All_problem

October 14, 2025

1 Lab 1 : Caesar Cipher

[29]:
```
def encrypt(pt, shift):
    ct = ""
    for char in pt:
        if char.isalpha():
            offset = 65 if char.isupper() else 97
            en = chr((ord(char)-offset+shift)%26 + offset)
            ct += en
        else:
            ct += char
    return ct
def decrypt(ct, shift):
    return encrypt(ct, -shift)
plaintext="Miju Ahmed"
ct = encrypt(plaintext, 5)
dt = decrypt(ct, 5)
print(f"Plaintext : {plaintext}")
print(f"Cipher text : {ct}")
print(f"Decrypt : {dt}")
```

```
Plaintext : Miju Ahmed
Cipher text : Rnoz Fmrji
Decrypt : Miju Ahmed
```

2 Lab 2: Polygram Substitution

[54]:
```
import itertools
import random

def generate_polygram():
    letters="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    polygrams=[''.join(p) for p in itertools.product(letters,repeat=3)]
    return polygrams
polygram_list=generate_polygram()
with open('lab_2_input.txt','w') as f:
    for p in polygram_list:
```

```
        f.write(p+"\n")
shuffled_polygrams=polygram_list.copy()
random.shuffle(shuffled_polygrams)

polygram_key=dict(zip(polygram_list,shuffled_polygrams))
revese_polygram_key={v:k for k,v in polygram_key.items()}

def polygram_encrypt(plaintext):
    text=plaintext.upper().replace(" ","")
    while len(text)%4!=0:
        text+='$'
    cipher=""
    for i in range(0,len(text),4):
        block=text[i:i+4]
        cipher+=polygram_key.get(block,block)
    return cipher
def polygram_decrypt(ciphertext):
    text=""
    for i in range(0,len(ciphertext),4):
        block=ciphertext[i:i+4]
        text+=revese_polygram_key.get(block,block)
    return text
if __name__=="__main__":
    # plain_text=input("Enter the plaintext: ")
    plain_text = "MynameisMdMijuAhmed"
    cipher_text=polygram_encrypt(plain_text)
    print(f"Cipher text : {cipher_text}")
    with open('lab_2_output.txt' , 'w') as f:
        f.write(cipher_text)
    print("Encryption complete. Ciphertext saved to lab_2_output.txt")
    decrypted_text = polygram_decrypt(cipher_text)
    print("Decrypted back to:", decrypted_text)
```

```
Cipher text : MYNAMEISMDMIJUAHMED$
Encryption complete. Ciphertext saved to lab_2_output.txt
Decrypted back to: MYNAMEISMDMIJUAHMED$
```

3 Lab 3 : Transposition Cipher

[30]:
```
def encrypt(pt, width):
    length = len(pt)
    ct = ""
    for k in range(width):
        for i in range(k,length,width):
            ct += pt[i]
    return ct
def decrypt(ct, width):
```

```
    length = len(ct)
    pt = [' ']*length
    idx = 0
    for k in range(width):
        for i in range(k,length, width):
            pt[i] = ct[idx]
            idx += 1
    return ''.join(pt)
plaintext = "DEPARTMENT OF COMPUTER SCIENCE AND TECHNOLGY UNIVERSITY OF␣
 ↪RAJSHAHI BANGLADESH"
ciphertext = encrypt(plaintext, width=5)
decrypted_text = decrypt(ciphertext, width=5)
print(f"Plaintext:           {plaintext}")
print(f"Ciphertext:          {ciphertext}")
print(f"Decrypted Plaintext: {decrypted_text}")
```

```
Plaintext:           DEPARTMENT OF COMPUTER SCIENCE AND TECHNOLGY UNIVERSITY OF
RAJSHAHI BANGLADESH
Ciphertext:          DT OEI TOUR AHNEEMOMREAELNSOJIGSPEFP NNCGIIFS LHAN USCDHYVT
HBARTCTCE N EYRAAD
Decrypted Plaintext: DEPARTMENT OF COMPUTER SCIENCE AND TECHNOLGY UNIVERSITY OF
RAJSHAHI BANGLADESH
```

