# ECE 4802 Project 5

Thomas Mackintosh

December 1, 2016

## 1 8.4

In this exercise we want to identify primitive elements (generators) of a multiplicative group since they play a big role in the DHKE and and many other publickey schemes based on the DL problem. You are given a prime p = 4969 and the corresponding multiplicative group $\mathbf{Z}^*_{4969}$.

1. Determine how many generators exist in $\mathbf{Z}^*_{4969}$. We can use the following code to determine the number of generators in $\mathbf{Z}^*_{4969}$:

```
import math

gen_count = 0

for i in range(1,4969):
    temp = math.gcd(i,4968)
    if temp == 1:
        gen_count += 1

print(gen_count) #1584
```

This gives us a result of 1584 generators for $\mathbf{Z}^*_{4969}$.

2. What is the probability of a randomly chosen element a $\in \mathbf{Z}^*_{4969}$ being a generator?

$$P(\text{a is a generator}) = 1584 \div 4969 = 0.3188$$

3. Determine the smallest generator a $\in \mathbf{Z}^*_{4969}$ with $a > 1000$.

The following code finds the first generator a $\in \mathbf{Z}^*_{4969}$ where $a > 1000$. The code uses the squareMultiply function I create in a later section. pow() would also work in place of squareMultiply.

```
#generators.py
#An integer is a generator if integer ^ 4968 = 1 mod 4969

for a in range(1000,5000):
    temp = squareMultiply(a,4968,4968)
    if temp == 1:
        print(a)
        break
```

The code gives us a result of 1013 as the first generator $a > 1000$.

4. What measures can be taken in order to simplify the search for generators for arbitrary groups $\mathbf{Z}_p^*$?

There are $\phi(p-1)$ generators for $\mathbf{Z}_p^*$. To make generators to be easy to find, we can choose a 'safe prime' $p = 2q + 1$ where q is prime since then $\phi(p-1) = \phi(2q) = q - 1$. This would cause half the group to be generators compared to the current third.

## 2 Square Multiply Algorithm

Implement the square and multiply algorithm using a computer language of your choice.

Below is my program which implements the square and multiply algorithm in Python:

```
#square_multiply.py
#Implements the square and multiply algorithm

def squareMultiply(b, e, m):
    r = 1
    while e > 0:

        if e % 2 == 1:
            r = (r * b) % m
        b = (b * b) % m
        e >>= 1
    return r

res = squareMultiply(275973,456789,583903)
print("For a = 235973; e = 456789; p = 583903, we get: ", res)

res = squareMultiply(984327457683,2153489582,994348472629)
print("For a = 984327457683; e = 2153489582; p = 994348472629, we get: ", res)
```

From this code we get the following output:

```
For a = 235973; e = 456789; p = 583903, we get:  277334
For a = 984327457683; e = 2153489582; p = 994348472629, we get:  331688688384
```

## 3 DHKE

The goal of this problem is to implement the Diffie Hellman Key Exchange. Below is my implementation DHKE.py:

```
import random
```

```
from square_multiply import squareMultiply
import hashlib

def isPrime(p):
    if(p==2): return True
    if(not(p&1)): return False
    return squareMultiply(2,p-1,p)==1

def rand_prime():
    while True:
        p = random.getrandbits(1023)
        q = int(p)
        if isPrime(q) == True:
            return q

def gen_Key(k_A):
    hashkey = hashlib.sha224(hex(k_A)[2:-1].decode('hex'))
    return hashkey

def checkEqual(a,b):
    if a == b:
        print "k_A = k_B"
    else:
        print "Keys not equal"

random.seed(12345)

p = rand_prime()
p_safe = int("""
136493091133649836164289363338682002944029379014077033130604363922256595
037775025761962540974136000872788770954019530710825866837156972143526820\
463536654299014569407235702375016873961943932744861043443226534544334946\
757053788912168896006231429590913701778250440452439749688803485407941150\
333770430825062090319""".replace("\n",""))

a = random.randrange(1,p_safe-2)
b = random.randrange(1,p_safe-2)
g = random.randrange(1,p_safe-1)
A = squareMultiply(g,a,p_safe)
B = squareMultiply(g,b,p_safe)

print "a= ", a,"\n\nb= ",b,"\n\np= ",p, "\n\np_safe= ", p_safe, "\n\ng= ",g, "\n"

k_A = squareMultiply(A,b,p_safe)
k_B = squareMultiply(B,a,p_safe)
checkEqual(k_A,k_B)

print "Hashed key: ",gen_Key(k_A).hexdigest()
```

