INF245 - Assignment 3

By Vegard Berge

1. Let the parameters of ElGamal signature Algorithm be

> $p = 593831971123607$
> $q = 13$

where p is prime and g is primitive root modulo p. Let x mod $p-1$ be a system private key and $y \equiv 13^x mod\, p$ be the system public key, where

> $y = 239505966643112$

Forge an ElGamal signature without knowledge of the private key, by constructing a triple m, a, b, where m is an integer and a, b is its signature.

In order to forge a signature for a message m we can find $a, b\ s.t: 0 < a < p,\ 0 < b < p-1$ and $g^m \equiv y^a a^b mod\, p$

We use the fact that choosing m, a, b simultaneously is easy. We fin i, j in $0 < i, j < p-2$ s.t $gcd(j, p-1) = 1$

From there we get:

$$a \equiv g^i y^j mod\, p, 0 < a < p,$$
$$j_1 \equiv j^{-1} mod\, p-1$$
$$b \equiv -aj_1 mod\, p-1$$
$$m \equiv -aij_1 mod\, p-1$$

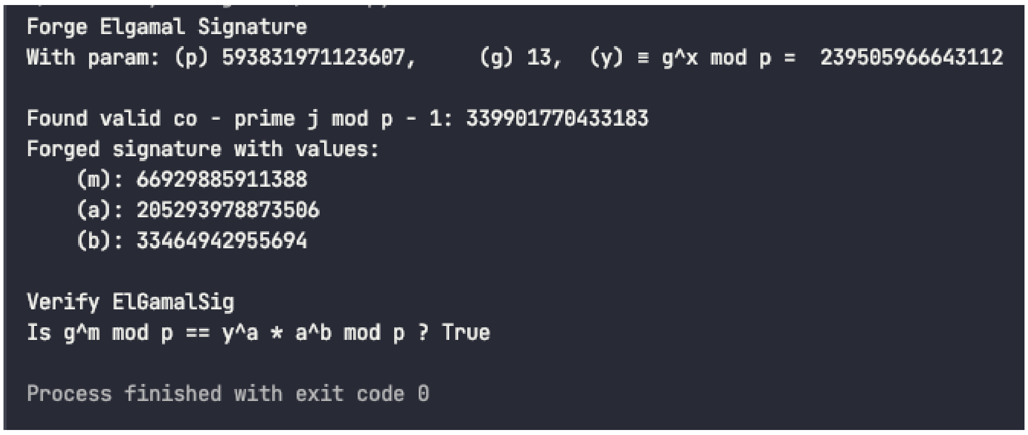
Through some arithmetic operations we get the following:

$y^a b^b \equiv g^m mod\, p$

The code should be some what clean and self explainatory.

```
def findJ(p):
    """
    j should be gcd(j, p-1) = 1
    :param p:
    :return:
    """
    j = 2
    while math.gcd(j, (p - 1)) != 1:
        j = random.randint(1, p - 2)
    return j


def forgeElGamalSigAlgorithm(p: int, g: int, y: int):
    """
    Forge ElGamal signature without knowing the private key
        Construct triple: m, a, b
            m: int
            a, b = Signature
    Forge signature for the message m, find a,b s.t
        0 < a < p,
        0 < b < p - 1
    and
        g^m ≡ y^a * a^b mod p

    :param p: prime p
    :param g: primitive root mod p
    :param y: y ≡ 13^x mod p
    :return:
    """
    # system private key: x mod p - 1
    # public key: y ≡ 13^x mod p

    i, j = 2, findJ(p)
    print(f"\nFound valid co - prime j mod p - 1: {j} ")
    g_i, y_j = binExp(g, i, p), binExp(y, j, p)
    a = (g_i * y_j) % p
    j_1 = pow(j, -1, p - 1)
    b = pow((-a * j_1), 1, p - 1)
    m = pow((-a * i * j_1), 1, p - 1)
    print(f"Forged signature with values: \n\t(m): {m}\n\t(a): {a}\n\t(b): {b}\n")
    return m, a, b


def verifyElGamalSig(p, g, y, m, a, b):
    print("Verify ElGamalSig")
    return binExp(g, m, p) == (binExp(y, a, p) * (binExp(a, b, p)) % p)


def runTaskOne():
    # forge ElGamal
    print(f"Forge Elgamal Signature")
    P = 593831971123607
    G = 13
    Y = 239505966643112
    print(f"With param: (p) {P},  \t (g) {G}, ", end=" ")
    print(f"(y) ≡ g^x mod p =  {Y}")
    m, a, b = forgeElGamalSigAlgorithm(P, G, Y)
    print(f"Is g^m mod p == y^a * a^b mod p ? {verifyElGamalSig(P, G, Y, m, a, b)}")
```

