Homework 4

April 2, 2022

You will need 50% of all homework points to qualify for the exam.

You may hand in your solutions in person or by email to kristiinessaarmann@gmail.com. If you submit by email, either scan a handwritten solution or typeset your solution readably. I do not consider ASCII formulas readable.

When submitting, indicate your name and your matriculation number.

The total number of points for each homework is 20 (not including points for bonus problems, if available).

For submitting your solution in a nicely typeset way (e.g., using LaTeX), you get up to 3 bonus points, but not more than 30% of the points you reached for content.

Problem 1: RSA with $N$ reuse

We note that in RSA, we use the values $d, e$ and $N$ for encrypting and decrypting but not the factors of $N$ — thus, in essence, we could devise a scheme where some central authority picks the values $p$ and $q$, computes $N = pq$ and $\varphi(N) = (p - 1)(q - 1)$, and uses those to randomly sample $e_1, e_2, \ldots, e_k$ and obtain pairs of keys $(d_1, e_1), (d_2, e_2), \ldots, (d_k, e_k)$. Then the authority could distribute the $d_1$ to party $P_1$, $d_2$ to $P_2$ etc and publish the $e_i$ as the public keys. Then each party $P_i$ could use $d_i$ as their secret key. In addition, because everyone is operating on the same algebraic structure — integers modulo $N$ — there would be gains in efficiency.

This scenario has been described as a Python program below that has been also uploaded as shared-n-rsa.py to the course webpage. Your task is to write an adversary adv1 in that program, that, given two pairs of keys $(d_1, e_1)$ and $(d_2, e_2)$ as well as the public parameters, is able to factor $N$ with a reasonable probability. This describes the scenario where two parties collaborate and are able to obtain the secret keys of other parties.

Your adversary does not have to succeed every time but should at least be able to occasionally succeed and not take too much time. You can pick how many tests you want to run, if you obtain at least one success with the total running time of a few minutes, this will be considered a valid solution. Use the function testSharedNRSA for testing, however, do use 50 for the value of bits in testSharedNRSA(bits, times, adv).

If you are not very comfortable with Python, you can obtain 75% of the points by simply providing pseudocode for the adversary and give intuition on why the strategy
should be successful if the adversary is lucky enough. ('Lucky' is left a bit vague in this problem. You don’t have to give the precise probability of a successful attack. Describing an attack and arguing that there is a realistic chance of it happening is enough. One per cent chance is considered a realistic attack. Chance with probability $2^{-40}$ is not.) However, I do suggest trying to write the code.

Hint: What does $e_1 d_1 = 1 \pmod{\varphi(N)}$ mean over integers?

Hint: Roughly speaking, the probability that two randomly picked integers are coprime is $\frac{6}{\pi^2} \approx 61\%$. While you may be dealing with integers that are not, strictly speaking, picked randomly, don’t worry about it, you either only have to give intuition about it or write code and see if it works.

Here is a template for your solution. You need to fill in code where there are ???.

```python
import random, math, time
from Crypto.Util.number import getPrime

#"Crypto" might need "pip install pycryptodome" if it's not installed

given integers a, b, returns integers g, x,y such that g is the gcd of a and b and g=x*a+y*b
#from https://stackoverflow.com/questions/4798654/modular-multiplicative-inverse-function

def egcd(a, b):
    if a == 0:
        return (b, 0, 1)
    else:
        g, y, x = egcd(b % a, a)
        return (g, x - (b // a) * y, y)

def modinv(a, m):
    g, x, y = egcd(a, m)
    if g != 1:
        raise Exception('modular inverse does not exist')
    else:
        return x % m

# Generate the values N, phi(N), p, q
def generateGeneralKeys(bits):
    p=getPrime(bits)
    q=getPrime(bits)
    N=p*q
    phiN=(p-1)*(q-1)
    return (N,phiN,p,q)
```

2
#this will output a pair (d,e) such that e is coprime with phiN and d*e=1 mod phi(N)
def generatePartyKeys(N,phiN,bits):
    noKeyYet=True
    while(noKeyYet):
        e= random.randint(0,2**(2*bits))
        if(math.gcd(e,phiN)==1):
            d=modinv(e,phiN)
            noKeyYet=False
    return (e,d)

