# Exercício 3 - Trabalho Prático 2

**Grupo 6:**

Ruben Silva - pg57900

Luís Costa - pg55970

# Problema:

1. Usando a experiência obtida na resolução dos problemas 1 e 2, e usando, ao invés do grupo abeliano multiplicativo $\mathbb{F}_p^*$, o grupo abeliano aditivo que usou na pergunta 2,
   A. Construa ambas as versões IND-CPA segura e IND-CCA segura do esquema de cifra ElGamal em curvas elípticas.
   B. Construa uma implementação em curvas elípticas de um protocolo autenticado de "Oblivious Transfer" $\kappa$-out-of-$n$.

## Parte I - IND-CPA

Paper ElGamal Elliptic Curve

In [112…
```
from sage.all import *
from cryptography.hazmat.primitives import hashes
import secrets
import hashlib
import base64
import random
```

## Class Edwards22519

1. É a class do exercício 2 sem a parte da "criptografia". Contendo primordialmente as funções das curvas elipticas

In [113…
```
class Edwards25519():
    #Private Functions
    def __init__(self):
        self.p = 2**255 - 19
        self.d = -121665 * inverse_mod(121666, self.p) % self.p
        self.q = 2**252 + 27742317777372353535851937790883648493


        self.gy = 4 * self._modp_inv(5) % self.p
        self.gx = self._recover_x(self.gy, 0)
        self.G = (self.gx, self.gy, 1, self.gx * self.gy % self.p)

    def _sha512(self, data):
```

```
        return hashlib.sha512(data).digest()

    def _sha512_modq(self, data):
        return int.from_bytes(self._sha512(data), 'little') % self.q

    def _point_add(self, P, Q):
        A, B = (P[1]-P[0]) * (Q[1]-Q[0]) % self.p, (P[1]+P[0]) * (Q[1]+Q[0]) % s
        C, D = 2 * P[3] * Q[3] * self.d % self.p, 2 * P[2] * Q[2] % self.p
        E, F, G, H = B-A, D-C, D+C, B+A
        return (E*F, G*H, F*G, E*H)

    def _point_sub(self, P, Q):
        A, B = (P[1]-P[0]) * (Q[1]+Q[0]) % self.p, (P[1]+P[0]) * (Q[1]-Q[0]) % s
        C, D = 2 * P[3] * Q[2] * self.d % self.p, 2 * P[2] * Q[3] % self.p
        E, F, G, H = B-A, D-C, D+C, B+A
        return (E*F, G*H, F*G, E*H)

    def _point_mult(self,s,P):
        Q = (0, 1, 1, 0)
        while s > 0:
            if s & 1:
                Q = self._point_add(Q, P)
            P = self._point_add(P, P)
            s >>= 1
        return Q

    def _point_equal(self,P,Q):
        if (P[0] * Q[2] - Q[0] * P[2]) % self.p != 0:
            return False
        if (P[1] * Q[2] - Q[1] * P[2]) % self.p != 0:
            return False
        return True


    def _mod_sqrt(self):
        return pow(2, (self.p-1)//4, self.p)

    def _modp_inv(self, x):
        return pow(x, self.p-2, self.p)

    def _recover_x(self, y, sign):
        if y >= self.p:
            return None
        x2 = (y*y-1) * self._modp_inv(self.d*y*y+1)
        if x2 == 0:
            if sign:
                return None
            else:
                return 0
        x = pow(x2, (self.p+3) // 8, self.p)
        if (x*x - x2) % self.p != 0:
            x = x * self._mod_sqrt() % self.p
        if (x*x - x2) % self.p != 0:
            return None
        if (x & 1) != sign:
            x = self.p - x
        return x

    def _point_compress(self, P):
        zinv = self._modp_inv(P[2])
```

```
        x = P[0] * zinv % self.p
        y = P[1] * zinv % self.p
        return int(y | ((x & 1) << 255)).to_bytes(32, 'little')

    def _point_decompress(self, s):
        if len(s) != 32:
            raise Exception("Invalid input length for decompression")
        y = int.from_bytes(s, "little")
        sign = y >> 255
        y &= (1 << 255) - 1
        x = self._recover_x(y, sign)

        if x is None:
            return None
        else:
            return (x, y, 1, x*y % self.p)


    def _secret_expand(self, secret):
        if len(secret) != 32:
            raise Exception("Invalid input length for secret key")
        h = self._sha512(secret)
        a = int.from_bytes(h[:32], "little")
        a &= (1 << 254) - 8
        a |= (1 << 254)
        return (a, h[32:])
```

