Question 1)

Breaking Blum Blum Shub is provably (polynomial time) equivalent to factoring. While this question does not prove this, it does show how to create a Sage function that gives considerable evidence for this fact. Specifically, we will show that given a function that gives you the previous Blum Blum Shub State from a Blum Blum Shub state, that we can write a probabilistic program that factors. The following function will break Blum Blum Shub (for small *N*):

def previous\_BBS\_state(state):

r"""

This function returns the previous BlumBlumShub state.

Note that this is a toy function and will only work on small N.

"""

N = state[0];

R = IntegerModRing(N);

X = R(state[1]);

if (not X.is\_square()):

print "Not a valid Blum-Blum-Shub RNG state."

return None

return [N, X.sqrt().lift()];

1. The first part of the problem is to notice that if you have integers *x*,*y* such that *x* ≠ ±*y* (mod *N*) and , then the usual difference of squares equation gives that . And so we can hope that gcd(*x*-*y*,*N*) or gcd(*x*+*y*,*N*) yield a nontrivial factor of N. Write a sage function that takes *x*,*y* such that *x* ≠ ±*y* (mod *N*) and and tries to find a nontrivial factor of *N*.
2. Using the function you wrote in part (a) and the supplied function previous\_BBS\_state, write a function that takes a number N (that is a product of two primes *p*,*q* both congruent to 3 mod 4) and factors *p*,*q*. [Hint: you have to create your own BBS state, so you will have to choose your square. How do you choose a square such that you know you have x,y?]

Question 1 - Solution

Breaking Blum Blum Shub is provably (polynomial time) equivalent to factoring. While this question does not prove this, it does show how to create a Sage function that gives considerable evidence for this fact. Specifically, we will show that given a function that gives you the previous Blum Blum Shub State from a Blum Blum Shub state, that we can write a probabilistic program that factors. The following function will break Blum Blum Shub (for small *N*):

def previous\_BBS\_state(state):

r"""

This function returns the previous BlumBlumShub state.

Note that this is a toy function and will only work on small N.

"""

N = state[0];

R = IntegerModRing(N);

X = R(state[1]);

if (not X.is\_square()):

print "Not a valid Blum-Blum-Shub RNG state."

return None

return [N, X.sqrt().lift()];

1. The first part of the problem is to notice that if you have integers *x*,*y* such that *x* ≠ ±*y* (mod *N*) and , then the usual difference of squares equation gives that . And so we can hope that gcd(*x*-*y*,*N*) or gcd(*x*+*y*,*N*) yield a nontrivial factor of N. Write a sage program that takes *x*,*y* such that *x* ≠ ±*y* (mod *N*) and and tries to find a nontrivial factor of *N*.

def FactorByCongruentSquares(x, y, N):

r"""

This function tries to factor N

given integers x,y such that x^2 = y^2 mod N

"""

return (gcd(x+y,N), gcd(x-y,N))

1. Using the function you wrote in part (a) and the supplied function previous\_BBS\_state, write a function that takes a number N (that is a product of two primes *p*,*q* both congruent to 3 mod 4) and factors *p*,*q*. [Hint: you have to create your own BBS state, so you will have to choose your square. How do you choose a square such that you know you have x,y?]

def FactorFromBreakingBlumBlumShub(N):

r"""

This Function Factors a number N by reducing to breaking the BBS RNG

"""

factored = False

factors = []

trivial\_factors = [1,N]

while (not factored):

x = randint(N,N^2)

bbs\_state = [N, (x^2)%N]

y = previous\_BBS\_state(bbs\_state)[1]

(p,q) = FactorByCongruentSquares(x,y,N)

if (p not in trivial\_factors):

factored = True;

factors.append(p)

p2 = N/p

if (p != p2): factors.append(p2)

if (q not in trivial\_factors):

factored = True;

if (q not in factors): factors.append(q)

q2 = N/q

if (q2 not in factors): factors.append(q2);

factors.sort()

return factors