4 Lab 4: Double transposition

[31]:
```
def encrypt(pt, width):
    length = len(pt)
    ct = ""
    for k in range(width):
        for i in range(k, length, width):
            ct += pt[i]
    return ct
def decrypt(ct, width):
    length = len(ct)
    pt = [' ']*length
    idx = 0
    for k in range(width):
        for i in range(k, length, width):
            pt[i] = ct[idx]
            idx += 1
    return ''.join(pt)

plaintext = "DEPARTMENT OF COMPUTER SCIENCE AND TECHNOLGY UNIVERSITY OF␣
 ↪RAJSHAHI BANGLADESH"
print("First Transposition:\n")
ciphertext1 = encrypt(plaintext, width=5)
print("Second Transposition:\n")
```

```
ciphertext2=encrypt(ciphertext1, width=5)

decrypted_text2 = decrypt(ciphertext2, width=5)
decrypted_text1=decrypt(decrypted_text2, width=5)
print(f"Plaintext:            {plaintext}")
print(f"Ciphertext1:           {ciphertext1}")
print(f"Ciphertext2:           {ciphertext2}")
print(f"Decrypted Plaintext2: {decrypted_text2}")
print(f"Decrypted Plaintext1: {decrypted_text1}")
```

```
First Transposition:

Second Transposition:

Plaintext:            DEPARTMENT OF COMPUTER SCIENCE AND TECHNOLGY UNIVERSITY OF
RAJSHAHI BANGLADESH
Ciphertext1:           DT OEI TOUR AHNEEMOMREAELNSOJIGSPEFP NNCGIIFS LHAN
USCDHYVT HBARTCTCE N EYRAAD
Ciphertext2:           DIRERNGPG  HHCNAT  EESS ILUYBT A
TAMAOPNIHSVACEDOOHOEJENFACTREYEUNMLIFCSND T R
Decrypted Plaintext2: DT OEI TOUR AHNEEMOMREAELNSOJIGSPEFP NNCGIIFS LHAN
USCDHYVT HBARTCTCE N EYRAAD
Decrypted Plaintext1: DEPARTMENT OF COMPUTER SCIENCE AND TECHNOLGY UNIVERSITY OF
RAJSHAHI BANGLADESH
```

5 Lab 5 : One-Time-Pad

[32]:
```
def encrypt(pt):
    key = ""
    with open("sender.txt","r") as file:
        key = file.read().strip()
    ct = ""
    idx = 0
    for ch in pt:
        x = (ord(ch) + ord(key[idx]))%26
        ct += chr(ord('A') + x + 1)
    remaining_key = key[idx:]
    key_used = key[:idx]
    with open("sender.txt", 'w') as file:
        file.write(remaining_key)
    return ct, key_used, remaining_key
def decrypt(ct):
    key = ""
    with open("received.txt", 'r') as file:
        key = file.read().strip()
    idx = 0
    pt=""
```

```
    for ch in ct:
        x = (ord(ch) - ord(key[idx]))%26
        pt += chr(ord('A') +x-1)
    remaining_key = key[idx:]
    key_used = key[:idx]
    with open("received.txt", 'w') as file:
        file.write(remaining_key)
    return pt, key_used, remaining_key
plaintext = "ONETIMEPAD"
ciphertext, key_used, remaining_key = encrypt(plaintext)
decrypted_text, key_used, remaining_key = decrypt(ciphertext)
print(f"Plaintext:           {plaintext}")
print(f"Ciphertext:          {ciphertext}, Key used: {key_used}, Remaining key:␣
 ↪{remaining_key}")
print(f"Decrypted Plaintext: {decrypted_text}, Key used: {key_used}, Remaining␣
 ↪key: {remaining_key}")
```

```
Plaintext:           ONETIMEPAD
Ciphertext:          IHYNCGYJUX, Key used: , Remaining key: TBFRGFARFMTBFRGFARFM
Decrypted Plaintext: ONETIMEPAD, Key used: , Remaining key: TBFRGFARFMTBFRGFARFM
```

6 Lab 6: Lehhman prime check

[33]:
```
import random
def lehhman_prime_check(p, t):
    if p<2:
        return False
    if p<=3:
        return True
    for _ in range(t):
        a = random.randint(2,p-1)
        k = pow(a,(p-1)//2, p)
        if k!=1 and k!=p-1:
            return False
    return True
n,t=503,10
check = lehhman_prime_check(n,t)
if check:
    print(f"{n} is probably prime")
else:
    print(f"{n} is composite")
```

```
503 is probably prime
```

