Cryptography and Network Security
Lab Assignment - V

Program 1: ECC Encryption over GF(P)

Code:

```python
def add(P, Q, a, p):
    if P is None:
        return Q
    if Q is None:
        return P

    x1,y1=P
    x2,y2=Q
    if x1==x2 and y1==p-y2:
        return None
    if x1==x2 and y1==y2:
        if y1==0:
            return None
        l=((3*x1*x1+a)*pow(2*y1,p-2,p))%p
    else:
        l=((y2-y1)*pow(x2-x1,p-2,p))%p
    x3=(l*l-x1-x2)%p
    y3= (l*(x1-x3)-y1)%p

    return (x3, y3)

def mul(P, k, a, p):
    R=None
    Q=P
    while k:
        if k&1:
            R=add(R, Q, a, p)
        Q=add(Q, Q, a, p)
        k>>=1

    return R

def ecc_encrypt(P, M, a, b, G, nB, k):
    public_key=mul(G, nB, a, P)
    shared_secret=mul(public_key, k, a, P)

    C1=mul(G, k, a, P)
    C2=add(M, shared_secret, a, P)

    return public_key, (C1, C2)

print("Siddhanth Monnappa\t22BCE3061\n")
P=int(input("Enter P (Prime Modulus): "))
M=list(map(int, input("Enter M (Message Point) Values: ").split()))
a=int(input("Enter a (Curve Parameter a) value: "))
b=int(input("Enter b (Curve Parameter b) value: "))
G=list(map(int, input("Enter G (Base Point) values: ").split()))
nB=int(input("Enter nB (Private key) value: "))
k=int(input("Enter k (Randomisation) value: "))

public_key,cipher_text=ecc_encrypt(P, M, a, b, G, nB, k)

print("\n-ECC Encryption Results-\n")
print(f"Public Key: {public_key}")
print(f"Cipher Text: {cipher_text}")
```

Input/Output:

```
Siddhanth Monnappa      22BCE3061

Enter P (Prime Modulus): 67
Enter M (Message Point) Values: 24 26
Enter a (Curve Parameter a) value: 2
Enter b (Curve Parameter b) value: 3
Enter G (Base Point) values: 2 22
Enter nB (Private key) value: 4
Enter k (Randomisation) value: 2

-ECC Encryption Results-

Public Key: (13, 45)
Cipher Text: ((35, 1), (21, 44))
```

```
Siddhanth Monnappa      22BCE3061

Enter P (Prime Modulus): 23
Enter M (Message Point) Values: 9 7
Enter a (Curve Parameter a) value: 1
Enter b (Curve Parameter b) value: 1
Enter G (Base Point) values: 3 10
Enter nB (Private key) value: 2
Enter k (Randomisation) value: 2

-ECC Encryption Results-

Public Key: (7, 12)
Cipher Text: ((7, 12), (3, 13))
```

Program 2: MD5 Round Function (F, G, H, I)

Code:

```python
print("Siddhanth Monnappa\t22BCE3061\n")

A=int(input("Enter A 8-Bit Hex Value: "),16)
B=int(input("Enter B 8-Bit Hex Value: "),16)
C=int(input("Enter C 8-Bit Hex Value: "),16)
D=int(input("Enter D 8-Bit Hex Value: "),16)

print("\n-MD5 Round Function Values-\n")
F=(B & C) | (~B & D) & 0xFFFFFFFF
G=(B & D) | (C & ~D) & 0xFFFFFFFF
H=B ^ C ^ D & 0xFFFFFFFF
I=C ^ (B | ~D) & 0xFFFFFFFF

print(f"F Function Result: {F:08X}")
print(f"G Function Result: {G:08X}")
print(f"H Function Result: {H:08X}")
print(f"I Function Result: {I:08X}")
```

Input/Output:

```
Siddhanth Monnappa      22BCE3061

Enter A 8-Bit Hex Value: 01234567
Enter B 8-Bit Hex Value: 89abcdef
Enter C 8-Bit Hex Value: fedcba98
Enter D 8-Bit Hex Value: 76543210

-MD5 Round Function Values-

F Function Result: FEDCBA98
G Function Result: 88888888
H Function Result: 01234567
I Function Result: 77777777
```

Program 3: DSS Implementation

Code:

```python
def dss(p, q, k, h, priv_key, hm, g):
    pub_key = pow(g, priv_key, p)
    r = pow(g, k, p)%q
    try:
        k_inv=pow(k, -1, q)
    except:
        print("Inverse does not exist")
        exit(0)
    s = (k_inv*(hm+priv_key*r)) % q

    try:
        w=pow(s, -1, q)
    except:
        print("Inverse does not exist")
        exit(0)

    u1 = (hm*w)%q
    u2 = (r*w)%q

    v = (pow(g, u1, p)*pow(pub_key, u2, p))%p%q

    if v==r:
        verify="Accepted"
    else:
        verify="Rejected"

    print("\n-DSS Implementation-\n")
    print(f"Public Key: {pub_key}")
    print(f"Signature: (r, s) = ({r}, {s})")
    print(f"Verification Values:   (u1, u2) = ({u1}, {u2})")
    print(f"Verification Status:   {verify}")

print("Siddhanth Monnappa\t22BCE3061\n")

p=int(input("Enter p (Prime Modulus): "))
q=int(input("Enter q (Prime Divisor): "))
k=int(input("Enter k (Randomisation): "))
h=int(input("Enter h (Hash function): "))
priv_key=int(input("Enter Private Key: "))
hm=int(input("Enter Message Hash: "))
g=int(input("Enter g (Base Point): "))

dss(p,q,k,h,priv_key,hm,g)
```