# ElGamal em Curva Eliptica IND-CPA

1. É herdado as propriedades da Curva `Edwards25519` com o *init*
2. É definido a função **ElGamal_GenKeys** que retorna a `public_key` e a `private_key`
    A. É gerado a `private_key` tal que $0 <$ private_key $< q$ reduzido ao módulo de $q$
    B. É gerado a `public_key` tal que $s.G$
        a. $s$->private_key
        b. $G$->ponto base da curva
    C. a public_key é comprimida e é devolvido o par "sk,pk" como esperado de um ElGamal
3. É definido a função **ElGamal_Enc** que retorna o `ciphertext` $(R, C)$
    A. A public_key é descomprimida para obter $P = s.G$
    B. É escolhido um valor descartável aleatório entre $0$ e $q$
    C. É Calculado $R = k.G$ e $kP = k.P$
        a. $R = k.G$ equivale $g^k$ em ElGamal do exercício 1
        b. $kP = k.P$ equivale $(g^s)^k$ em ElGamal do exercício 1
    D. É cifrada a mensagem apartir da soma dos pontos mantendo assim a estrutura de um `grupo abeliano aditivo` resultando no R
    E. Retorna $R$ e $C$ comprimidos em tuplo

4. É definido a função **ElGamal_Dec** que retorna o `ponto na curva decifrado`

   A. A $R$ e $C$ são descomprimidos
   B. É calculado o ponto $S = s.R = s.(k.G)$ usando a private_key $s$
   C. É invertido o eixo $x$ de S para facilitar na "adição" (obtermos assim a subtração para recuperarmos a mensagem)
   D. Devolve o ponto decifrado

In [114…

```python
class EC_ElGamal_CPA(Edwards25519):
    def __init__(self):
        super().__init__()

    def ElGamal_GenKeys(self):
        private_key = int.from_bytes(os.urandom(32), "little") % self.q
        public_key = self._point_mult(private_key, self.G)
        return private_key, self._point_compress(public_key)

    def ElGamal_Enc(self, public_key, message_point):
        pub_point = self._point_decompress(public_key)
        if pub_point is None:
            raise ValueError("Invalid public key")

        k = int.from_bytes(os.urandom(32), "little") % self.q
        R = self._point_mult(k, self.G)
        kP = self._point_mult(k, pub_point)
        C = self._point_add(message_point, kP)

        return self._point_compress(R), self._point_compress(C)

    def ElGamal_Dec(self, private_key, R_compressed, C_compressed):
        R = self._point_decompress(R_compressed)
        C = self._point_decompress(C_compressed)
        if R is None or C is None:
            raise ValueError("Invalid ciphertext")

        S = self._point_mult(private_key, R)
        S_neg = (-S[0] % self.p, S[1], S[2], -S[3] % self.p)  # Negação (-x, y)
        M = self._point_add(C, S_neg)
        return M
```

# ElGamal em Curva Eliptica IND-CCA

1. É herdado as propriedades da Curva `Edwards25519` com o *init*

2. É definido a função **ElGamal_GenKeys_CCA** que retorna a `public_key` e a `private_key`

   A. É gerado a `private_key` tal que $0 <$ private_key $< q$ reduzido ao módulo de $q$
   B. É gerado a `public_key` tal que $s.G$
      a. $s$->private_key
      b. $G$->ponto base da curva

C. a public_key é comprimida e é devolvido o par "sk,pk" como esperado de um ElGamal

3. É definido a função **ElGamal_Enc_CCA** que retorna o `ciphertext` $(R, C)$

   A. A public_key é descomprimida para obter $P = s.G$
   B. É escolhido um valor descartável aleatório $r$ entre $0$ e $q$
   C. É obtido $k$ derivando $r$ numa hash
   D. É Calculado $R = k.G$ e $kP = k.P$
      a. $R = k.G$ equivale $g^k$ em ElGamal do exercício 1
      b. $kP = k.P$ equivale $(g^s)^k$ em ElGamal do exercício 1
   E. É cifrada a mensagem apartir da soma dos pontos resultando no R
   F. É gerar a `tag` com um hash de $(R||C||r)$
   G. Retorna $R$ e $C$ comprimidos em tuplo

4. É definido a função **ElGamal_Dec_CCA** que retorna o `ponto na curva decifrado`