7 Lab 7 : Robin Miller

[34]:
```
import random
def robin_miller_prime_check(p,k):
    if p<2:
        return False
    if p<=3:
        return True
    if p%2==0:
        return False
    m = p-1
    b=0
    while m%2==0:
        m//=2
        b += 1
    for _ in range(t):
        a = random.randint(2,p-1)
        z = pow(a,m,p)
        if z==1 or z==p-1:
            continue
        for _ in range(b-1):
            z = pow(z,2,p)
            if z==p-1:
                break
            else:
                return False
    return True

p=155589
k=10

if robin_miller_prime_check(p,k):
    print(f"{p} probably prime")
else:
    print(f"{p} is not prime")
```

```
155589 is not prime
```

8 Lab 8: MD5

[35]:
```
import hashlib
def generate_md5_hash(pt):
    md_hash = hashlib.md5()
    md_hash.update(pt.encode('utf-8'))
    return md_hash.hexdigest()
pt = "Miju Chowdhury"
hash_value = generate_md5_hash(pt)
```

```
print(f"Plain text : {pt}")
print(f"Hash Value : {hash_value}")
```

```
Plain text : Miju Chowdhury
Hash Value : ff0994529f1e3ec91e689539c85131d4
```

9 Lab 9: SHA

[36]:
```
import hashlib
def hash_message(message, algorithm='sha256'):
    message_bytes = message.encode('utf-8')
    if algorithm == 'sha1':
        hash_obj = hashlib.sha1(message_bytes)
    elif algorithm == 'sha224':
        hash_obj = hashlib.sha224(message_bytes)
    elif algorithm == 'sha256':
        hash_obj = hashlib.sha256(message_bytes)
    elif algorithm == 'sha384':
        hash_obj = hashlib.sha384(message_bytes)
    elif algorithm == 'sha512':
        hash_obj = hashlib.sha512(message_bytes)
    else:
        print("Invalid algorithm")
    return hash_obj.hexdigest()

message = "Miju Chowdhury"
algorithm = "sha1"
hashed_output = hash_message(message, algorithm)
print(f"Plaintext: {message}")
print(f"Algorithm: {algorithm}")
print(f"Hashed Output using {algorithm.upper()}: {hashed_output}")
```

```
Plaintext: Miju Chowdhury
Algorithm: sha1
Hashed Output using SHA1: dcf1546195827f1cfbac6fe419e92be497df9cb9
```

10 Lab 10: RSA

[37]:
```
e=79
d=1019
M=6880023
n=3337

M_str = str(M)
print(M_str)

msg_block = []
```

```
for i in range(0,len(M_str), 3):
    block = M_str[i:i+3]
    msg_block.append(int(block))
print(msg_block)

cipher_block = []
for m in msg_block:
    c = pow(m,e,n)
    cipher_block.append(c)
print(cipher_block)

cipher_text=""
for c in cipher_block:
    cstr = str(c).zfill(4)
    cipher_text+=cstr
print(cipher_text)

cipher_block = []
for i in range(0,len(cipher_text),4):
    mstr = cipher_text[i:i+4]
    cipher_block.append(int(mstr))
print(cipher_block)

decrypted_text = ""
for i,c in enumerate(cipher_block):
    m = pow(c,d,n)
    if i<len(cipher_block)-1:
        mstr = str(m).zfill(3)
    else:
        remaining_length = len(M_str)-len(decrypted_text)
        mstr = str(m).zfill(remaining_length)
    decrypted_text += mstr
decrypted_text = decrypted_text[-len(M_str):]
print(decrypted_text)
```

```
6880023
[688, 2, 3]
[1570, 3139, 158]
157031390158
[1570, 3139, 158]
6880023
```

11 Lab 11: Diffie-Hellman key generator

[38]:
```
import random
def diffie_hellman_key(p):
    a = 7
    xa = random.randint(2,p-1)
    ya = pow(a,xa,p)

    xb = random.randint(2,p-1)
    yb = pow(a,xb,p)

    ka = pow(yb,xa,p)
    kb = pow(ya,xb,p)
    return ka, kb
p = 152
ka,kb = diffie_hellman_key(p)
print(f"A : {ka}\nB : {kb}")
```

```
A : 1
B : 1
```

