Question 6)

Suppose we want to create a Random Number Generator with hardness based on the Discrete Log problem. In this problem we will investigate such an RNG and show that it has some weaknesses

First, suppose that we have primes *P*, *Q* such that *P* = 2∙*Q* + 1. Now suppose that we have two points *X* and *Y* with multiplicative order *Q* mod *P*. This means that *XQ* ≡ *YQ* ≡ 1 mod *P*.

Let s[i] denote the value of the internal state at time i. We generate the following values as follows:

The intermediate data value: t[i] = s[i]  
The next internal state s[i+1] = Xt[i] mod P  
The output of the generate function o[i] = Yt[i] mod P

The following diagram shows the flow of the RNG.

We will call this RNG the Dual DL Rng (for Dual Discrete Log Random Number Generator)

For the following questions feel free to use Sage’s built in modular exponentiation functionality, the example function for modular exponentiation, or any functions from previous problems.

1. Implement a Sage function that takes primes *P, Q* and integers *X*, *Y* of multiplicative order *Q* mod *P* and generates a random initial internal state *s* (an integer reduced mod *Q*). Have this function return a list with entries [P, Q, X, Y, s].
2. Implement a Sage function that takes as a parameter a five element list corresponding to the internal state initialized by the function you wrote for part (a). This function should generate a single block of output, and update the list parameter’s last element to correspond to the next RNG state.
3. Suppose that we have *P* = 15116301544809716639, *Q* = 7558150772404858319, *X* = 10655637283854386401, *Y* = 5886823825742381258, and furthermore we know that *Xe* ≡ *Y* mod *P*, where *e* = 1534964830632783921. Find the positive integer *f* such that *Yf* ≡ *X* mod *P.* [Hint: remember that *XQ* ≡ 1 mod *P*. Find the positive integer *f* such that *e*∙*f* = 1 + *k*∙*Q*.]
4. Now, using the values for *P*, *Q*, *X*, *Y* from part (c), write a Sage function that, given one output of the generate function (from part (b)) and gives the output from the next call to the generate function. Use the functions you wrote in part (a) and (b) to show that your function works. [HINT: What happens if you exponentiate the output of the generate function by the value you found in part (c).]
5. Write a version of the function that you wrote in part (b) that takes only *P*, *Q*, and *X*. Have it generate the value *Y* in a manner such that you know the positive integer *f* such *Yf* ≡ *X* mod *P*. Your function should return a tuple (rngstate, *f*) where rngstate, is a valid rng state like the function from part (b) returns.
6. Generalize your attack function from part (d) to work given a block of output, with the *Y* and *f* values you generated in part (e).
7. How would you modify this RNG to overcome this problem?

Solutions to Question 6)

Suppose we want to create a Random Number Generator with hardness based on the Discrete Log problem. In this problem we will investigate such an RNG and show that it has some weaknesses

First, suppose that we have primes *P*, *Q* such that *P* = 2∙*Q* + 1. Now suppose that we have two points *X* and *Y* with multiplicative order *Q* mod *P*. This means that *XQ* ≡ *YQ* ≡ 1 mod *P*.

Let s[i] denote the value of the internal state at time i. We generate the following values as follows:

The intermediate data value: t[i] = s[i]  
The next internal state s[i+1] = Xt[i] mod P  
The output of the Generate function o[i] = Yt[i] mod P

The following diagram shows the flow of the RNG.

We will call this RNG the Dual DL Rng (for Dual Discrete Log Random Number Generator)

For the following questions feel free to use Sage’s built in modular exponentiation functionality, the example function for modular exponentiation, or any functions from previous problems.

1. Implement a Sage function that takes primes *P, Q* and integers *X*, *Y* of multiplicative order *Q* mod *P* and generates a random initial internal state *s* (an integer reduced mod *Q*). Have this function return a list with entries [P, Q, X, Y, s].

def DualDLSetup(p, q, X, Y):

state = Integer(randint(2, q-1))

ret = [p, q, X, Y, state]

return ret

1. Implement a Sage function that takes as a parameter a five element list corresponding to the internal state initialized by the function you wrote for part (a). This function should generate a single block of output, and update the list parameter’s last element to correspond to the next RNG state.

def DualDLGenerate(rng\_state):

r"""

Generates a single output of the Dual DL Rng.