   A. A $R$ e $C$ são descomprimidos
   B. É calculado o ponto $S = s.R = s.(k.G)$ usando a private_key $s$
   C. É invertido o eixo $x$ de S para facilitar na "adição" (obtermos assim a subtração para recuperarmos a mensagem)
   D. É verificado a "tag" para detetar algum ataque à `integridade da mensagem`
   E. Devolve o ponto decifrado se não estiver afetado

In [115…

```python
class EC_ElGamal_CCA(Edwards25519):
    def __init__(self):
        super().__init__()
        self.H = lambda x: hashlib.sha256(x).digest()
        self.G_point = self.G

    def ElGamal_GenKeys_CCA(self):
        private_key = int.from_bytes(os.urandom(32), "little") % self.q
        public_key = self._point_mult(private_key, self.G_point)
        return private_key, self._point_compress(public_key)

    def ElGamal_Enc_CCA(self, public_key, message_point):
        pub_point = self._point_decompress(public_key)
        if pub_point is None:
            raise ValueError("Invalid public key")

        r = os.urandom(32)
        k_bytes = self.H(r)
        k = int.from_bytes(k_bytes, "little") % self.q

        R = self._point_mult(k, self.G_point)
        kP = self._point_mult(k, pub_point)
        C = self._point_add(message_point, kP)

        # Gerar a tag
        #hash(R || C || r)
        tag_input = self._point_compress(R)+ self._point_compress(C) + r
        tag = self.H(tag_input)

        return self._point_compress(R), self._point_compress(C), r, tag
```

```python
    def ElGamal_Dec_CCA(self, private_key, R_compressed, C_compressed, r, tag):
        R = self._point_decompress(R_compressed)
        C = self._point_decompress(C_compressed)
        if R is None or C is None:
            raise ValueError("Invalid ciphertext")

        S = self._point_mult(private_key, R)
        S_neg = (-S[0] % self.p, S[1], S[2], -S[3] % self.p)
        M = self._point_add(C, S_neg)

        # Verificar a tag
        #hash(R || C || r) (inverso da cifragem)
        tag_input = self.H(R_compressed + self._point_compress(C) + r)

        if tag_input != tag:
            raise ValueError("Ciphertext integrity check failed (CCA security)")

        return M
```

# Oblivious Transfer em Curva Elíptica ElGamal

1. É herdado as propriedades da Curva `Edwards25519` com o *init*
    A. É definido $n$ e $\kappa$ segundo o protocolo
    B. É Criado algumas hashs especificas para o protocolo ser mantido "fiel"
2. É definido a função **ElGamal_GenKeys_OT** que retorna o `seed` , `lista de chaves privadas` , `lista de chaves publicas` e `tag tau`
    A. É gerado uma `seed` secreta aleatória com 32 bytes
    B. É Calculado uma "tag" $\tau$ tal que $H(seed||I)$
    C. Para cada índice i em I:
        a. $private\_key_i = H(seed||i) \mod q$
        b. $public\_key_i = private\_key_i . G$
    D. Preenchimento da lista
        a. Preenche $public\_key_i[i]$ com chaves públicas para $i \in I$
        b. Para $i \notin I$, gera chaves aleatorias tambme dentro de $G$
3. É definido a função **ElGamal_Enc_OT** que retorna `n ciphertexts` $
    A. É gerado um valor $r$ aleatório com 32 bytes
    B. É mascarado a mensagem tal que $y = m_i \oplus G(r)$
    C. É calculado $r' = H(r||y||\tau)$ e convertido no escalar $k$
    D. É calculado $R = k . G$ (similar ao $g^k$ do ElGamal Tradicional)
    E. É descomprimido $public\_key_i[i]$ para se obter $P_i = private\_key_i . G$
    F. É Calculado $kP = k . P_i$ (similar ao $(g^s)^k$)
    G. É mapeado o valor $M = y . G$
    H. Cifrar o ponto $C = M + kP$
    I. Gerar a verficação $c = F(r||r')$
    J. Retonar n ciphertexts
4. É definido a função **ElGamal_Dec_OT** que retorna o `n msns decifradas`