output of program:



```
Forge Elgamal Signature
With param: (p) 593831971123607,     (g) 13,  (y) ≡ g^x mod p =  239505966643112

Found valid co - prime j mod p - 1: 339901770433183
Forged signature with values:
    (m): 66929885911388
    (a): 205293978873506
    (b): 33464942955694

Verify ElGamalSig
Is g^m mod p == y^a * a^b mod p ? True

Process finished with exit code 0
```

2. Task two: Find DSA private key x
    a. Task 2A, Find DSA private key through the same k attack.

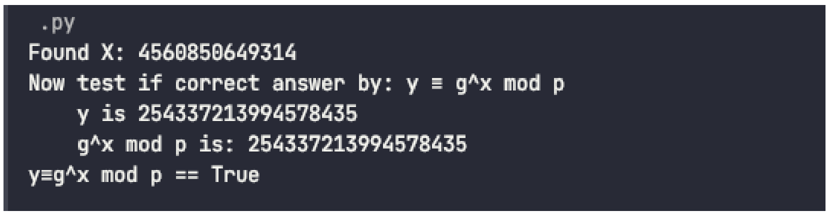
I think the math behind this attack is sufficiently covered in the “recoverXFromSameKDSA” sub - function.

```
def recoverXFromSameKDSA(q, r_1, s_1, s_2, m_1, m_2):
    """
    When the same k is used to generate the signatures of two messages: m1 & m2, we can easily recover the
    secret x.
    The following should hold:
    Given the two signatures M1 & M2, we solve for k
        s1 = inv(k)*(M1 + x*r) mod q
        s2 = inv(k)*(M2 + x*r) mod q
        # subtract both signatures:
        (**All is under mod q)
            s1 - s2 = inv(k)(M1 + x*r) - inv(k)(M2 + x*r)
            s1 - s2 = inv(k)(M1 + x*r - M2 + x*r) # reduce exp.
            s1 - s2 = inv(k)(M1 - M2) # reduce exp.
        Thus,
            k = (M1 - M2) / (s1 - s2)
            recover x from the expression
                x = r^-1 * (s * k - M) mod q
    :return: Secret x
    """
    delta = pow((s_1 - s_2) % q, -1, q)
    m_ = (m_1 - m_2) % q
    k = (m_ * delta) % q
    rINV = pow(r_1, -1, q)
    x1 = (rINV * (s_1 * k - m_1)) % q
    x2 = (rINV * (s_2 * k - m_2)) % q
    assert x1 == x2, "Recover x failed"
    return x1


def verifyXDSA(g, x, p, y):
    return y == binExp(g, x, p) and y == pow(g, x, p)
```

Driver code for this task:

```
def runTaskTwo():
    """"""
    p = 949772751547464211
    q = 4748626326421
    g = 314668439607541235

    # y = g^x mod p
    y = 254337213994578435

    m_1, m_2 = 2393923168611338985551149, 9330804276406639874387938

    sig_1, r_1, s_1 = 2393923168611338985551149, 2381790971040, 3757634198511
    sig_2, r_2, s_2 = 9330804276406639874387938, 2381790971040, 4492765251707


    x = recoverXFromSameKDSA(q, r_1, s_1, s_2, m_1, m_2)
    print(f"Found X: {x}")
    print(f"Now test if correct answer by: y ≡ g^x mod p")
    print(f"\ty is {y}")
    print(f"\tg^x mod p is: {binExp(g, x, p)}\ny≡g^x mod p == {verifyXDSA(g, x, p, y)}")
```

Output of program:



```
 .py
Found X: 4560850649314
Now test if correct answer by: y ≡ g^x mod p
    y is 254337213994578435
    g^x mod p is: 254337213994578435
y≡g^x mod p == True
```

Task 2B

Compute the DSA private key x by solving the discrete algorithm problem with ρ method

ρ - method, This is a quite standard implemetation (followed course lecture notes), only design choices I want to note, is that I found it in our case *faster* computationally to choose the parameter b to be zero, and not random for the initial setup.

