Institut for Matematik og Datalogi
Syddansk Universitet, Odense

30. november 2019
JFB/RF

Lab DM534 Uge 49

Meet in IMADA's Computer Lab. Work in groups of size 2 (maybe some of size 3).

The best known public key cryptographic system, RSA, was presented in lectures. It is one of the systems included in encryption toolbox PGP and GPG. Its security is based on the assumption that factoring large integers is hard. (The system you will be using later today in GPG is based on discrete logarithms, rather than factoring as in RSA, but the problems are similar in many ways. Factoring is easier to understand and test in Maple, which we do in the first part.)

In RSA, a user's public key consists of a large integer n (currently numbers with 2048 bits are recommended) and an exponent e. The integer n should be a product of two prime numbers p and q, both of which should be about half as long as n. Thus, in order to implement the system it must be possible to find two large primes and multiply them together in a reasonable amount of time. For the security of the system, it must be the case that no one who does not know p or q could factor n.

At first glance this seems strange, that one should be able to determine if a number is prime or not, but not be able to factor it. However, there are algorithms, such as the Rabin-Miller algorithm discussed in lectures, for testing primality, which can discover that a number is composite (not prime) without finding any of its factors. (The algorithms most commonly used are probabilistic, so they could with small probability declare a composite number prime; the probability of this happening can be made arbitrarily small.)

Primes and factoring

Using the symbolic algebra program Maple, you will in the following try producing primes and composites and try factoring.

1. Small numbers.

   Start your Maple program, using the command `/opt/maple2019/bin/xmaple &` . Ignore any message boxes on updates or license issues (click "no" or similar). Click on the icon for opening a new worksheet. Type `restart;` at the beginning to make it easier to execute your worksheet after you have made changes. (You can execute the worksheet after changes from `Execute` in the `Edit` menu.)

   Use *Help* (choose *Maple Help*) to find out about the function *ithprime* (just close the help window afterwards). Experiment to find out approximately how big a prime it can find. When it cannot find such a big prime (in a reasonable amount of time), you can use the STOP button on the toolbar in order to continue (it is an exclamation mark crossed over in red). To assign a value to a variable, you use the assignment operator `:=`; for example `x:= ithprime(4);`. Multiply two of the large primes it finds together, and try to factor the result, using the function *ifactor*. Notice how quickly the factors are found for these small numbers. (Large numbers are clearly necessary for security.)

2. Finding larger primes.

   In order to find good prime factors p and q for use in RSA, one can choose random numbers of the required length and check each one for primality until finding a prime.

   Maple contains a function *isprime* which will test for primality. Try it on some some small numbers, such as 3, 4, 7, 10. Maple has another function *rand* which returns a random 12-digit number. Try typing `x:=rand();` and check if your result is prime. In order to execute these commands until you find a prime, you can use a *while loop*. To continue creating new random numbers until you get a prime, you can type `while (not isprime(x)) do`, followed by `x:=rand();` and `end do;`. (Note that `x` should be assigned some composite number before this, so that the loop is executed at least once.) If you write the loop on more than one line, you should get to the next line using `SHIFT ENTER` instead of just `ENTER`.

   How many different values were chosen before a prime was found? (I got 43, but you could get another number.) Now create a second prime called y (remember that y will need some value before your start your *while loop*). Multiply x and y together and try factoring the result. This should also go relatively quickly.

3. Finding even larger primes.

   To get random numbers which are three times as long, you can create three random numbers a, b and c and create $10^{24} * a + 10^{12} * b + c$ (10 raised to the power 24 times a, plus 10 raised to the power 12 time b, plus c). Unfortunately, two calls to *rand* in the same statement will give the same result both times, so you need to choose values for a and b independently and then combine them. (Suppose you typed `m1 := 10^12*rand() + rand();`. Why wouldn't you ever find a prime testing values found this way?) Try finding two primes, each 36 digits long. The first can be found by starting with `m1:=4;` so you start out with a composite. Then use the following:

```
while (not isprime(m1)) do
a := rand();
b := rand();
c := rand();
m1 := 10^24 * a + 10^12 * b + c;
end do;.
```

   Multiply the two primes together and try to factor the result. If your machine is not fast enough, use the STOP button on the toolbar after a few minutes; the computation takes too long. Otherwise, create even longer primes and try factoring them. As you might imagine, no known algorithm would factor a 1024-bit (about 300 digits) number on your PC in your lifetime. It is easy to find the primes and multiply them together, but it is very difficult to factor the result! (If this was not the case, RSA would not be secure.)

   Try to get a feeling for how long it takes to factor numbers of different lengths; you can try changing the 10^{12} or 10^{24} in your *while loops* to larger or smaller values.

4. Find the multiplicative inverse of 25 modulo 43 (a number between 0 and 42, which when multiplied by 25 gives the result 1 modulo 43). Note that you can find out about the function for computing the Extended Euclidean Algorithm by typing `?igcdex`.

5. (You may skip this problem for now and come back to it later if time is short.) Switch your worksheet to *Text* mode instead of *Math* mode, just above your worksheet. Then to exponentiate modulo 1083, try typing

```
3&^10293846675 mod 1083;
```

Next, try removing the `&` from this statement. [Recall that there is a STOP button.] Also try `3^5 mod 1083;` Try to figure out why there is a difference with the large exponent.

Encryption

Now, we leave Maple and instead switch to the program `gpg` to try encryption. Start up a command line terminal window. Usage information can be obtained by typing `gpg -h | less` (the vertical line says pipe the output through the next program, and `less` shows a page at a time). Note that this also tells you what types of encryption can be chosen. You can find directions on how to use `gpg` on this tutorial: `https://www.linuxbabe.com/security/a-practical-guide-to-gpg-part-1-generate-your-keypair`. If you are in doubt in the following exercises, you may want to browse part 1 and part 2 of that tutorial (note that you will be mailing keys and messages to other persons, instead of transfering files between machines using `scp` as done in the tutorial).

1. Create a public and private key using `gpg --full-gen-key`. You should choose DSA and El Gamal, and size 2048. A passphrase is similar to a password and should be chosen to be secure.

2. Go to the directory `.gnupg` using `cd .gnupg`. List what is in the directory using `ls -al`. Try the commands `gpg --list-keys` and `gpg --fingerprint` to list the keys you have, with the fingerprints, which make it easier for you to check that you have the correct key from someone. How would you use fingerprints?

3. You can save your public key in a file in a form that can be seen on a screen using `gpg --armor --export Your Name >filename`. You are "exporting" your key and specifying where the output should go. Then look at it using `less filename`; the `--armor` made it possible to see it reasonably on your screen, since it changes it to ASCII.

4. Mail this file to someone else in the class, sitting close to you.

5. Try to figure out how to use `gpg` to "import" the public key that you yourself got from someone else. (You need to specify the input filename you received from someone else.) Check the fingerprint.

6. Create a little file and sign it. You can use `gpg --clearsign filename`. What does this do? Check the new file you got, looking at it and using `gpg --verify signedfilename`. Now encrypt the signed file. You can use `gpg --armor --encrypt signedfilename`. What does this do?

7. Mail your file to whoever has your public key. Read their file using the command `gpg --decrypt inputfile >outputfile`. Then look at the output file you created and verify the signature as you did above.

8. How could you encrypt a file so only you can read it?