12 Lab 12 : PGP

[55]:
```
from Crypto.PublicKey import RSA
from Crypto.Signature import pkcs1_15
from Crypto.Hash import SHA1
from Crypto.Cipher import AES, PKCS1_OAEP
from Crypto.Util.Padding import pad, unpad
from Crypto.Random import get_random_bytes

class SimplePGP:
    def __init__(self):
        self.sender_key = RSA.generate(1024)
        self.receiver_key = RSA.generate(1024)
        print("  Simple PGP System initialized")

    # ---------------- Authentication ----------------
    def authenticate(self, message):
        h = SHA1.new(message.encode())
        signature = pkcs1_15.new(self.sender_key).sign(h)
        return {'message': message, 'signature': signature}

    def verify(self, signed_msg):
        h = SHA1.new(signed_msg['message'].encode())
        try:
            pkcs1_15.new(self.sender_key.publickey()).verify(h,␣
 ↪signed_msg['signature'])
```

```
            return True
        except (ValueError, TypeError):
            return False

    # ---------------- Confidentiality ----------------
    def encrypt_message(self, message):
        session_key = get_random_bytes(16)
        cipher_aes = AES.new(session_key, AES.MODE_CBC)
        ct = cipher_aes.encrypt(pad(message.encode(), AES.block_size))
        cipher_rsa = PKCS1_OAEP.new(self.receiver_key.publickey())
        enc_key = cipher_rsa.encrypt(session_key)
        return {'enc_key': enc_key, 'iv': cipher_aes.iv, 'ct': ct}

    def decrypt_message(self, package):
        cipher_rsa = PKCS1_OAEP.new(self.receiver_key)
        session_key = cipher_rsa.decrypt(package['enc_key'])
        cipher_aes = AES.new(session_key, AES.MODE_CBC, package['iv'])
        pt = unpad(cipher_aes.decrypt(package['ct']), AES.block_size)
        return pt.decode()

    # ---------------- Complete PGP ----------------
    def pgp_send(self, message):
        signed = self.authenticate(message)
        full_message = f"{message}|SIGNED"
        encrypted = self.encrypt_message(full_message)
        return encrypted, signed['signature']

    def pgp_receive(self, encrypted, signature):
        decrypted = self.decrypt_message(encrypted)
        message = decrypted.split('|SIGNED')[0]
        return message, self.verify({'message': message, 'signature':␣
↪signature})


# ---------------- TESTING ----------------
pgp = SimplePGP()
msg = "This is a confidential message for PGP testing"

print("\n Testing Authentication")
signed = pgp.authenticate(msg)
print("Verified:", pgp.verify(signed))

print("\n Testing Confidentiality")
enc = pgp.encrypt_message(msg)
dec = pgp.decrypt_message(enc)
print("Decrypted:", dec)
```

```
print("\n  Testing Complete PGP")
enc_pkg, sig = pgp.pgp_send(msg)
final_msg, auth_ok = pgp.pgp_receive(enc_pkg, sig)
print("Decrypted:", final_msg)
print("Authentication Verified:", auth_ok)
```

```
 Simple PGP System initialized

 Testing Authentication
Verified: True

 Testing Confidentiality
Decrypted: This is a confidential message for PGP testing

 Testing Complete PGP
Decrypted: This is a confidential message for PGP testing
Authentication Verified: True
```

13 Sajjad

[39]:
```
import hashlib
import os
from Crypto.PublicKey import RSA
from Crypto.Signature import pkcs1_15
from Crypto.Hash import SHA1
from Crypto.Cipher import AES, PKCS1_OAEP
from Crypto.Random import get_random_bytes
from Crypto.Util.Padding import pad, unpad
```