Below is the output of DHKE.py:

```
a=
113932264828811031499998907582354125839977544700394975992106306
541368094048343738673665232070237154511881603757538021018841028
403063666653255093534336696687753713332940126767877373833615248
392916395921339141647849520042035488629508783383430347580636842
546861310780051238545125136953574482770599097074594064508

b=
126626831213495643304735514354614556800261562053466595780231899
358989161286924851285810952783577819170756527592438994158731276
746424153476456379226153790428920685854369667993183146166866881
223915207991912774025452899327038993796224713499507935329742695
616152820481902000507295224703996190730617499207101085820

p=
523633092592178195893526328469239769606612725211844383462007177
951575938155186456020928588747527192409767947469066041434417536
080791321590420184634380944668733233648080355859275748383165406
209569453289981241748804766181443112929482799639828141249950440
367735217611209906414511473188150852313941103025427160 97

p_safe=
136493091133649836164289363338682002944029379014077033130604363
922256595037775025761962540974136000872788770954019530710825866
837156972143526820463536654299014569407235702375016873961943932
744861043443226534544334946757053788912168896006231429590913701
778250440452439749688803485407941150333770430825062090319

g=
218664716382907224032769338234721593000728013493733282508968368
268144566917238746876102897991182842205365544444737537716345776
976788062148658164102024339885546656326412043391576494706734717
366992088052216152656376977617225261050893449328066614544330277
755491804506750164511141871677202541681218477252801349 20

k_A = k_B
Hashed key:   80b3d1eed89742514603fb4998f41fd5c18a7b91869e6d040974c088
```

# 4 ElGamal Encryption

The ElGamal encryption scheme is non-deterministic: A given message m has many valid encryptions.

(a) Why is the ElGamal encryption scheme non-deterministic?

The encryption function for the ElGamal encryption is $c' = m \times h^a$. $m$ could always be the same, but the shared key $h^a = g^{ab} \bmod p$ will change as values for a and b change.

(b) How many valid ciphertexts exist for each message m (general expression)? How many are there for the system in problem 8.13 from the book (numerical answer)?

From the above equation for ciphertext encryption, we see there are $h^a = g^{ab} \bmod p$ possible ciphertexts.

From 8.13:
$k_M = \beta^i \bmod p$ where $\beta = \alpha^d \bmod p$, $\alpha = 2$, $p = 467$
1. $k_{pr} = d = 105, i = 213, x = 33$
2. $k_{pr} = d = 105, i = 123, x = 33$
3. $k_{pr} = d = 300, i = 45, x = 248$
4. $k_{pr} = d = 300, i = 47, x = 248$
We may use the following code to find the number of possible ciphertexts i.e. $k_M$:

```
from square_multiply import squareMultiply

def getKey_M(a, d, i, p):
    beta = squareMultiply(a, d, p)
    k_m = squareMultiply(beta, i, p)

    return k_m

a = 2
p = 467

arr_d = [105, 105, 300, 300]
arr_i = [213, 123, 45, 47]

for i in range(4):
    temp = getKey_M(a, arr_d[i], arr_i[i], p)
    print "Number of ciphertexts for ", i + 1, ":", temp
```

From this code we get the following output:

```
Number of ciphertexts for  1 : 292
Number of ciphertexts for  2 : 278
Number of ciphertexts for  3 : 12
Number of ciphertexts for  4 : 74
```

(c) Consider the case that for two messages $m_1 \neq m_2$ the same session parameter $y_1 = y_2$ has been chosen for the ElGamal encryption. This kind of behavior occurs if no or a bad cryptographic PRNG is being used. Show how the message $m_2$ can be recovered from a known message ciphertext pair $\langle m_1, c_1 \rangle$ if the same y is used.

Knowing pair $\langle m_1, c_1 \rangle$, we can do a search for the integer $y$ in $c_1 \equiv m_1 \times y \bmod p$. We can take the result of the search $y$ and decrypt for the pair $\langle m_2, c_2 \rangle$, assuming we know a ciphertext $c_2$, using the decryption function $m_2 = c_2 \times y^{-1} \bmod p$.