A. É feito o inverso do anterior retornando as mensagens decifradas

In [116…

```
class ObliviousTransferKofN(Edwards25519):
    def __init__(self, n, k):
        super().__init__()
        self.n = n  # nr Mensagens
        self.k = k  # nr Mensagens pedidas
        self.H = lambda x: hashlib.sha256(x).digest()  # Hash for key derivation
        self.G_hash = lambda x: hashlib.sha256(x + b"g").digest()  # Fog
        self.F = lambda x: hashlib.sha256(x + b"f").digest()  # Randomness check

    # Receiver: Gerar as keys
    def ElGamal_GenKeys_OT(self, I):
        seed = os.urandom(32)  # Secret seed
        tau = self.H(seed + bytes(sorted(I)))  # Authentication tag tau: hash(I,

        # Gerar k chaves privadas e respetivas chaves públicas
        sk = {}
        pk = [None] * self.n
        for i, idx in enumerate(sorted(I), 1):
            seed = self.H(seed + i.to_bytes(4, "little"))
            sk[idx] = int.from_bytes(seed, "little") % self.q
            pk[idx] = self._point_compress(self._point_mult(sk[idx], self.G))

        # Encher o resto do array com chaves públicas aleatórias
        for j in range(self.n):
            if pk[j] is None:
                rand_sk = int.from_bytes(os.urandom(32), "little") % self.q
                pk[j] = self._point_compress(self._point_mult(rand_sk, self.G))

        return seed, sk, pk, tau

    # Provider: Cifrar as mensagens
    def ElGamal_Enc_OT(self, p, messages, tau):
        assert len(messages) == self.n, "Number of messages must match n"
        ciphertexts = []
        p_points = [self._point_decompress(pk) for pk in p]

        for i in range(self.n):
            m = messages[i]
            r = os.urandom(32)  # Randomness
            y = bytes(a ^ b for a, b in zip(m, self.G_hash(r)))  # XOR FOG
            r_prime = self.H(r + y + tau)  # Tau randomness
            k = int.from_bytes(r_prime, "little") % self.q
            R = self._point_mult(k, self.G)
            kP = self._point_mult(k, p_points[i])
            M_point = self._point_mult(int.from_bytes(y, "little") % self.q, sel
            C = self._point_add(M_point, kP)
            c = self.F(r + r_prime)  # Consistency check
            ciphertexts.append((y, self._point_compress(R), self._point_compress

        return ciphertexts

    def ElGamal_Dec_OT(self, sk, ciphertexts, tau):

        assert len(ciphertexts) == self.n, "Number of ciphertexts must match n"
        decrypted_messages = {}

        for idx in sk.keys():  # Apenas os índices em I têm chaves privadas
```

```
            y, R_compressed, C_compressed, c, r = ciphertexts[idx]
            R = self._point_decompress(R_compressed)
            C = self._point_decompress(C_compressed)
            if R is None or C is None:
                raise ValueError(f"Invalid ciphertext for index {idx}")

            # Calcula S = sk[idx] * R
            S = self._point_mult(sk[idx], R)
            S_neg = (-S[0] % self.p, S[1], S[2], -S[3] % self.p)  # Negação do p
            M = self._point_add(C, S_neg)  # Recupera o ponto M = C - S

            # Verifica se M corresponde a y * G
            y_recovered = int.from_bytes(y, "little") % self.q
            M_expected = self._point_mult(y_recovered, self.G)
            if not self._point_equal(M, M_expected):
                raise ValueError(f"Decryption failed for index {idx}: point mism

            # Verifica consistência com c
            r_prime = self.H(r + y + tau)
            if c != self.F(r + r_prime):
                raise ValueError(f"Consistency check failed for index {idx}")

            # Recupera a mensagem original: m = y ⊕ G_hash(r)
            m = bytes(a ^ b for a, b in zip(y, self.G_hash(r)))
            decrypted_messages[idx] = m

        return decrypted_messages
```