[44]:
```
class PGPSystem:
    def __init__(self):
        # Generate RSA key pairs using built-in functions
        self.sender_key = RSA.generate(1024)
        self.receiver_key = RSA.generate(1024)

        print("PGP System Initialized with Built-in RSA")
        print(f"Sender Key Size: {self.sender_key.size_in_bits()} bits")
        print(f"Receiver Key Size: {self.receiver_key.size_in_bits()} bits")

    def pgp_authentication(self, message):
        print("\n=== PGP AUTHENTICATION SERVICE ===")
        print(f"1. Sender creates message: '{message}'")

        # Step 2: Generate SHA-1 hash (160-bit)
        message_bytes = message.encode('utf-8')
        hash_obj = SHA1.new(message_bytes)
```

```
        print(f"2. SHA-1 generates 160-bit hash: {hash_obj.hexdigest()[:32]}...
↪")

        # Step 3: Encrypt hash with sender's private key (Digital Signature)
        signature = pkcs1_15.new(self.sender_key).sign(hash_obj)
        print(f"3. Hash encrypted with sender's private key (signature␣
↪created)")

        # Prepend signature to message
        signed_message = {
            'message': message,
            'signature': signature,
            'hash': hash_obj.digest()
        }
        print(f"4. Signature prepended to message")

        return signed_message

    def pgp_authentication_verify(self, signed_message):
        print("\n=== PGP AUTHENTICATION VERIFICATION ===")

        message = signed_message['message']
        signature = signed_message['signature']

        print(f"1. Receiver got message: '{message}'")

        # Step 4: Decrypt signature using sender's public key
        try:
            # Generate new hash of received message
            new_hash = SHA1.new(message.encode('utf-8'))
            print(f"2. Generated new hash: {new_hash.hexdigest()[:32]}...")

            # Verify signature using sender's public key
            pkcs1_15.new(self.sender_key.publickey()).verify(new_hash,␣
↪signature)
            print(f"3. Signature decrypted with sender's public key")
            print(f"4. Hash comparison: MATCH - Message is AUTHENTIC")
            return True

        except (ValueError, TypeError):
            print(f"4. Hash comparison: MISMATCH - Message is NOT AUTHENTIC")
            return False

    def pgp_confidentiality(self, message):
        print("\n=== PGP CONFIDENTIALITY SERVICE ===")
        print(f"1. Sender generates message: '{message}'")
```

```
        # Step 2: Generate random 128-bit session key
        session_key = get_random_bytes(16)  # 128 bits = 16 bytes
        print(f"2. Random 128-bit session key generated")

        # Step 3: Encrypt message using AES with session key
        cipher_aes = AES.new(session_key, AES.MODE_CBC)
        iv = cipher_aes.iv
        padded_message = pad(message.encode('utf-8'), AES.block_size)
        encrypted_message = cipher_aes.encrypt(padded_message)
        print(f"3. Message encrypted using AES with session key")

        # Step 4: Encrypt session key with RSA using recipient's public key
        cipher_rsa = PKCS1_OAEP.new(self.receiver_key.publickey())
        encrypted_session_key = cipher_rsa.encrypt(session_key)
        print(f"4. Session key encrypted with RSA using recipient's public key")
        print(f"5. Encrypted session key prepended to message")

        # Create encrypted package
        encrypted_package = {
            'encrypted_session_key': encrypted_session_key,
            'iv': iv,
            'encrypted_message': encrypted_message
        }

        return encrypted_package


    def pgp_confidentiality_decrypt(self, encrypted_package):
        print("\n=== PGP CONFIDENTIALITY DECRYPTION ===")

        # Step 5: Decrypt session key using receiver's private key
        cipher_rsa = PKCS1_OAEP.new(self.receiver_key)
        session_key = cipher_rsa.
↪decrypt(encrypted_package['encrypted_session_key'])
        print(f"1. Session key decrypted using receiver's private key")

        # Step 6: Decrypt message using session key
        cipher_aes = AES.new(session_key, AES.MODE_CBC, encrypted_package['iv'])
        decrypted_padded = cipher_aes.
↪decrypt(encrypted_package['encrypted_message'])
        decrypted_message = unpad(decrypted_padded, AES.block_size).
↪decode('utf-8')
        print(f"2. Message decrypted using session key")
        print(f"3. Decrypted message: '{decrypted_message}'")

        return decrypted_message
```

```
    def pgp_complete_service(self, message):
        print("\n" + "="*60)
        print("COMPLETE PGP SERVICE - AUTHENTICATION + CONFIDENTIALITY")
        print("="*60)

        # First apply authentication (digital signature)
        signed_message = self.pgp_authentication(message)

        # Then apply confidentiality to the signed message
        message_with_signature =␣
↪f"{signed_message['message']}|SIG|{len(signed_message['signature'])}"
        encrypted_package = self.pgp_confidentiality(message_with_signature)

        # Combine both
        complete_package = {
            'encrypted_package': encrypted_package,
            'signature': signed_message['signature']
        }

        return complete_package, signed_message


    def pgp_complete_decrypt(self, complete_package, original_signed):
        print("\n" + "="*60)
        print("COMPLETE PGP DECRYPTION + VERIFICATION")
        print("="*60)

        # First decrypt the message
        decrypted_message = self.
↪pgp_confidentiality_decrypt(complete_package['encrypted_package'])

        # Extract original message
        parts = decrypted_message.split('|SIG|')
        original_message = parts[0]

        # Verify authentication using the signature
        signed_msg = {'message': original_message, 'signature':␣
↪complete_package['signature']}
        is_authentic = self.pgp_authentication_verify(signed_msg)

        return original_message, is_authentic

# Initialize PGP System
pgp = PGPSystem()
```

```
PGP System Initialized with Built-in RSA
Sender Key Size: 1024 bits
```

```
Receiver Key Size: 1024 bits
```

