Implementation Summary

Initialization Phase-

- Initialization phase is implemented in the CreateShares.py file
- The correct password is taken as the input when CreateShares.py file is executed
- CreateShares.py consists of multiple functions such as - generatehpwd, encryptandshare and createhistoryfile
- generatehpwd function creates a random hardened password less than 626436561503166578967805515819693654293211766937

```
def generatehpwd(m):
    q=626436561503166578967805515819693654293211766937 #binary 160 bits Prime No.
    p=Decimal(random.random()*q) #Generate a Random hardened password
    print (p)
    return p
```

Fig 1: Snippet of the generate hardened password function

- encryptandshare creates the shares of the hardened password generated in the previous step using Shamir secret sharing scheme. Once the shares have been created, they are encrypted using the password (taken as input from user) and written onto InstructionTable.txt after base64 encoding. Encryption algorithm is AES in CFB mode

```
def encryptandshare(p,m,keys):

    #Create the shares
    shares =(tss.share_secret(m,(2*m),p,'',tss.Hash.SHA256))
    #Create instruction table file
    file = open('InstructionTable.txt', 'w')

    #Padding to ensure the key length is a multiple of 16 bytes (as required by AES)
    length = 16 - (len(keys) % 16)
    keys += chr(length) * length

    #Encrypt the shares
    for i in range(0,2*m):
        iv = Random.new().read(AES.block_size)
        encryption_suite = AES.new(keys, AES.MODE_CFB , iv)
        cipher_text0 = iv+(encryption_suite.encrypt(shares[i]))
        cipher = base64.b64encode(cipher_text0)
        file.write(cipher+' ')
        if(i%2==1):
            file.write('\n')
    file.close()
```

Fig 2: Snippet of the encrypt and share function

- createhistoryfile writes out dummy content –'000' after encrypting to History.txt file. Encryption algorithm is AES in CFB mode
- History file contains the last 5 login attempt details
- If the password has m characters, the history file would have 5* (m-1) entries
- History file contents are encrypted by the first 32 bytes (256 bits) of the hardened password before writing to History.txt file.

```
def createhistoryfile(keys,h,m):
    keys=str(keys)[0:32]
    print(keys) #32 byte key from Hpwd to encrypt history file; AES needs only 32 bytes
    keys=keys.encode()

    #Create the history file
    print ('History file size would be '+str(m)+' X '+str(h))
    a='000' #dummy content on the initialized history file
    s=0
    file=open('History.txt','w') #Create History File and encrypt dummy history file contents with hardened password
    for i in range(0,h):
        for i in range(0,m):
            iv = Random.new().read(AES.block_size)
            encryption_suite = AES.new(keys, AES.MODE_CFB, iv)
            cipher_text0 = iv + encryption_suite.encrypt(a)
            cipher=base64.b64encode(cipher_text0)
            file.write(cipher+' ')

        file.write('\n')
    file.close()
```

Fig 3: Snippet of the create history file function

- To check whether the Instruction Table has been correctly formed, one can execute the DecryptInstructions.py file. Its output, if correctly formed would be the secret hardened password
- To check whether the History Table has been correctly formed, one can execute the DecryptHistoryFileContents.py file. DecryptedHistory.txt is formed once this python script executes. If the initialization of history file has happened correctly, all values in the history file would decrypt to 000

Login Phase-

- Login.py checks for successful login attempt and perform the update of history file and instruction table after each successful login attempt.
- The path of file that contains login attempt and its corresponding feature values is given as input.