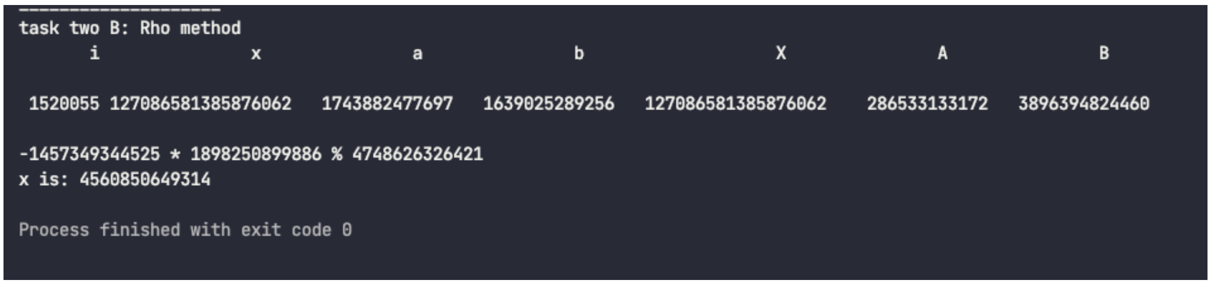
```python
# Rho - method implementationd
def fab(x, a, b, alfa, beta, N, n):
    if x % 3 == 0:
        x = pow(x, 2, N)  # x * x % N
        a = a * 2 % n
        b = b * 2 % n

    elif x % 3 == 1:
        x = x * alfa % N
        a = (a + 1) % n
    else:
        x = x * beta % N
        b = (b + 1) % n

    return x, a, b


def solveForX(a, b, A, B, n):
    print(f"a {(A - a)} * {pow(b - B, -1, n)} % {n}")
    return ((A - a) * pow((b - B), -1, n)) % n


def f(N=949772751547464211, n=4748626326421, alfa=314668439607541235, beta=254337213994578435, y=254337213994578435):
    x, a, b, = 1, 0, 0
    X, A, B = x, a, b
    print("%8s %15s %15s %15s  %18s %15s %15s\n" % ("i", "x", "a", "b", "X", "A", "B"))
    for i in range(1, n):
        x, a, b = fab(x, a, b, alfa, beta, N, n)
        X, A, B = fab(X, A, B, alfa, beta, N, n)
        X, A, B = fab(X, A, B, alfa, beta, N, n)
        if x == X and math.gcd(b - B, n) == 1:
            print("%8d %15d %15d %15d   %18d %15d %15d\n" % (i, x, a, b, X, A, B))
            x = solveForX(a, b, A, B, n)
            print(f"x is: {x}")
            break
```

Driver code:

```python
def runTaskTwo():
    """"""
    p = 949772751547464211
    q = 4748626326421
    g = 314668439607541235
```

```
# y = g^x mod p
y = 254337213994578435

m_1, m_2 = 2393923168611338985551149, 9330804276406639874387938

sig_1, r_1, s_1 = 2393923168611338985551149, 2381790971040, 3757634198511
sig_2, r_2, s_2 = 9330804276406639874387938, 2381790971040, 4492765251707

print("_" * 20)
print("task two B: Rho method")

f()
```

Output of the program:



```
____________________
task two B: Rho method
      i              x               a              b                X              A             B

 1520055 127086581385876062   1743882477697   1639025289256   127086581385876062    286533133172   3896394824460

-1457349344525 * 1898250899886 % 4748626326421
x is: 4560850649314

Process finished with exit code 0
```

Task 3

Implement the *Index Calculus Algorithm*, with parameter:

> $p = 2602163$
> $g = 2$
> $2^x \equiv 1535637 \; mod \; p$

Again, followed course notes from lecture. Implementation uses the same matrix to row echelon form used in previous assignments.