## Correr e Testar as Classes acima

1. ElGamal CPA
2. ElGamal CCA
3. ElGamal em Oblivious Transfer

In [117…

```
def test_ElGamalCPA():
    ec_elgamal = EC_ElGamal_CPA()

    # Gerar as keys
    priv_key, pub_key = ec_elgamal.ElGamal_GenKeys()
    print(f"Private Key: {priv_key}")
    print(f"Public Key: {pub_key.hex()}")

    # cifrar a mensagem
    message_point = ec_elgamal.G
    R, C = ec_elgamal.ElGamal_Enc(pub_key, message_point)
    print(f"Ciphertext (R, C): {R.hex()}, {C.hex()}")

    # decifrar a mensagem
    decrypted_message = ec_elgamal.ElGamal_Dec(priv_key, R, C)
    print(f"Decrypted Message: {decrypted_message}")
    print(f"MEssage Point: {message_point}")
    assert ec_elgamal._point_equal(message_point, decrypted_message)
    print("Decryption successful")

def test_ElGamalCCA():
    ec_elgamal = EC_ElGamal_CCA()
```

```
    # gerar as chaves
    priv_key, pub_key = ec_elgamal.ElGamal_GenKeys_CCA()
    print(f"Private Key: {priv_key}")
    print(f"Public Key: {pub_key.hex()}")

    # cifrar a mensagem
    message_point = ec_elgamal.G
    R, C, r, tag = ec_elgamal.ElGamal_Enc_CCA(pub_key, message_point)
    print(f"Ciphertext (R, C, r, tag): {R.hex()}, {C.hex()}, {r.hex()}, {tag.hex

    # decifrar a mensagem
    decrypted_message = ec_elgamal.ElGamal_Dec_CCA(priv_key, R, C, r, tag)
    print(f"Decrypted Message: {decrypted_message}")
    print(f"MEssage Point: {message_point}")
    assert ec_elgamal._point_equal(message_point, decrypted_message)
    print("Decryption successful")

def test_ElGamalCCA_OT():
    ot = ObliviousTransferKofN(4, 2)

    # Receiver: gerar as keys e autenticação
    s, sk, pk, tau = ot.ElGamal_GenKeys_OT([0, 2])
    print(f"Secret Seed: {s.hex()}")
    print(f"Private Keys: {sk}")
    print(f"Public Keys: {[pk_i.hex() for pk_i in pk]}")
    print(f"Authentication Tag: {tau.hex()}")

    # Provider: ciphertexts
    messages = [b"Hello", b"World", b"Foo", b"Bar"]
    ciphertexts = ot.ElGamal_Enc_OT(pk, messages, tau)
    for i, (y, R, C, c, r) in enumerate(ciphertexts):
        print(f"Message {i}: {messages[i]}")
        print(f"Ciphertext (y, R, C, c, r): {y.hex()}, {R.hex()}, {C.hex()}, {c.

    # Receiver: decifrar as mensagens
    decrypted_messages = ot.ElGamal_Dec_OT(sk, ciphertexts, tau)
    for i, m in decrypted_messages.items():
        print(f"Decrypted Message {i}: {m}")
        assert m == messages[i], f"Decrypted message {m} does not match original
    print("Decryption successful")
```