#this function gets phi(N)=(p-1)*(q-1) and N=p*q as inputs and outputs p and q
def fromPhiNToFactors(phiN,N):
    pPlusQ=N-phiN+1 #from p*q and (p-1)*(q-1) easy to compute p+q
    p= (pPlusQ + math.sqrt(pPlusQ**2-4*N))/2 #from p+q and p*q easy to compute p and q
    q= (pPlusQ - math.sqrt(pPlusQ**2-4*N))/2
    return (p,q)

#this tests whether you have found the correct factors.
def testSolution(N,p,q):
    if(p==1 or q==1):
        return 0
    if(int(p)*int(q)==N):
        return 1
    else:
        return 0

#it is your job to define this adversary
def adv1(bits, N, e1, d1, e2,d2):
    ???
    return (p,q)

#this is the game
def oneGame(bits, adv):
    (N,phiN,p,q)=generateGeneralKeys(bits)
    (e1,d1)=generatePartyKeys(N,phiN,bits)
    (e2,d2)=generatePartyKeys(N,phiN,bits)
    (p,q)=adv(bits, N, e1, d1, e2,d2)
    isSuccessful=testSolution(N,p,q)
    return isSuccessful

#use this to test your solution
def testSharedNRSA(bits,times,adv):
    start_time = time.time()
count=0
for i in range(times):
    count+=oneGame(bits,adv)
ratio= (1.0*count)/times
print("ratio=", ratio)
print("number of successes=", count)
print("--- %s seconds ---" % (time.time() - start_time))

testSharedNRSA(50,100, adv1)

Note: It is in fact possible to factor \(N\) knowing only one pair \((d_1, e_1)\), however, this is more complicated. If you want, you can of course write Python code for that. If you want to submit a pseudocode solution however, merely saying "we use the commonly known algorithm for factoring \(N\) based on a pair \((d_1, e_1)\) such that \(d_1e_1 = 1 \pmod{\varphi(N)}\)" will not give you points. If you, in the pseudocode, want to use some kind of algorithm not provided in the Python script, you should describe that algorithm in detail.

Solution

We know that \(d_1e_1 = 1 \pmod{\varphi(N)}\) and \(d_2e_2 = 1 \pmod{\varphi(N)}\). This means that there exist integers \(k_1\) and \(k_2\) such that \(d_1e_1 = k_1\varphi(N) + 1\) and \(d_2e_2 = k_2\varphi(N) + 1\) over the integers. Thus, if \(k_1\) and \(k_2\) happen to be coprime, we have that the greatest common divisor of \(d_1e_1 - 1\) and \(d_2e_2 - 1\) is \(\varphi(N)\), in that case running the GCD algorithm on \(d_1e_1 - 1\) and \(d_2e_2 - 1\) will give us \(\varphi(N)\). From \(\varphi(N)\) and \(N\) we can obtain the factors of \(N\). Although \(k_1\) and \(k_2\) are not randomly picked, the chance that they are coprime seems in practice to be quite high, giving us a practical solution.

import random, math, time
from Crypto.Util.number import getPrime

#"Crypto" might need "pip install pycryptodome" if it’s not installed

given integers a, b, returns integers g, x,y such that g is the gcd of a and b and g=x*a+y*b
#from https://stackoverflow.com/questions/4798654/modular-multiplicative-inverse-function
def egcd(a, b):
    if a == 0:
        return (b, 0, 1)
    else:
        g, y, x = egcd(b % a, a)
return (g, x - (b // a) * y, y)

#returns the modular inverse of a modulo m.
#from https://stackoverflow.com/questions/4798654/modular-multiplicative-inverse-function
def modinv(a, m):
g, x, y = egcd(a, m)
if g != 1:
    raise Exception('modular inverse does not exist')
else:
    return x % m

# Generate the values N, phi(N), p, q
def generateGeneralKeys(bits):
p=getPrime(bits)
q=getPrime(bits)
N=p*q
phiN=(p-1)*(q-1)
return (N,phiN,p,q)

#this will output a pair (d,e) such that e is coprime with phiN and d*e=1 mod phi(N)
def generatePartyKeys(N,phiN,bits):
oKeyYet=True
while(noKeyYet):
e= random.randint(0,2**(2*bits))
if(math.gcd(e,phiN)==1):
d=modinv(e,phiN)
oKeyYet=False
return (e,d)

#this function gets phi(N)=(p-1)*(q-1) and N=p*q as inputs and outputs p and q
def fromPhiNToFactors(phiN,N):
pPlusQ=N-phiN+1 #from p*q and (p-1)*(q-1) easy to compute p+q
p= (pPlusQ + math.sqrt(pPlusQ**2-4*N))/2 #from p+q and p*q easy to compute p and q
q= (pPlusQ - math.sqrt(pPlusQ**2-4*N))/2
return (p,q)