"""

p = rng\_state[0]

q = rng\_state[1]

X = rng\_state[2]

Y = rng\_state[3]

state = rng\_state[4]

temp = ModExp(X, state, p) % q

next\_state = temp

output = ModExp(Y, temp, p)

# update the next state

rng\_state[4] = next\_state

return output

1. Suppose that we have *P* = 15116301544809716639, *Q* = 7558150772404858319, *X* = 10655637283854386401, *Y* = 5886823825742381258, and furthermore we know that *Xe* ≡ *Y* mod *P*, where *e* = 1534964830632783921. Find the positive integer *f* such that *Yf* ≡ *X* mod *P.* [Hint: remember that *XQ* ≡ 1 mod *P*. Find the positive integer *f* such that *e*∙*f* = 1 + *k*∙*Q*.]

sage: p = 15116301544809716639

sage: q = 7558150772404858319

sage: X = 10655637283854386401

sage: Y = 5886823825742381258

sage: e = 1534964830632783921

sage: # find the multiplicative inverse of e mod q

sage: f = xgcd(e, q)[1] % q

sage: ModExp(Y, f, p)

10655637283854386401

sage: f

6285400490143952176

sage: ModExp(Y, f, p)

10655637283854386401

sage: X == ModExp(Y, f, p)

True

1. Now, using the values for *P*, *Q*, *X*, *Y* from part (c), use Sage to show that, given one output of the generate function (from part (b)) you can predict the next output from the next call to the generate function. Use the functions you wrote in part (a) and (b). [HINT: What happens if you exponentiate the output of the generate function by the value you found in part (c).]

sage: o = DualDLGenerate(rngstate); o

2883667907655668839

sage: # predict the next output

sage: nexttemp = ModExp(o, f, p)

sage: nextoutput = ModExp(Y, nexttemp, p)

sage: nextoutput

12623496892479340564

sage: o2 = DualDLGenerate(rngstate); o2

12623496892479340564

sage: o2 == nextoutput

True

sage: # and we can do it again...

sage: nexttemp = ModExp(o2, f, p)

sage: nextoutput = ModExp(Y, nexttemp, p)

sage: nextoutput

4541819456189879395

sage: o3 = DualDLGenerate(rngstate); o3

4541819456189879395

sage: o3 == nextoutput

True

1. Write a version of the function that you wrote in part (b) that takes only *P*, *Q*, and *X*. Have it generate the value *Y* in a manner such that you know the positive integer *f* such *Yf* ≡ *X* mod *P*. Your function should return a tuple (rngstate, *f*) where rngstate, is a valid rng state like the function from part (b) returns.

def DualDLSetupWithBackdoor(p, q, X):

state = Integer(randint(2, q-1))

backdoor\_inv = Integer(randint(2, q-1))

backdoor = xgcd(backdoor\_inv,q)[1] % q

Y = ModExp(X, backdoor\_inv, p)

rngstate = [p, q, X, Y, state]

return (rngstate, backdoor)

1. Generalize your attack from part (d) to work given a block of output, as well as *P*, *X* and with the *Y* and *f* values you generated in part (e).

def DualDLPredictNextOutput(p, X, Y, output, backdoor):

r"""

Given p, X, Y, one block out output and the backdoor,

"""

nexttemp = ModExp(output, backdoor, p)

nextoutput = ModExp(Y, nexttemp, p)

return nextoutput

The output of this function is shown below:

sage: (rngstate, backdoor) = DualDLSetupWithBackdoor(p, q, X)

sage: Y = rngstate[3]

sage: backdoor

89069958402085048

sage: o1 = DualDLGenerate(rngstate)

sage: nextoutput = DualDLPredictNextOutput(p, X, Y, o1, backdoor)

sage: nextoutput

14777415054912272669

sage: o2 = DualDLGenerate(rngstate)

sage: o2

14777415054912272669

sage: nextoutput == o2

True

sage: nextoutput = DualDLPredictNextOutput(p, X, Y, o2, backdoor)

sage: nextoutput

14256674731245031230

sage: o3 = DualDLGenerate(rngstate)

sage: o3

14256674731245031230

sage: o3 == nextoutput

True

1. How would you modify this RNG to overcome this problem?

You could truncate off the top half of the bits in the output value. The following sage function shows how this could be done:

def DualDLGenerateWithFix(rng\_state):

r"""

Generates a single output of the Dual DL Rng.

Fixes the forward security attack by truncating off the MSB from the output.

"""

p = rng\_state[0]

q = rng\_state[1]

X = rng\_state[2]

Y = rng\_state[3]

state = rng\_state[4]

n = p.nbits()

mask = 2^((n - (n%2))/2) - 1

temp1 = ModExp(X, state, p) % q

next\_state = temp1

temp2 = ModExp(Y, temp1, p)

output = mask & temp2

# update the next state

rng\_state[4] = next\_state

return output