- The output of the program after checking for login attempts is stored as 0 and 1 in the output.txt file in the executing folder. Grant is represented by 1 and deny by 0 in the output.txt file.
- Login.py consists of multiple functions such as - initial_hdpwd, history, update_history, create_inst_table, decryption_hpwd, mean, standard_deviation and reconstruct_shares
- 3 global constant t(threshold value),k(kappa),h(rows in history file) are taken as given by the instructor.
- The initial_hdpwd function creates initial harden password by taking any shares which are created in 'CreateShares.py.'
- It takes the password from first genuine login attempt from first five genuine attempts.
- It then opens InstructionTable file to read any m shares from it and then decrypting them by the password and storing the decrypting shares to a list variable. This list variable than passes to reconstruct the harden password.
- After computing harden password, it returns 32 characters (160 bytes) of harden password.

```
def initial_hdpwd(pwd):
    shares = list()  # defining a empty list which can store decrypted shares
    length = 16 - (len(pwd) % 16)  # Key padding
    pwd += chr(length) * length
    pwd = pwd.encode()
    with open('InstructionTable.txt', 'r') as ins:
        for line in ins:
            words = line.split(' ')
            encrypted_share = words[0]
            cipher_text0 = base64.b64decode(encrypted_share)
            iv = cipher_text0[:AES.block_size]
            decryption_suite = AES.new(pwd, AES.MODE_CFB, iv)
            plain_text0 = decryption_suite.decrypt(cipher_text0[AES.block_size:])
            shares.append(plain_text0)
        first_hdpwd = (tss.reconstruct_secret(shares))  # reconstructing harden password from m shares
        first_hdpwd = first_hdpwd[0:32] # only taking first 32 digit as AES use keys in multiples of 16
        return Decimal(first_hdpwd) # returning harden password
    ins.close()
```

Fig 4: Snippet of the initial_hdpwd file function

- The history function updates the history file for the first five genuine attempts.
- It takes the feature value of i^{th} login, where i = 1 to 5 and harden password that was generated by the initial_hdpwd function.
- It then opens the History file for reading it's contain than storing it in a list type variable.
- We delete last feature vector entry, i.e., oldest entry of feature values in the history file.
- It then performs encryption of feature values of i^{th} login, where i = 1 to 5, and store it in the first row of the history file.
- At Last, it opens history file and override all its data by new encrypted feature values entry by the i^{th} user and then by all data which is stored in list data type variable.

```
# updating history file for first 5 genuine login attempt
def history(feature_val, hdpwd):
    hdpwd = str(hdpwd)[0:32] # padding
    hdpwd = hdpwd.encode()
    shares_history = list() # creating a list datatype to store
    temp_array = list()
# encrypting feature value
    for i in range(0, m):
        iv = Random.new().read(AES.block_size)
        encryption_suite = AES.new(hdpwd, AES.MODE_CFB, iv)
        cipher_text0 = iv + encryption_suite.encrypt(str(feature_val[i]))
        cipher = base64.b64encode(cipher_text0)
        shares_history.append(cipher)

    file = open('History.txt', 'r') # Opening History file to read its contents
    lines = file.readlines()
    lines = lines[:-1]              #deleting last entry (oldest entry), to make room for new entry
    lines = ''.join(lines)
    file.close()                    # Closing History file after read operation
    s = ''
    for i in shares_history:
        s = s + i + ' '
    file = open('History.txt', 'w') # Opening History fie for writing
    file.write(s + "\n" + lines )   # Writing  new feature value
    file.close()
```