#this tests whether you have found the correct factors.
def testSolution(N,p,q):
    if(p==1 or q==1):
        return 0
    if(int(p)*int(q)==N):
        return 1
    return
else:
    return 0

# it is your job to define this adversary
def adv1(bits, N, e1, d1, e2, d2):
    phiNCandidate = math.gcd(e1*d1-1, e2*d2-1)
    (p, q) = fromPhiNToFactors(phiNCandidate, N)
    return (p, q)

# this is the game
def oneGame(bits, adv):
    (N, phiN, p, q) = generateGeneralKeys(bits)
    (e1, d1) = generatePartyKeys(N, phiN, bits)
    (e2, d2) = generatePartyKeys(N, phiN, bits)
    (p, q) = adv(bits, N, e1, d1, e2, d2)
    isSuccessful = testSolution(N, p, q)
    return isSuccessful

# use this to test your solution
def testSharedNRSA(bits, times, adv):
    start_time = time.time()
    count = 0
    for i in range(times):
        count += oneGame(bits, adv)
    ratio = (1.0*count) / times
    print("ratio=", ratio)
    print("number of successes=", count)
    print("--- %s seconds ---" % (time.time() - start_time))

testSharedNRSA(50, 100, adv1)

Problem 2: An unsavory group

Recall that ElGamal can be defined with respect to many different groups. Here we give an example of a group one should not use.

Let $p > 0$ be a large prime. Let $G := \{0, \ldots, p-1\}$. The group operation is defined as follows: For $a, b \in G$, let $a \cdot b := (a + b \mod p)$\footnote{This is notationally highly confusing, of course, because it looks like we claim that plus and times...} Recall that $a^i$ for $i \in \mathbb{N}$ is defined as...
\[ a^i := a \cdot a \cdot a \cdot a \cdots a \ (i\text{-times}). \]

(a) What is \( a^i \) written in terms of +? (I.e., when unfolding the definition of the group operation, and using modular arithmetic.) This should be quite a simple operation!

Solution.

\[ a^i = a + a + \cdots + a \mod p = ia \mod p \]

(Note that \( ia \) here is normal multiplication.)

(b) Show that there is an efficient algorithm for solving the discrete logarithm problem in \( G \). That is, given \( a \in G \) and \( b := a^i \in G \) (but not given \( i \)), the algorithm should compute \( i \).

Note: You can use, without proof, the fact that there is an efficient algorithm (Extended Euclidean Algorithm, EEA) that, given \( p \) and \( x \in \{1, \ldots, p-1\} \) computes \( y \) with \( xy \equiv 1 \mod p \). (This is multiplication modulo \( p \), not the group operation. The fact that inverting works for all \( x \) uses that \( p \) is prime.)

Solution.

- Input: \( a, b \in G \).
- \( c := \text{invert}(p,a) \) (gives \( c \) s.t. \( ca \equiv 1 \mod p \)).
- \( i := bc \mod p \) (then \( ia \equiv bca \equiv b \mod p \), so \( a^i = b \)).
- Return \( i \).

(c) Show that the DDH assumption does not hold for \( G \). (I.e., there is an efficient algorithm that distinguishes the two games from the definition of the DDH assumption with probability close to 1.)

Solution.

- Input: \( g, a, b, c \) (where \( a, b, c \) are either \( g^x, g^y, g^{xy} \) or \( g^x, g^y, g^z \)).
- \( x := \text{dlog}(g,a) \) (here \( \text{dlog} \) is the algorithm from (b)).

are the same thing. But keep in mind that we are defining a new operation on \( G \) here, and it is just a notational convention that we write it like multiplication. In particular, do not confuse \( a \cdot b \) (the group operation) with \( a \cdot b \mod p \) (actual multiplication modulo \( p \)). Often one would use the symbol +, but that would be confusing as well because in our definitions of ElGamal we used multiplicative notation. If you wish, you can introduce a different symbol for the group operation, say \( \circ \), and then the definition becomes \( a \circ b := (a + b \mod p) \). You are free to do it either way in your solution, but make sure that you do not mix up the different meanings of \( a \cdot b \)!
• $y := \text{dlog}(g, b)$.
• Return true if $c = g^y$. (Otherwise, return false.)