(dict['time']))

print ('-----')

transactions = []

Dinesh = Client()

Ramesh = Client()

Seema = Client()

Vijay = Client()

t1 = Transaction( Dinesh, Ramesh.identity, 15.0)

t1.sign\_transaction()

transactions.append(t1)

t2 = Transaction( Dinesh, Seema.identity, 6.0)

t2.sign\_transaction()

transactions.append(t2)

t3 = Transaction( Ramesh, Vijay.identity, 2.0)

t3.sign\_transaction()

transactions.append(t3)

t4 = Transaction( Seema, Ramesh.identity, 4.0)

t4.sign\_transaction()

transactions.append(t4)

t5 = Transaction( Vijay, Seema.identity, 7.0)

t5.sign\_transaction()

transactions.append(t5)

t6 = Transaction( Ramesh, Seema.identity, 3.0)

t6.sign\_transaction()

transactions.append(t6)

t7 = Transaction( Seema, Dinesh.identity, 8.0)

t7.sign\_transaction()

transactions.append(t7)

t8 = Transaction( Seema, Ramesh.identity, 1.0)

t8.sign\_transaction()

transactions.append(t8)

t9 = Transaction( Vijay, Dinesh.identity, 5.0)

t9.sign\_transaction()

transactions.append(t9)

t10 = Transaction( Vijay, Ramesh.identity, 3.0)

t10.sign\_transaction()

transactions.append(t10)

for transaction in transactions:

display\_transaction (transaction)

print ('--------------')

class Block:

def \_\_init\_\_(self):

self.verified\_transactions = []

self.previous\_block\_hash = ""

self.Nonce = ""

last\_block\_hash = ""

Dinesh = Client()

t0 = Transaction ( "Genesis", Dinesh.identity, 500.0)

block0 = Block()

block0.previous\_block\_hash = None

Nonce = None

block0.verified\_transactions.append (t0)

digest = hash (block0)

last\_block\_hash = digest

TPCoins = []

def dump\_blockchain (self):

print ("Number of blocks in the chain: " + str(len (self)))

for x in range (len(TPCoins)):

block\_temp = TPCoins[x]

print ("block # " + str(x))

for transaction in block\_temp.verified\_transactions:

display\_transaction (transaction)

print ('--------------')

print ('=====================================')

TPCoins.append (block0)

dump\_blockchain(TPCoins)

3. a) Create a mining function and test it.

b) Add blocks to the miner and dump the blockchain.

Code#

def sha256(message):

return hashlib.sha256(message.encode('ascii')).hexdigest()

def mine(message, difficulty=1):

assert difficulty >= 1

prefix = '1' \* difficulty

for i in range(1000):

digest = sha256(str(hash(message)) + str(i))

if digest.startswith(prefix):

print ("after " + str(i) + " iterations found nonce: "+ digest)

return digest

last\_transaction\_index = 0

block = Block()

for i in range(3):

temp\_transaction = transactions[last\_transaction\_index]

# validate transaction

# if valid

block.verified\_transactions.append (temp\_transaction)

last\_transaction\_index += 1 mine ("test message", 2)

block.previous\_block\_hash = last\_block\_hash

block.Nonce = mine (block, 2)

digest = hash (block)

TPCoins.append (block)

last\_block\_hash = digest

# Miner 2 adds a block

block = Block()

for i in range(3):

temp\_transaction = transactions[last\_transaction\_index]

# validate transaction

# if valid

block.verified\_transactions.append (temp\_transaction)

last\_transaction\_index += 1

block.previous\_block\_hash = last\_block\_hash

block.Nonce = mine(block, 2)

digest = hash (block)

TPCoins.append (block)

last\_block\_hash = digest

# Miner 3 adds a block

block = Block()

for i in range(3):

temp\_transaction = transactions[last\_transaction\_index]

#display\_transaction (temp\_transaction)

# validate transaction

# if valid

block.verified\_transactions.append (temp\_transaction)

last\_transaction\_index += 1

block.previous\_block\_hash = last\_block\_hash

block.Nonce = mine (block, 2)

digest = hash (block)

TPCoins.append (block)

last\_block\_hash = digest

dump\_blockchain(TPCoins)

4.Implement and demonstrate the use of the following in Solidity: a) Varaible b) Operators c) Loops d) Decision Making e) Strings

Code:-

#a.Variable

pragma solidity ^0.5.0;

contract SolidityTest {

uint storedData; // State variable

constructor() public {

storedData = 10;

}

function getResult() public view returns(uint){

uint a = 1; // local variable

uint b = 2;

uint result = a + b;

return storedData; //access the state variable

}

}

#start // Solidity program to demonstrate state variables

pragma solidity ^0.5.0;

