Lab #4: RSA and DH

Student ID ____________________ Section # ________

Marks ____________________

Introduction (Now for something from applied number theory)

This lab has two objectives. The first, that of becoming familiar with the basics of public key encryption focusing on the RSA algorithm. After becoming familiar with the specification you will implement RSA. Using this implementation we will open a file that is not encrypted and save it in its encrypted form. This form would be suitable for transmission over a public network or just hiding stuff on your computer ☺. The remainder of the implementation opens an encrypted file and decrypts it. This is not exactly how RSA would be used but illustrates some difficulties with encrypting and decrypting a file. The second is to become familiar with the workings of Diffie-Hellman key exchange.

If you are writing this lab in Java familiarize yourself with the BigInteger class library, in particular be sure that you know exactly what each of the constructors can be used for. The BigInteger class allows the programmer to utilize numbers of arbitrary length unrestricted by the underlying machine architecture. This is very important for algorithms such as RSA as the numbers used are typically very, very large. Python has similar as do many languages. For sure, both Python and Java have complete OTS packages for RSA.

RSA

RSA is a public key encryption scheme in which each user has two sets of keys, a private and a public set, which together can be used to encrypt and decrypt a message. In general the security of public key encryption works on the basis of one way functions which are easy to compute, but difficult to reverse (without some special knowledge). They are typically called hash functions as they represent general class of functions that are easy to compute but where the inverse mapping is difficult. For many hash functions it is impossible. That is, more traditionally a hash may be a compact identification or something larger where it is impossible to do the inverse mapping. It is also known as hash function because it is commonly what is smoked by cryptographists when they come up with this stuff. There is a similar reasoning to those who attempt to crack encrypted files.

Specifically RSA utilizes a one way hash with a modulus value formed by the multiplication of two large prime numbers (p and q). The one way nature is due to the fact that it is easy to multiply two big integers but it is difficult to factor a large integer. These same two special numbers (p and q) are used to calculate the private key and public keys. The special knowledge here are the two large prime numbers (p and q), where pq=n, without which it is necessary to factor the modulus n. Thus the security of RSA is based on the difficulty of factoring very large numbers. The difference in complexity between multiplying and factoring is what RSA is based on. Other encryption techniques may be based on operations such as mod exponentiation and inverse mod (discrete logs). Same idea, easy to exponentiate difficult to take a log. In the case of normal log functions one could as what is lg(100). You could guess and say it might be 6, then check 2^6=64 so that is too low a guess you could guess 7, 2^7=128, so the answer is less than 7. That is you could home in on the answer. Assume you were asked to find a=$\text{dlog}_3 13$ mod 17. Guess a=5, check 3^5mod 17 =5 so you might think the answer is greater than 5 and try 6. Check 3^6mod 17 =15, really no help at all. That is why it is called a one way function.

Back to RSA:

For example, if Bob wishes to send a message to Alice:

• First Alice would need to generate a set of public and private keys:

i. Alice generates two large prime numbers p and q (typically 100 digits each)
ii. Alice creates a new large number, the modulus n:
 n = p x q
iii. Alice chooses a public exponent, **e**, such that e is relatively prime to:
 PHI(n) = (p-1)(q-1)
iv. Alice then calculates a private exponent, d, such that:
 e d mod PHI(n) = 1

i.e. d is calculated to be the multiplicative inverse of e mod PHI(n).

• Next Alice sends the public key to Bob (and anyone else listening). The public key consists of the public exponent e and the modulus n.

• Now Bob can encrypt a message M into Ciphertext C for Alice:

• Bob sends the encrypted text, C, to Alice (which again anyone can see).

• Finally Alice, and only Alice, can decrypt the ciphertext C with her private key:

$C = M^e$ mod n.

$M = C^d$ mod n.

RSA in Java

The BigInteger class in Java has all of the methods/algorithms necessary to generate keys and perform encryption and decryption using RSA.

(1) The program Rsa.java provides a working example that implements the RSA algorithm, sort of. It is a very good example of very poorly written code. It is just hack and slash to see how the BigInteger functions work.

• Enter, compile and run the program Rsa.java.

• This simply illustrates encryption and decryption of an integer.