In [118…

```
print("=== Testes ===")
print("CPA ->")
test_ElGamalCPA()
print("===")
print("CCA ->")
test_ElGamalCCA()
print("===")
print("CCA OT ->")
test_ElGamalCCA_OT()
```

```
=== Testes ===
CPA ->
Private Key: 336499984632062093484748441681192234800848987133005616138628623871790
08889663
Public Key: 214fed5ee81e36f8ba1e0803efbbe4f7de10313531191c38114640a6cb40b128
Ciphertext (R, C): 133500b0307c23289ef66b504c676809a67bb990b9e023f4adbe665d6746fb
d8, 9bbd3d7f8d0227abc91a545e3fb7a9a2f247bb93bfac7b6543a1034df4627b1a
Decrypted Message: (-256341229311468359481510727788760774949256101980455350474722
714342155350852078258437759269720168713633602149729895098282941492266780407080358
175057481900, 362644562088856310110391508299368221875762256687051235740845741470 2
675193944708059660763657813484343855672282546394729965737715201666560116092191180
454784, -147510039098851454774828175292769504241666632514069331961182041536224173
489095337835502389317512637522037944702834788491358104268175656864190350118472960
0, 630199499755270237692991074328671249515866378777733683056918756108024013566865
885453186444610864870089094573474561404420771409141312666750517916358738301)
MEssage Point: (151122213495354007725011514095885315114540126930418572060461132 83
949847762202, 463168356949264781694283940034751631413079938662562256157830336031 6
5251855960, 1, 468274038508231792450722166302771975651442055541256549766741658295
33817101731)
Decryption successful
===
CCA ->
Private Key: 393938619559820553871674080004382650978860638127466341774229832697 99
91006660
Public Key: c6cb3bb939f69333b388689b75caedeee5bc82787c5b22b6a5a773f92e2c51b1
Ciphertext (R, C, r, tag): acbe71988b444c90268be6f8a32d54bd540b3b45249933e2c98194
30224903d1, 08682c0c4915104a204effc84029226856fcb6d6c2d1ac5e03cb4378e64b039b, 512
820546dd8fccd2db0f8c03e16b3ba86b8fb4fb6e317754340f2540b717905, 84e42d307f4ddc150d
a84e873fd5ef5a4dda2da1d2b2a0ece3903095bdf21a5d
Decrypted Message: (-177646144381689856818902579174602970231961155995743622649523
566695927756500149023518701069250011297929581450513260734356818559509433138011826
041195498475, 682657691515215109511045275624256216163772791512364147388154606844 4
836502202515080762339597908682568537824518015769773683798158232168805485851333066
440213, 110589779781555221317059355632493303827915724080476861556984248538446975 0
950978625083260670007497319532842749844699453962860001895375726003976268665464 65
5, -109658873604529353270763086386127350685277080754667382551457046984583968528 34
049768230281578986782520709744710094408999436645513345901514957307993525589 85)
MEssage Point: (151122213495354007725011514095885315114540126930418572060461132 83
949847762202, 463168356949264781694283940034751631413079938662562256157830336031 6
5251855960, 1, 468274038508231792450722166302771975651442055541256549766741658295
33817101731)
Decryption successful
===
CCA OT ->
Secret Seed: b0703dd13b1b18f543544c484d1ee34650c8ccb9f383c6de6d6fcf442bef0e61
Private Keys: {0: 376222966094909376262004689898542226495420572217351089445324824
0308412538986, 2: 478699391391323155302899578340444998092069489620306539662447718
920674244898}
Public Keys: ['74afe2964280d67f971ae313e6b58babb76d4da4514f66c9c1cf763fbdc121cd',
'fe54efa3d8f03c3200b250e2a36914b17bd81cf817d502e4b2855785c3966e8c', '0af3e2a1abb6
6e96dd8a64083f4c0d2860884f7e46a5287df0d82e63b131292c', 'd4aab72eac60363facc99d954
4d6cd8975c875f984cb6b1898d833670161ba17']
Authentication Tag: 95d43f49ea7f8508de9f43daf66f24b1ec42a5c991d30deffbac6f70998a9
2e6
Message 0: b'Hello'
Ciphertext (y, R, C, c, r): 0cfe382e7c, 225578009ebb202e50b874d1297ce92113cb75a51
536ee8553f9c5d12fd87bc7, c891adeea8e93178758c8314eb269cea370f9581180cc49090c31f6a
52ed9bd4, 00e822a38152bde339492aa7dd5e2c14e671916414a222c17f6f5b3260d3759f, fd437
9c33c38486bb35938bd5fdee6633c5c35432ddc3b4c89008840d5ba2564
Message 1: b'World'
```

```
Ciphertext (y, R, C, c, r): 60ba7e771f, d3dae8f49e1a97787c5a5f51d9271cf9afb84df66
f0d14294d065fa9587ae15a, 0268045273f18dd470b5db5c87b5b3f4a8e3655720e1db2013dd7365
c7011486, 6ff58f68a4e0e4edc20af744a7ffc849454f2c49b91fa4062eceabd7241b7045, 80bd2
ec924683681fe8b0937471c0de499d373b03d5312482feee5f0d0ae7f91
Message 2: b'Foo'
Ciphertext (y, R, C, c, r): 5c53ec, 91f26ae13a00646f45e05a31a8af1f295a18d88581b60
08f7834bf56b1f03494, 3ce5fc1f532db19d3603519a9fd04225986a1cf219da924e9606e0edd31e
a9f0, f044b3b48ea553c2e7c8ac425ab5d4588b08bda43ee96ee7ea4c1847b562baa5, 3c35d7929
65ff23ea4fda1bddb45c661b4404e787e9da92b07f40f7c8d0dca9b
Message 3: b'Bar'
Ciphertext (y, R, C, c, r): 41d1ab, f7f76afbfb32dca5da884a84f48ac1541c799ec3349bb
f2de9f0e5cd34cd3a51, 140f05e1dc2471e3ae03cf469016bc207fc8aa51a3ff67e7408241db861e
e08e, 0956a30b56a473f3fa05ea2cf6585ea1c80c42553a8618abaca5371cc08e15ab, 0e055dcf4
59f2aecd27f8f3e2aa21e9a45231b4acc9d6d35de74de93774b70f5
Decrypted Message 0: b'Hello'
Decrypted Message 2: b'Foo'
Decryption successful
```