// Creating a contract

contract Solidity\_var\_Test {

// Declaring a state variable

uint8 public state\_var;

// Defining a constructor

constructor() public {

state\_var = 16;

}

}

#start // Solidity program to show Global variables

pragma solidity ^0.5.0;

// Creating a contract

contract Test {

// Defining a variable

address public admin;

// Creating a constructor to

// use Global variable

constructor() public {

admin = msg.sender;

}

}

#b.Operators

// Solidity contract to demonstrate Arithematic Operator

pragma solidity ^0.5.0;

// Creating a contract

contract SolidityTest {

// Initializing variables

uint16 public a = 20;

uint16 public b = 10;

// Initializing a variable with sum

uint public sum = a + b;

// Initializing a variable with the difference

uint public diff = a - b;

// Initializing a variable with product

uint public mul = a \* b;

// Initializing a variable with quotient

uint public div = a / b;

// Initializing a variable with modulus

uint public mod = a % b;

// Initializing a variable decrement value

uint public dec = --b;

// Initializing a variable with increment value

uint public inc = ++a;

}

C.Decision Making

// Solidity program to demonstrate the use of 'if statement'

pragma solidity ^0.5.0;

// Creating a contract

contract Types {

// Declaring state variable

uint i = 10;

// Defining function to demonstrate use of 'if statement'

function decision\_making() public view returns(bool){

if(i<10){

return true;

}

}

}

#start // Solidity program to demonstrate the use of 'if...else' statement

pragma solidity ^0.5.0;

// Creating a contract

contract Types {

// Declaring state variables

uint i = 10;

bool even;

// Defining function to

// demonstrate the use of

// 'if...else statement'

function decision\_making() public {

if(i%2 == 0){

even = true;

}

else{

even = false;

}

}

function getresult() public view returns(bool)

{

return even;

}

}

##(III) Strings

// Solidity program to demonstrate

// how to create a contract

pragma solidity ^0.4.23;

// Creating a contract

contract Test {

// Declaring variable

string str;

// Defining a constructor

constructor(string str\_in){

str = str\_in;

}

// Defining a function to

// return value of variable 'str'

function str\_out() public view returns(string memory){

return str;

}

}

5. Implement and demonstrate the use of the following in Solidity: a) Arrays b) Enums c) Structs d) Mappings e) Coversations f) Ether Units g) Special Varaibles

Code:-

##(I).Arrays

// Solidity program to demonstrate

// creating a fixed-size array

pragma solidity ^0.5.0;

// Creating a contract

contract Types {

// Declaring state variables

// of type array

uint[6] data1;

int[5] data;

// Defining function to add

// values to an array

function array\_example() public returns (int[5] memory, uint[6] memory){

data = [int(50), -63, 77, -28, 90];

data1 = [uint(10), 20, 30, 40, 50, 60];

}

function getresult() public view returns (int[5] memory,uint[6] memory){

return (data, data1);

}

}

##c.Structs

pragma solidity ^0.5.0;

contract test {

struct Book {

string title;

string author;

uint book\_id;

}

Book book;

function setBook() public {

book = Book('Learn Java', 'TP', 1);

}

function getBookId() public view returns (uint) {

return book.book\_id;

}

}

6-Implement and demonstrate the use of the following in Solidity: a) Functions b) View Functions c) Pure Functions d) Fallback Functions e) Function Overloading f) Mathematical Functions g) Cryptographic Functions

Code:-

##A.Functions

pragma solidity ^0.5.0;

contract SolidityTest {

function testpgmresult() public view returns(uint){

uint a = 1000; // local variable

uint b = 2000;

uint result = a + b;

return result; //access the state variable

}

}

##B.View Functions

pragma solidity ^0.5.0;

contract Test {

function getResult() public view returns(uint product, uint sum){

uint a = 1; // local variable

uint b = 2;

product = a \* b;

sum = a + b;

}

}

##C.Pure Functions

pragma solidity ^0.5.0;

contract C {

//private state variable

uint private data;

//public state variable

uint public info;

//constructor

constructor() public {

info = 10;

}

//private function

function increment(uint a) private pure returns(uint) { return a + 1; }

//public function

function updateData(uint a) public { data = a; }

function getData() public view returns(uint) { return data; }

function compute(uint a, uint b) internal pure returns (uint) { return a + b; }

}

//Derived Contract

contract E is C {

uint private result;

C private c;

constructor() public {

c = new C();

}

function getComputedResult() public {

result = compute(3, 5);

}

function getResult() public view returns(uint) { return result; }

function getData() public view returns(uint) { return c.info(); }

}

##D.Function Overloading

pragma solidity ^0.5.0;

contract Test {

function getSum(uint a, uint b) public pure returns(uint){

return a + b;