Fig 5: Snippet of the history file function

- The update_history function updates the history file for every successful login attempt after the five genuine attempts.
- It takes the feature value of i^{th} login, where i > 5 , previous harden password that was used to encrypt history file for last successful login attempt and new random harden password from create_inst_table function after a successful login attempt.
- The update_history function updates the history file for the login attempt greater than five genuine login attempt.
- It takes the feature value of i^{th} login, where i > 5, previous harden password (hdwpd) that was used to encrypt history file for last successful login attempt and new random harden password from the create_inst_table function after a successful login attempt.
- First, it opens the History file for reading and decrypting the History file contain by hdpwd than storing it in a list type variable. Then, it encrypts it by new random harden password and delete last feature vector entry, i.e., oldest entry of feature values in the history file.
- It then performs encryption of feature values for current login, and store it in the first row of the history file.
- At Last, it opens history file and override all its data by new encrypted feature values entry by the i^{th} user and then by all data which is stored in list data type variable.

```
# Updating history file after every successful login

def update_history(feature_val,hdpwd,hard_pwd):
    hdpwd = str(hdpwd)[0:32]    #padding
    hdpwd = hdpwd.encode()
    hard_pwd = str(hard_pwd)[0:32]   #padding
    hard_pwd = hard_pwd.encode()
    shares_history = list()
    temp_array = list()
    d_history=list()

    with open('History.txt', 'r') as ins:
        for line in ins:
            line = line.strip()
            words = line.split(' ')
            for i in range(0, len(words)):
                word = words[i]
                cipher_text0 = base64.b64decode(word)
                iv = cipher_text0[:AES.block_size]
                decryption_suite = AES.new(hdpwd, AES.MODE_CFB, iv)
                plain_text0 = decryption_suite.decrypt(cipher_text0[AES.block_size:])
                temp_array.append(plain_text0)
            d_history.append(temp_array)
            temp_array = list()       #storing decrypted history context to temp

    ins.close()
#encrypting history file with new hardened password
    file=open('History.txt','w')
    for i in range(0,h):
        for j in range(0,m):
            iv = Random.new().read(AES.block_size)
            encryption_suite = AES.new(hard_pwd, AES.MODE_CFB, iv)
            cipher_text0 = iv + encryption_suite.encrypt(d_history[i][j])
            cipher=base64.b64encode(cipher_text0)
            file.write(cipher+' ')
        file.write('\n')
    file.close()
# storing latest value of feature vector
    for i in range(0, m):
        iv = Random.new().read(AES.block_size)
    encryption_suite = AES.new(hard_pwd, AES.MODE_CFB, iv)
    cipher_text0 = iv + encryption_suite.encrypt(str(feature_val[i]))
    cipher = base64.b64encode(cipher_text0)
    shares_history.append(cipher)

file = open('History.txt', 'r')
lines = file.readlines()
lines = lines[:-1]         # deleting last row to update it with the latest value of feature vector
lines = ''.join(lines)
file.close()
s = ''
for i in shares_history:
    s = s + i + ' '
file = open('History.txt', 'w') # opening history file
file.write(s + "\n" + lines )
file.close()                    # closing history file
```

Fig 6: Snippet of the update_history file function

- The create_inst_table function updates the instruction table.

- It takes the password used for login, and previous harden password (hdwpd) that was used to encrypt history file for last successful login attempt.
- The create_inst_table function only gets invoked after a successful login attempt.
- First, it calls login function and standard deviation function to compute mean(u) and standard deviation(sigma) that are used in comparison formule, u_i-k*$sigma_i$>t_i and u_i+k*$sigma_i$<t_i. Flag values are updated for all feature values.
- By Comparison, it is decided where to store good and bad share in instruction table.
- The create_inst_table function updates the instruction table.
- It takes the password used for login, and previous harden password (hdwpd) that was used to encrypt history file for last successful login attempt.
- The create_inst_table function only gets invoked after a successful login attempt.
- First, it calls login function and standard deviation function to compute mean(u) and standard deviation(sigma) that are used in comparison formule, ui-k*sigmai>ti and ui+k*sigmai<ti.
- Then, a new harden password is generated which is used to update history file.
- 2m shares are generated, and on the basis of flag values, shares are encrypted by new harden password if we have to save right share on the instruction table, otherwise by any random password to create any vague share.