(2) Using the example Rsa.java program as the basis open a file and encrypt it. Using the same program or similar decrypt the encrypted file.

The program should contain or have access to both a private and a public RSAkey, appropriate constructors, as well as methods to encrypt and decrypt a text string (String).

It is realtively easy to use RSA to encrypt a small message (i.e. less than n in size), it is more difficult to encrypt a larger file because RSA is in effect a block cypher. Try not to encrypt one character at a time as that would work but be very inefficient.

• Do not forget the relationship between the string size that can be encoded and the modulus, n. To be safe ensure that any text string is divided into segments no larger than the key size before encryption. This implies that the encryption method should return an array of BigIntegers.

• Consequently, decryption should accept an array of BigIntegers and rebuild the string if it was segmented.

• If too difficult, start by assuming the message is always smaller than ‘n’. This can be modified later, time permitting.

(3) Write a simple program to test the code, name this program RSAtest.java.

• In your program, measure the time (use the java.Util.Date class) required to calculate keys, and to encrypt, and decrypt messages for different size modulus: e.g. n=64, 128, 256, 512, 1024, and 2048 bits, with e=O(1) and d=O(PHI(n)).

Questions:

1) What is the distribution of prime numbers among all numbers? This is a function of n.

2) How many numbers on average are selected/tested for primality using the BigInteger

constructor before one is likely found? This is a function of n and s. This is not as accurate as you may think, more on that in the next lab.

3) What are the basic number theory algorithms used in RSA?

4) How can RSA be used within private key crypto systems?

5) Does RSA require prime numbers or numbers that are very likely prime?

6) Late-ish breaking news: When was the problem of generating a larger prime number shown to be in P.

The following may also be of some assistance.

http://javadigest.wordpress.com/2012/08/26/rsa-encryption-example/

The Diffie-Hellman Part:

Try to set up all the required functions to implement Diffie-Hellman key exchange. You might not yet know why each of the following steps are required by you can certainly follow the implementation. Also it may be possible to implement DH without some of the following but this reflects how it should be done (RFC 2409, 2631, 5114 etc.).

- Generate a big prime **q** using the BigInteger constructor.
- Generate **2q+1 = p**. This is called a safe prime. Test if p is prime. i.e. test a^{p-1}modp for several values of a. Eventually p will also be declared prime to the degree you are happy with. (**If not start over with new prime q**). You can generate q with the BigInteger constructor.
- **Finding a generator.** Pick g=2 as a possible generator. Test if 2^qmodp= 1, if it is then 2 is not a generator. (Wraps around too soon, so pick another g). Pick g=3 as a possible generator. Test if 3^qmodp= 1, if it is not then 3 is a generator. Do this until you find a generator g.
- Therefor the g that passes the above, is a generator of Z^*_p. $\Phi(p)=p-1$.
- That was not obvious but not painful. (Relied on property of safe primes, if wrapping occurs, we know where it will occur (p-1)=2q)
- Now assume you are Bob and Alice and have agreed on p and g. Generate secrets for both Bob and Alice (b and a). Calculate g^bmodp=B as if you were Bob and g^amodp=A as if you were Alice. Assume these are exchanged. Calculate K=A^bmodp as if you were Bob and B^amodp as if you were Alice. Those should be the same.
- How would either Alice or Bob transmit a shared secret such as a specific key such as 0101010101 between each other?

Demonstrate all of the above in your write-up.

DH Question(s):

What are RFCs?

Repeat the above without p being a safe prime but still prime, did it work? My guess is that it did, at least for the instance you tried.

Repeat the above without p being a prime but at all, did it work? My guess is that it may have, at least for the instance you tried.

Speculate as to why safe primes are selected as well as the test for a generator which generates Z^*_p.

In any event, real implementations likely use well vetted ps and gs.

Report

Report Submission Instructions The lab report is due at the start of the next lab, and must be submitted electronically via UMLearn.

• Each student is required to write and submit a separate report.

• In the body of the lab report include answers for each of the questions posed in the lab.

• Write a well-organized lab report. If uncertain, then ask a friend.

• Document all Java Programs with comments in the program code.

• Lab write ups will only be accepted in PDF format.

• Programs should be in plain text.

• Lab write-up and required programs, sent as attachments. Zip up everything if you like.