Input/Output:

```
Siddhanth Monnappa      22BCE3061

Enter p (Prime Modulus): 23
Enter q (Prime Divisor): 11
Enter k (Randomisation): 5
Enter h (Hash function): 7
Enter Private Key: 3
Enter Message Hash: 4
Enter g (Base Point): 3

-DSS Implementation-

Public Key: 4
Signature: (r, s) = (2, 2)
Verification Values:   (u1, u2) = (2, 1)
Verification Status:   Accepted
```

Program 4: SHA-512 Input Sequence Generation

Code:

```python
def rotr(x, n):
    return (x >> n) | (x << (64 - n)) & 0xFFFFFFFFFFFFFFFF

def shr(x, n):
    return x >> n

def sigma0(x):
    return rotr(x, 1) ^ rotr(x, 8) ^ shr(x, 7)

def sigma1(x):
    return rotr(x, 19) ^ rotr(x, 61) ^ shr(x, 6)

def sha_512_inp_seq(M):
    M_bytes=M.encode('utf-8')
    M_bits=''.join(format(byte, '08b') for byte in M_bytes)
    M_len=len(M_bits)

    M_bits+='1'
    while len(M_bits)%1024!=896:
        M_bits+='0'

    M_bits+=format(M_len, '0128b')
    print(f"Hex Message of 1024 bits: {hex(int(M_bits, 2))}")

    W=[int(M_bits[i:i+64], 2) for i in range(0, 1024, 64)]

    for t in range(16, 80):
        Wt=(sigma1(W[t-2])+W[t-7]+sigma0(W[t-15])+W[t-16]) & 0xFFFFFFFFFFFFFFFF
        W.append(Wt)

    return W

print("Siddhanth Monnappa\t22BCE3061\n")
M=input("Enter Message: ")
W=sha_512_inp_seq(M)

print("SHA-512 Input Sequence:")
for t, Wt in enumerate(W):
    print(f"W{t} = {hex(Wt)}")

```

Input/Output:

Input – Message String (Input String from DSS Implementation test case values)

Output – Word Sequence Blocks from 0 to 79

```
Siddhanth Monnappa      22BCE3061

Enter Message: p=23,q=11,k=5,h=7,priv_key=3,hM=4,g=3
Hex Message of 1024 bits: 0x703d32332c713d31312c6b3d352c683d372c707269765f6b65793d332c684d3d342c673d338000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000128
SHA-512 Input Sequence:
W0 = 0x703d32332c713d31
W1 = 0x312c6b3d352c683d
W2 = 0x372c707269765f6b
W3 = 0x65793d332c684d3d
W4 = 0x342c673d33800000
W5 = 0x0
W6 = 0x0
W7 = 0x0
W8 = 0x0
W9 = 0x0
W10 = 0x0
W11 = 0x0
W12 = 0x0
W13 = 0x0
W14 = 0x0
W15 = 0x128
W16 = 0x160273570a3a7dd7
W17 = 0x2220b7e6d72d26d5
W18 = 0xc630f8067048e3bc
W19 = 0x351b1e183fc62614
W20 = 0x63150252b806f061
W21 = 0x6cce1a1afd0d2fc1
W22 = 0xc7286afe2a9dd0f2
W23 = 0xd83e387d4b1892ec
W24 = 0xa26266a678485c5b
W25 = 0x96fd9812db30400a
W26 = 0x4f2ce6fd573d32b9
W27 = 0x20cb667bd9f38a93
W28 = 0x4adac7afcd77fba8
W29 = 0x3eb2e55c76d2b27f
W30 = 0xa8c697b8d7988e85
W31 = 0xa1bfac85903125da
W32 = 0x46bd5815b3c7aec6
W33 = 0x7b609a98c3785acc
W34 = 0xb6f7fdcae5d0a6a0
W35 = 0x21d67db824a5c443
W36 = 0xbb4a6e517b064e95
W37 = 0x5d2ecfa0f5e2bdea
W38 = 0xfbcc6255ad349a84
W39 = 0x6a92b428cac3b394
W40 = 0xad65883e4a10dbd7
W41 = 0x8f8a02f8111c4fea
W42 = 0xe6eb75caf5201d62
```

```
W44 = 0xbf909c308fba1e41
W45 = 0x6d4f64381c1a31c9
W46 = 0x5c48f69f956d0f1f
W47 = 0x62b1d2a0ada08650
W48 = 0xa5ac4a159cbe2a4
W49 = 0x61bb5560bc454017
W50 = 0x1a88611f689b0fe8
W51 = 0xcf2eadea0ce51250
W52 = 0xa2de11386869f7fd
W53 = 0x8a65f7cece632549
W54 = 0x2af9689ad7dcd331
W55 = 0xaae1fc37833f71d4
W56 = 0x887d0145ffbdcf51
W57 = 0x760fdea397b8516d
W58 = 0x264c3cddee102671
W59 = 0x5ecdc542dbdf1d20
W60 = 0xff3dea8524e94912
W61 = 0x5eba8601cb4003a5
W62 = 0x3c5c2a635e5c6cc3
W63 = 0x813cd2c50a5b2762
W64 = 0x97724ecdec41203d
W65 = 0xdc816fb01cf2882e
W66 = 0x4de4591f99438c0a
W67 = 0xb271a6ccb86c2610
W68 = 0xaed8fd5da767d0b7
W69 = 0x804cc8d7b6aeb609
W70 = 0xbb309442cd5ba21e
W71 = 0xada0983e1414b04b
W72 = 0xeac82749bf755f9a
W73 = 0x9ffc38130e2812e7
W74 = 0xe65ce0232abe5a7f
W75 = 0x7a0939a55adae3db
W76 = 0x847ace3a08f210b2
W77 = 0x530922107f156a6
W78 = 0x70ebb91f3318a8ae
W79 = 0x664a3fab13b62b0e
```