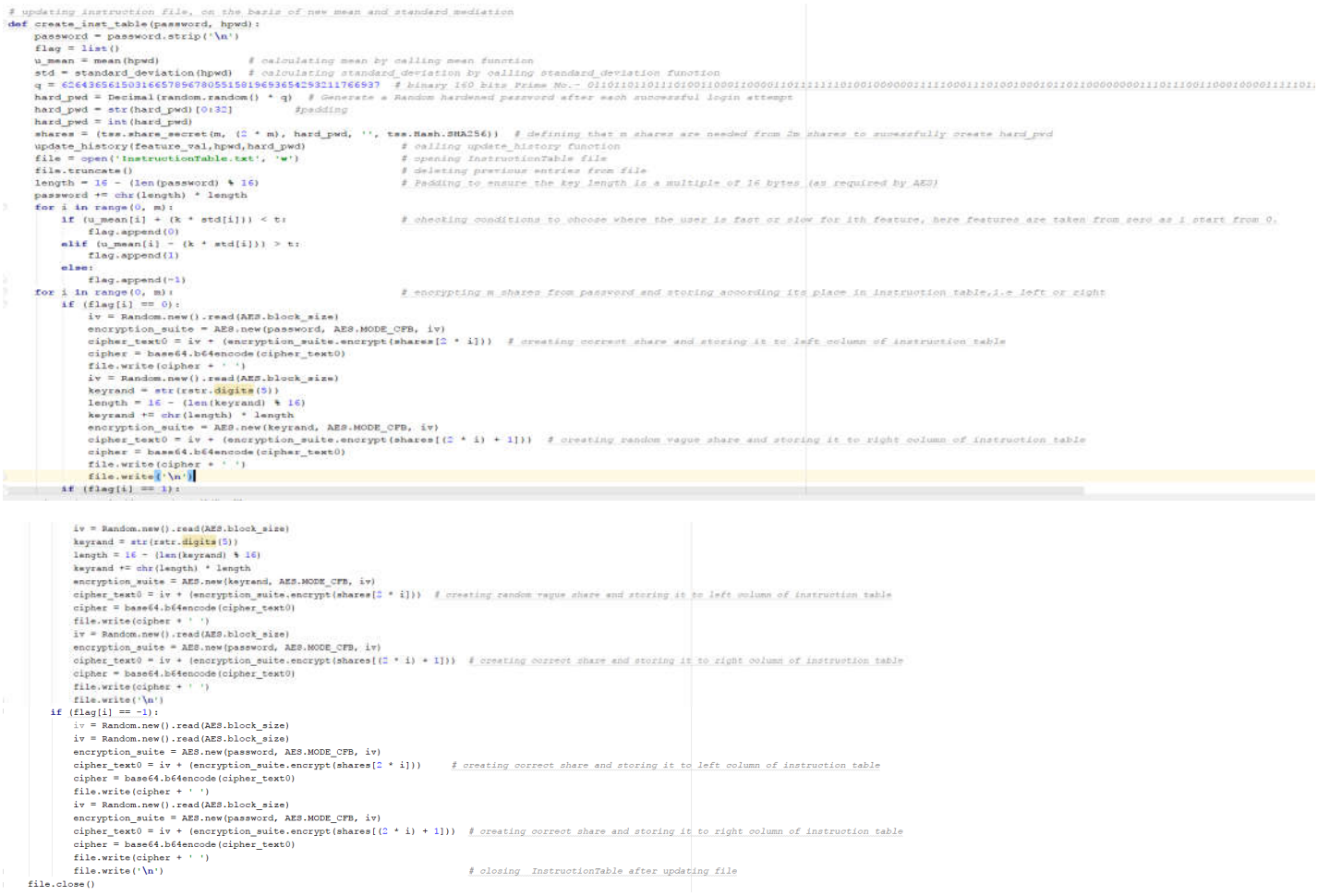

Fig 7: Snippet of the create_inst_table file function

- The decryption_hdpwd function decrypts the content of history file.
- It takes a cipher word to decrypt and harden password from the function which tries to invoke it.
- It then decrypt the cipher and provide the plaintext to the invoking function.

```
# Decrypting cipher text which are provided by mean and standard deviation function for calculating mean and standard deviation
def decryption_hpwd(word, hdpwd):
    hdpwd = str(hdpwd)                #padding
    hdpwd = hdpwd.encode()
    cipher_text0 = base64.b64decode(word)
    iv = cipher_text0[:AES.block_size]
    decryption_suite = AES.new(hdpwd, AES.MODE_CFB, iv)
    plain_text0 = decryption_suite.decrypt(cipher_text0[AES.block_size:])
    return plain_text0
```

Fig 8: Snippet of the decryption_hpwd file function

- The mean function calculates the mean for the feature vectors stored in the history file.
- It takes harden password from the function which tries to invoke it.
- It then opens the history file for reading its contents and simultaneously decrypting them by calling the decryption_hdpwd function.
- It stores all decrypted feature vector in a list type variable and passes it to compute mean by using np.mean function. It returns the mean values to the function which invoked the mean function.

```
# calculating mean
def mean(keys):
    keys = Decimal(keys)
    arr = list()
    mean_list = list()
    counter = 0
    with open('History.txt', 'r') as ins:          # Open History file for reading
        for line in ins:
            line = line.strip()
            words = line.split(' ')
            counter = counter + 1
            if counter == 6:                       # Reading only 5 rows of history file
                break
            for i in range(0, len(words)):
                word = words[i]
                plain = int(decryption_hpwd(word, keys)) # Decrypting history file for  calculating mean
                arr.append(plain)
            mean_list.append(arr)    # storing decrypted features values into list
            arr = list()
    ins.close()                      # Closing History file
    return np.mean(mean_list, axis=0) # returning mean
```

