Practical Exercises 1 - RSA

September 7, 2022

The goal of these exercises is to familiarize with RSA's internals and to recognize its main weaknesses. For the implementation (Exercise 1), you will use GMP[1], a C/C++ multi-precision library. To implement the attacks (Exercices 2 and 3), it is strongly advised to use `sage`[2], a python-based mathematical toolbox.

You can contact me anytime by mail at `andrea.lesavourey@irisa.fr`.

Some appetizer: Textbook RSA in SageMath

1. Write an encryption procedure.
2. Write an decryption procedure.
3. Given an integer l, write a key generation procedure.
4. Generate some RSA keys of 30 bits and mesure the time taken to encrypt a random message. If it takes too much time, search why.
5. Do the same with keys of 2014 or 2048 bits.

Écrire une procédure SageMath de chiffrement.

Exercise 1: Textbook RSA

Remainder An RSA *public* key consists in a modulus n and an exponent e, where $n = p \times q$ is the product of two large prime numbers. Here, we will consider $e = 2^{16} + 1$ which is the most commonly used value for both security (see Exercise 2) and performance[3].
The corresponding *private* key is the same modulus, with a exponent d such that d is the inverse of e modulo $(p-1)(q-1)$, meaning we have $ed \equiv 1 \bmod (p-1)(q-1)$.

In this exercise, you need to use GMP as a multi-precision library (`libgmp-dev` on debian-based, `gmp-devel` on fedora).

1. Implement a function for RSA key generation, taking the bit size of the modulus as an input.
   (a) Implement a function taking an integer ℓ and randomly generating a prime number of ℓ bits. Use the function `mpz_cryptrand` (file `mpz_rand.c` reproduced at the end). Be careful, `mpz_cryptrand` takes a byte size. You can use `mpz_nextprime` to find a prime number bigger than the input.
   (b) Using the previous function, implement a function generating two primes p and q and computing the modulus n.

[1] `https://gmplib.org/`
[2] `https://www.sagemath.org`
[3] $2^{16} + 1 = (10000000000000001)_2$ which is ideal for most modular exponentiation algorithms.

(c) Using $e = 2^{16} + 1$, compute d. To do so, implement the extended Euclid algorithm[4], and use it on e and $(p-1)(q-1)$.

(d) Write the key generation function, returning a public key, and a private key (you can store them in two data structures for instance).

2. Implement an encryption function, taking a string (the message) and a public key, returning the encrypted message. The message length should be strictly less than the byte size of the modulus (we omit padding here, which should **never** be done in real world application!).

   (a) Implement a modular exponentiation function, taking the integers m, e, n and returning $m^e \bmod n$. This can be done using the appropriate GMP function.

   (b) Implement an encoding function which converts a character string into integer (here we are talking about a `mpz_t`). This conversion can, for instance, represent each character with its ASCII hexadecimal value. For instance, `encode("word") = 0x776F7264 = 66428301924`.
   Hint: take a look at `mpz_import`, as used in `mpz_cryptrand`.

   (c) Implement a decoding function reversing the previous operation.
   Hint: take a look at `mpz_export`.

   (d) Implement the overall encryption function.

3. Implement a decryption function taking an encrypted message (as an integer) and a private key, returning the corresponding message. You can test your encryption scheme by sharing your public key with your classmates, and exchanging various messages.

Exercise 2: Classical attacks

We strongly advise to use `sage`, or at least Python to avoid wasting time on implementation details. For each question, explain the attack concept and how it works.

Use the file `tp1-rsa_material.sage`, containing all necessary data. Function `int_to_ascii` allows you to convert an integer into ASCII characters (useful to recover the message after decryption).

1. A short message has been encrypted using the public key `(n1, e1)`. We have been informed that no padding has been used. Recover the message.

2. You know that `c1` has been encrypted using the public key `(n2_1, e2)`. You also know that the corresponding entity generated the key `(n2_2, e2)`, and is not very careful with the key generation process... Recover the message.

   (a) Think about which elements are given. Which information could you be able to recover ?

   (b) From the recovered element(s), recover the full private key (`sage` has an `inverse_mod` method which can turn out helpful).

3. The same messages have been sent to multiple people. The public key of each receiver has been used to encrypt the message before sending it. Recover the message.

   (a) Note the similarity between the keys.

   (b) The function `crt` in `sage` allows to use the *Chinese Remainder Theorem* (*Théorème des Restes Chinois* in French). The function `CRT_list` do the same with more than two elements.

[4] `https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm`

Exercise 3: Wiener's attack

Let

$$n = 263004885194704826527404387677458597683161772072822725475342 1$$

and

$$e = 6017756679935389768703896403733360404653947478880246420123 5$$

be the parameters of a RSA public key. We will consider here Wiener's attack[5].

1. Write a function allowing you to compute the convergents of a real number.
2. Deduce from this an attack function on a RSA secret key if it is too small.
3. Retrieve the factorisation of n.

[5] `https://en.wikipedia.org/wiki/Wiener's_attack`

Code of mpz_rand.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <gmp.h>


/* use /dev/urandom to generate random number of the given size */
void mpz_cryptrand(mpz_t rop, size_t size) {
  unsigned char* buf = NULL;
  FILE* f = NULL;

  buf = malloc(size*sizeof(unsigned char));
  if(!buf)
    goto err;

  f = fopen("/dev/urandom", "r");
  if (!f)
    goto err;

  fread(buf, size, 1, f);
  mpz_import(rop, size, 1, 1, 0, 0, buf);

 err:
  if (buf) free(buf);
  if (f) fclose(f);
}

 /* compile with: */
 /* $ gcc -o gmp_rand mpz_rand.c -lgmp */
```