[45]:
```
message = "This is a confidential message for pgp testing"
print("Testing pgp service with rsa and aes")

# test 1
print("\nTest autehtication service")
signed_msg = pgp.pgp_authentication(message)
auth_result = pgp.pgp_authentication_verify(signed_msg)
```

```
Testing pgp service with rsa and aes

Test autehtication service

=== PGP AUTHENTICATION SERVICE ===
1. Sender creates message: 'This is a confidential message for pgp testing'
2. SHA-1 generates 160-bit hash: bbdd96e5826ee325109332fe1e2f8407…
3. Hash encrypted with sender's private key (signature created)
4. Signature prepended to message

=== PGP AUTHENTICATION VERIFICATION ===
1. Receiver got message: 'This is a confidential message for pgp testing'
2. Generated new hash: bbdd96e5826ee325109332fe1e2f8407…
3. Signature decrypted with sender's public key
4. Hash comparison: MATCH - Message is AUTHENTIC
```

[46]:
```
# test 2
print("testing confidentiality service")
encrypted_pkg = pgp.pgp_confidentiality(message)
decrypted_msg = pgp.pgp_confidentiality_decrypt(encrypted_pkg)
```

```
testing confidentiality service

=== PGP CONFIDENTIALITY SERVICE ===
1. Sender generates message: 'This is a confidential message for pgp testing'
2. Random 128-bit session key generated
3. Message encrypted using AES with session key
4. Session key encrypted with RSA using recipient's public key
5. Encrypted session key prepended to message

=== PGP CONFIDENTIALITY DECRYPTION ===
1. Session key decrypted using receiver's private key
2. Message decrypted using session key
3. Decrypted message: 'This is a confidential message for pgp testing'
```

[47]:
```
# Test 3: Complete PGP Service (Authentication + Confidentiality)
print("\n>>> TESTING COMPLETE PGP SERVICE <<<")
complete_pkg, signed_original = pgp.pgp_complete_service(message)
```

```
final_msg, final_auth = pgp.pgp_complete_decrypt(complete_pkg, signed_original)
```

```
>>> TESTING COMPLETE PGP SERVICE <<<

============================================================
COMPLETE PGP SERVICE - AUTHENTICATION + CONFIDENTIALITY
============================================================

=== PGP AUTHENTICATION SERVICE ===
1. Sender creates message: 'This is a confidential message for pgp testing'
2. SHA-1 generates 160-bit hash: bbdd96e5826ee325109332fe1e2f8407…
3. Hash encrypted with sender's private key (signature created)
4. Signature prepended to message

=== PGP CONFIDENTIALITY SERVICE ===
1. Sender generates message: 'This is a confidential message for pgp
testing|SIG|128'
2. Random 128-bit session key generated
3. Message encrypted using AES with session key
4. Session key encrypted with RSA using recipient's public key
5. Encrypted session key prepended to message

============================================================
COMPLETE PGP DECRYPTION + VERIFICATION
============================================================

=== PGP CONFIDENTIALITY DECRYPTION ===
1. Session key decrypted using receiver's private key
2. Message decrypted using session key
3. Decrypted message: 'This is a confidential message for pgp testing|SIG|128'

=== PGP AUTHENTICATION VERIFICATION ===
1. Receiver got message: 'This is a confidential message for pgp testing'
2. Generated new hash: bbdd96e5826ee325109332fe1e2f8407…
3. Signature decrypted with sender's public key
4. Hash comparison: MATCH - Message is AUTHENTIC
```

[48]:
```
print("\n" + "="*70)
print("FINAL RESULTS")
print("="*70)
print(f"Original Message: '{message}'")
print(f"Authentication Test: {'PASSED' if auth_result else 'FAILED'}")
print(f"Confidentiality Test: {'PASSED' if decrypted_msg == message else␣
↪'FAILED'}")
print(f"Complete Service: {'PASSED' if final_msg == message and final_auth else␣
↪'FAILED'}")
```

```
print(f"Final Decrypted: '{final_msg}'")
print(f"Final Authentication: {'VERIFIED' if final_auth else 'FAILED'}")
```

```
======================================================================
FINAL RESULTS
======================================================================
Original Message: 'This is a confidential message for pgp testing'
Authentication Test: PASSED
Confidentiality Test: PASSED
Complete Service: PASSED
Final Decrypted: 'This is a confidential message for pgp testing'
Final Authentication: VERIFIED
```