Mathematics 3159A Introduction to Cryptography

Assignment 3

Problem (Part 1)

Solution

For brevity, one line congruences modulo p will occasionally use the notation $a \equiv_p b$.

We seek to perform a modified Miller-Rabin ('MR') test on ed, so begin by decomposing the product such that $ed - 1 = 2^k r$ where r odd. By the congruence defining e and d, we know for some $\ell \in \mathbb{Z}$ it is true that

$$2^k r = ed - 1 = \ell(p-1)(q-1).$$

Combining this with Euler's theorem, for any $a \not\equiv 0 \pmod{p}$,

$$a^{2^k r} \equiv a^{\ell(p-1)(q-1)} \equiv \left(a^{(p-1)(q-1)}\right)^{\ell} \equiv 1 \pmod{p}$$

and likewise for q. Thus,

$$\left(a^{2^{k-1} r}\right)^2 \equiv \pm 1 \pmod{p}$$

and likewise for q, by previous assignments.

Note that this form is identical to the form used in arguing the MR theorem, hence we know we can perform MR on p and q using r, assuming $p \nmid m$ and $q \nmid m$. Let us take integers u, v so $1 \leq u < v \leq k$ and further assume that $N \nmid m^{2^u k}$ for all such u.

We know that we cannot have $m^{ur} \equiv_p \pm 1$ and $m^{ur} \equiv_q \pm 1$ simultaneously as this would imply one of p or q is not prime. Thus, we consider without loss of generality the results of the MR test to only be

$$\begin{cases} m^r \equiv_p 1 & m^{2^v r} \equiv_q -1 \\ m^{2^u r} \equiv_p -1 & m^{2^v r} \equiv_q -1. \end{cases}.$$

In the first case, this implies

$$\gcd(m^r - 1, N) = p.$$

In the second case, we similarly have

$$\gcd(m^{2^u r} + 1, N) = p.$$

From here it is clear how to extract an algorithm - so long as we can somehow find such messages m. We require such an m to satisfy three divisibility conditions: $p \nmid m$ and $q \nmid m$, and also $N \nmid m^{2^u r}$ (for all applicable u). The first two divisibility conditions are simple to assert, as we can take $m > 0$, and then note that if $m < p$ and $m < q$, it will hold. In the event $m \geq p$ or $m \geq q$, if we iterate m by 1 in our algorithm then we can catch $m = p$ or $m = q$, so we can amend our algorithm by checking if $m \mid N$ before doing any computation.

The final divisibility condition means

$$m^{2^u r} \equiv m^{2^k r} \equiv m^{ed-1} \equiv 0 \mod N.$$

However, as $m^{ed} \equiv_N m$, this implies that m is a multiple of p or q, and so this has already been addressed.

Thus, our final algorithm is as follows:

Input: RSA triple (N, e, d).

Output: Primes (p, q) such that $N = pq$.

Algorithm:

1. Find odd r such that $2^k r = ed - 1$. Set $m := 2$.
2. If $2 \mid N$, return $\left(2, \frac{N}{2}\right)$.
   This is a technical condition as the MR test does not work for even primes.
3. Set $m := m + 1$. Set $u := 0$.
4. If $m \mid N$, return $\left(m, \frac{N}{m}\right)$.
5. If $g := \gcd(m^r - 1, N) \neq 1$ and $g \neq N$, return $\left(g, \frac{N}{g}\right)$.
6. If $u = k$, go to 3.
   Else, set $u := u + 1$.
7. If $g := \gcd(m^{2^u r} + 1, N) \neq 1$ and $g \neq N$, return $\left(g, \frac{N}{g}\right)$.
   Else, go to 6.

We know that computation of such an r is just repeated division by 2, and that computing the gcd can be done using the Euclidean algorithm. As we are guaranteed that either m is actually p itself or meets the conditions for the MR test, we know that this algorithm will likely not perform too many iterations. Moreover, we are guaranteed it will halt as eventually, the worst case scenario is we run into $m = p$.