Fig 9: Snippet of the mean file function

- The standard_deviation function calculates the standard deviation for the feature vectors stored in the history file.
- It takes harden password from the function which tries to invoke it.
- It then opens the history file for reading its contents and simultaneously decrypting them by calling the decryption_hdpwd function.
- It stores all decrypted feature vector in a list type variable and passes it to compute standard deviation by using np.std function . It returns the standard deviation values to the function which invoked the standard_deviation function.

```
# calculating standard deviation
def standard_deviation(keys):
    arr = []
    std_list = list()
    counter = 0
    with open('History.txt', 'r') as ins:          # Open History file for reading
        for line in ins:
            line = line.strip()
            words = line.split(' ')
            counter = counter + 1
            if counter == 6:                       # Reading only 5 rows of history file
                break
            for i in range(0, len(words)):
                word = words[i]
                plain = int(decryption_hpwd(word, keys))  # Decrypting history file for  calculating standar deviation
                arr.append(plain)
            std_list.append(arr)                          # storing decrypted features values into list
            arr = list()
    ins.close()                                       # Closing History file
    return np.std(std_list, axis=0)                   # returning mean
```

Fig 10: Snippet of the standard_deviation file function

- The reconstruct_shares function checks whether the login attempt is correct or not by reconstructing the harden password from "m" correct shares, which are chosen from 2m shares by the feature values provided in the login attempt.
- It takes feature values and password from the user, which is trying to log in.
- It then compares the feature value with t(threshold), and on the basis of result picks left or right shares from instruction table and simultaneously decrypts them.
- These decrypted shares are stored in a list type variable, which is given to reconstruct_secret function after all comparisons to get the harden password. If the harden password is successfully generated from these m shares than the login attempt is correct otherwise it is wrong.
- It returns harden password if login attempt comes from legitimate user otherwise return zero.

```
def reconstruct_shares(feature_val, pwd):
    ctr = -1
    shares = list()
    length = 16 - (len(pwd) % 16)  # Key padding
    pwd += chr(length) * length
    flag = 1
    with open('InstructionTable.txt', 'r') as ins: # Opening Instruction Table as read mode
        for line in ins:
            words = line.split(' ')
            ctr = ctr + 1
            if (int(feature_val[ctr]) < t):          # if feature value is less than 't' choose left share
                encrypted_share = words[0]
                cipher_text0 = base64.b64decode(encrypted_share)
                iv = cipher_text0[:AES.block_size]
                decryption_suite = AES.new(pwd, AES.MODE_CFB, iv)
                plain_text0 = decryption_suite.decrypt(cipher_text0[AES.block_size:])
                shares.append(plain_text0)
            else:                                    # if feature value is not less than 't' choose right share
                encrypted_share = words[1]
                cipher_text0 = base64.b64decode(encrypted_share)
                iv = cipher_text0[:AES.block_size]
                decryption_suite = AES.new(pwd, AES.MODE_CFB, iv)
                plain_text0 = decryption_suite.decrypt(cipher_text0[AES.block_size:])
                shares.append(plain_text0)
        try:
            hdpwd = tss.reconstruct_secret(shares)    # try to generate harden password using m shares.
            hdpwd = hdpwd[0:32]
            hdpwd = Decimal(hdpwd)
        except Exception:                             # if m shares are not correct than it will not generate harden password and will through a error, which is covered in exception.
            flag = 0
        if flag == 1:
            return hdpwd                              # if harden password is successfully created by m shares return harden password otherwise 0
        else:
            return 0
    ins.close()                                       # closing instruction table file
```

Fig 11: Snippet of the reconstruct_shares file function

Error Handling in Login.py:

- Login.py successfully deal if there is any error in the path of the input file.
- Login.py also handle errors regarding the length of features, and the number of features (m-1) is valid (where the m= length of correct password) or not.
- It fails login attempt if the password is not correct.
- It checks if feature value is int or can be converted it to int if none happens than it treats this login attempt as failed login and move to next login attempt.