}

function getSum(uint a, uint b, uint c) public pure returns(uint){

return a + b + c;

}

function callSumWithTwoArguments() public pure returns(uint){

return getSum(1,2);

}

function callSumWithThreeArguments() public pure returns(uint){

return getSum(1,2,3);

}

}

##E.Mathematical Functions

pragma solidity ^0.5.0;

contract Test {

function callAddMod() public pure returns(uint){

return addmod(4, 5, 3);

}

function callMulMod() public pure returns(uint){

return mulmod(4, 5, 3);

}

}

##F.Cryptographic Functions

pragma solidity ^0.5.0;

contract Test {

function callsha256() public pure returns( bytes32 result){

return sha256("ronaldo");

}

function callkeccak256() public pure returns( bytes32 result){

return keccak256("ronaldo");

}

}

7. Implement and demonstrate the use of the following in Solidity: a) Contracts b) Inheritance c) Constructors d) Abstract Class e) Interfaces

Code:-

##A.Contracts

// Solidity program to

// demonstrate how to

// write a smart contract

pragma solidity >= 0.4.16 < 0.7.0;

// Defining a contract

contract Test

{

// Declaring state variables

uint public var1;

uint public var2;

uint public sum;

// Defining public function

// that sets the value of

// the state variable

function set(uint x, uint y) public

{

var1 = x;

var2=y;

sum=var1+var2;

}

// Defining function to

// print the sum of

// state variables

function get(

) public view returns (uint) {

return sum;

}

}

##B.Inheritance

// Solidity program to demonstrate Single Inheritance

pragma solidity >=0.4.22 <0.6.0;

// Defining contract

contract parent{

// Declaring internal state varaiable

uint internal sum;

// Defining external function to set value of internal state variable sum

function setValue() external {

uint a = 10;

uint b = 20;

sum = a + b;

}

}

// Defining child contract

contract child is parent{

// Defining external function to return value of internal state variable sum

function getValue() external view returns(uint) {

return sum;

}

}

// Defining calling contract

contract caller {

// Creating child contract object

child cc = new child();

// Defining function to call setValue and getValue functions

function testInheritance() public {

cc.setValue();

}

function result() public view returns(uint ){

return cc.getValue();

}

}

##C.Constructors

// Solidity program to demonstrate

// creating a constructor

pragma solidity ^0.5.0;

// Creating a contract

contract constructorExample {

// Declaring state variable

string str;

// Creating a constructor

// to set value of 'str'

constructor() public {

str = "GeeksForGeeks";

}

// Defining function to

// return the value of 'str'

function getValue(

) public view returns (

string memory) {

return str;

}

}

8#Implement and demonstrate the use of the following in Solidity: a) Libraries b) Assembly c) Events d) Error Handling

Code:-

##a) Libraries

// Solidity program to demonstrate require statement

pragma solidity ^0.5.0;

// Creating a contract

contract requireStatement {

// Defining function to check input

function checkInput(

uint \_input) public view returns(

string memory){

require(\_input >= 0, "invalid uint8");

require(\_input <= 255, "invalid uint8");

return "Input is Uint8";

}

// Defining function to use require statement

function Odd(uint \_input) public view returns(bool){

require(\_input % 2 != 0);

return true;

}

}

##b) Assembly

// Solidity program to demonstrate assert statement

pragma solidity ^0.5.0;

// Creating a contract

contract assertStatement {

// Defining a state variable

bool result;

// Defining a function to check condition

function checkOverflow(

uint \_num1, uint \_num2) public {

uint8 sum = \_num1 + \_num2;

assert(sum<=255);

result = true;

}

// Defining a function to print result of assert statement

function getResult() public view returns(string memory){

if(result == true){

return "No Overflow";

}

else{

return "Overflow exist";

}

}

}

##c) Events

// Solidity program to demonstrate assert statement

pragma solidity ^0.5.0;

// Creating a contract

contract assertStatement {

// Defining a state variable

bool result;

// Defining a function

// to check condition

function checkOverflow(uint8 sum) public {

assert(sum<=255);

result = true;

}

// Defining a function to print result of assert statement

function getResult() public view returns(string memory){

if(result == true){

return "No Overflow";

}

else{

return "Overflow exist";

}}}

##d) Error Handling

// Solidity program to demonstrate revert statement

pragma solidity ^0.5.0;

// Creating a contract

contract revertStatement {

// Defining a function to check condition

function checkOverflow(

uint \_num1, uint \_num2) public view returns(

string memory, uint) {

uint sum = \_num1 + \_num2;

if(sum < 0 || sum > 255){

revert(" Overflow Exist");}

else{ return ("No Overflow", sum);

}}}}