This algorithm surprised me in both speed, and its relative easy steps! Summarized, the algorithm does this:

Find random y in $1 \leq y < p - 1$, find $g^y \equiv b \; mod \; p$ such that $2 \leq b < p$

Check if b is B - smooth. if no: choose new y

if yes:

$b = \prod_{i=1}^{n} q_i^{l_i}$
$g^y \equiv b = \prod_{i=1}^{n} q_i^{l_i} mod \; p$

We collect n such congruences

$$y_j \equiv \sum_{i=1}^{n} l_{li} x_i \; mod\, p-1,\; 1 \leq j \leq m$$

Put all the congruences in a matrix, and perform Gaussian elimination to find unique solution ⇒ finding x_i $mod\, p-1$

If system is unique, continue, else try with more congruences or try again.

Then find random y where $0 \leq y < p-1$ and find $a * g^y \equiv b \; mod\, p$

b is B - smooth

no: take another y

yes:

$$b = \prod_{i=1}^{n} q_i^{l_i}$$

$$a * g^y \equiv b = \prod_{i=1}^{n} q_i^{l_i x_i} mod\, p$$

We have thus found congruence that implies:

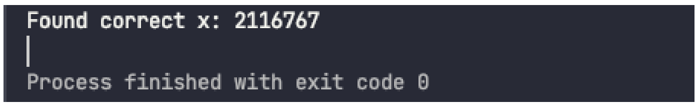
$y + x \equiv \sum_{i=1}^{n} l_i x_i mod\, p-1$ ⇒ compute $x \; mod\, p-1$

```
def IndexCalculusAlgorithmj(p=2602163, alfa=1535637, B=30, g=2):
    S_b, m, n, random_X_s = FindRowsBsmooth(B, p)
    out = None
    while out is None:
        try:
            while True:
                out = rref_mod_n(random_X_s, p - 1)
                a = np.array(out, dtype=int)
                matrix = a.reshape(m, 11)

                if matrix[0][-1] == 0:
                    S_b, m, n, random_X_s = FindRowsBsmooth(B, p)
                else:
                    printMatrix(matrix)
                    while True:
                        y = random.randint(0, p - 1)
                        b = (pow(g, y, p) * alfa) % p
                        fac_ = prime_factors(b)
                        if is_b_smooth(fac_, S_b):
                            break

                    # solve:
                    l = [0] * n
                    for i_s in range(len(S_b)):
                        l[i_s] = fac_.count(S_b[i_s])
                    x_vals = findXi(matrix)
                    x = 0
                    for x_i, l_i in zip(x_vals, l):
                        x += x_i * l_i  # (x_i**l_i)
                    x = (x - y) % (p - 1)

                    return x
        except:
            S_b, m, n, random_X_s = FindRowsBsmooth(B, p)


def FindRowsBsmooth(B, p):
    S_b = createBsmooth(B)
    n = len(S_b)
    random_X_s = []
    # find m + c rows
    m = n
    while len(random_X_s) != m:
        # find random b
        x_s = random.randint(math.floor(math.sqrt(p)), p - 1)
        # compute congruence
        b = pow(2, x_s, p)  # (x_s ** 2) % p
        if b == 1:
            continue  # if prime and >= B ??
        # find factors
        factors_of_b = prime_factors(b)
        if max(factors_of_b) > B: continue
        # check if valid
        if is_b_smooth(factors_of_b, S_b):
            l = [0] * n
            for i_s in range(len(S_b)):
                l[i_s] = factors_of_b.count(S_b[i_s])
            temp = l
            temp.append(x_s)
            random_X_s.append(temp)
            # random_X_s.append((x_s, l))
    return S_b, m, n, random_X_s


def findXi(m: list):
    xi, piv = [], 0
    for _ in m:
        xi.append(m[piv][-1])
        piv += 1
    return xi
```

Driver code

```
def runTaskThree():
    while True:
        x = IndexCalculusAlgorithmj()
        if x is not None:
            x = int(x)
            if pow(2, x, 2602163) == 1535637:
                print(f"Found correct x: {x}")
                break
            else:
                print(f"Wrong: got x ==  {x}", end="\n")
                print()
```

Output of the algorithm with task parameters:



```
Found correct x: 2116767

Process finished with exit code 0
```

Tested & verified, correct solution