Program (Part 2)

```
### Bradley Assaly-Nesrallah
### bassalyn@uwo.ca
### 250779140
### Written Oct 23, 2020
###
### Usage: solve(N,e,d)

##Implementation of fast powering algorithm
##input:base, power and modulo p integers
##output base^power mod p integer

from generate_input import generate_input

def rapidExponentiation( base, power, p ):

    result = 1 #start with 1
    while power != 0: # loop until power = 0
        if power % 2 == 1: #odd pow -1 exp
            result = (result * base) % p
        power = power // 2 # ^power = ^power/2
        base = (base * base) % p # base = base * base

    return result

##implementation of exteded euclidean algorithm go compute gcd
##input integers a and b
##output gcd of the integers a and b
def eeagcd(a, b):
    x,y, u,v = 0,1, 1,0 ##base case
    while a != 0: ##while a not 0 loop
        q, r = b//a, b%a ##EEA implenetation from textbook
        m, n = x-u*q, y-v*q
        b,a, x,y, u,v = a,r, u,v, m,n
    gcd = b
    return gcd ##eturns gcd

# Based on math proof of part 1
# Input : (N,e,d) N=pq primes, e an encryption exponent,
# d a decryption exponent in RSA
# Output: Returns either p or q in a tuple (p,q) or (q,p) or fails
def solve(N, e, d):
    m=2 ##set variables
```

```
    k=0
    f=e*d-1 ##computes ed-1modN
    while(f%2==0): ##finds k and r dividing f by 2 repeatedly
        f=f//2
        k=k+1
    r=f
    if 2//N: ##checks if 2|N for completeness
        return 2,N//2
    while (m<N): ##for each m value loops
        u=0 ##sets u=0 and increments m
        m = m + 1
        if m//N: ##if m//N then returns p=m,q=N/m
            p=m
            q=N//m
            return p,q
        g= eeagcd(rapidExponentiation(m,r,N)-1,N) ##computes gcd(m^r-1,N)
        if g!=1 and g!=N: ##if gcd!=1 or N then returns p=g,q=N/g
                p=g
                q=N//g
                if p<q:
                    return p,q
                else:
                    return q,p
        while u!=k: ##loops through different exponents u<k
                            ##computes gcd(m^(2u)r-1,N)
            g = eeagcd(rapidExponentiation(m,((2**u)*r)%N,N)+1,N)
            if g!=1 and g!=N: ##if gcd!=1 or N then returns p=g,q=N/g
                p = g
                q = N // g
                if p < q:
                    return p, q
                else:
                    return q, p
            u=u+1 ##increment u

##function to verify that e*dmodphi(pq)=1
##returns True if so, false otherwise
def check(N,e,d):
    p=solve(N,e,d)[0]
    q=solve(N,e,d)[1]
    phi=(p-1)*(q-1) ##returns bool true if check holds otherwise false
    if(d*e)%phi==1:
        return True
    return False
```

```
##main function, solves for all tuples obtained from generate input.py
if __name__ == "__main__":
    input_tuples = generate_input("140")
    for tuple in input_tuples: ##solves and outputs for each tuple
        print(
            'solve({0}, {1}, {2})'.format(tuple[0], tuple[1], tuple[2]),
            solve(tuple[0], tuple[1], tuple[2]),
            check(tuple[0], tuple[1], tuple[2]),
            sep='\n\t\t=',
            end='\n\n'
        )
```

Generated Inputs:
Generated using "140"

- (105080927816417904 9079, 23369701885629628111, 524478494407057810591)
- (10508093502577841027 03, 933042691175928717067, 955285026411151420243)
- (10508092889911864544 93, 984278837819278329659, 532073520485459877299)
- (10508093007906790869 97, 850084711918267324861, 694659902314249934061)
- (1050809249119274879159, 579405103944194398163, 853562181014976162467)
- (1050809285684735342471, 45091654693497220531, 253934618127804659371)
- (1050809314016483952913, 51812811698794312703, 501157746939984269439)
- (10508093072739171792 77, 1046607689162852135761, 8696889781897814442 61)
- (1050809322833687783537, 394344035858555243861, 333433239980204588861)
- (1050809241209725117819, 667360188200986404907, 973792218467526147043)

Generated Outputs:
Corresponding to the input order above.

- (32416187659, 32416189381)
- (32416189193, 32416190071)
- (32416188113, 32416189261)
- (32416188227, 32416189511)
- (32416187627, 32416188517)
- (32416188191, 32416189081)
- (32416188269, 32416189877)
- (32416188647, 32416189291)
- (32416188697, 32416189721)
- (32416187659, 32416188241)