Project 2

Diogo Miguel de Almeida Silveira 85117

February 2, 2022

Contents

Abstract

This report describes the development of the D-RSA key generation method as required for project 2.

It begins with a brief introduction followed by a specification of the software implemented: the core cipher module and its applications for demonstration.

1 Introduction

Rivest-Shamir-Adleman (RSA) is widely used as a standard cryptosystem for asymmetric encryption. It is a very useful method both for symmetric key distribution and digital signature of documents. It begins by taking two random big prime numbers and using them to compute the private and public keys. The public key is then published while the private key is kept secret.

This project aims to demonstrate an implementation of the RSA key generation without the need for saving (and protecting) the private key. The security will be achieved by the time it takes to compute the key, more specifically, the time it takes to generate the prime numbers.

In order to achieve this, two core modules will be built:

- A first one for generating a pseudo-random stream of bytes;
- A second one which takes a stream of bytes as input and extracts from it the two prime numbers which are used to compute the key-pair.

These modules will then be used by two applications:

- *randgen*: will operate the byte stream generator, both for producing a stream as output and for testing the time taken to generate the stream given different input values;
- *rsagen*: will operate the key pair generator, by taking a stream of bytes as input and extracting from it the values needed to compute the pair, storing it in appropriate files afterwards;

Every piece of software in this project will be developed using Python 3.8.10.

2 Byte Stream Generator

The first core module of this project is the pseudo-random byte generator. It produces a deterministic stream of bytes from three parameters: the password, the confusion string and the number of iterations. The last two parameters have direct influence on the time taken to produce the stream (and its length). This module is composed of four sub-modules, which are explained below.

2.1 Generator

This sub-module can be seen as the top of the byte stream generator. It handles the raw input from the user, processing it and using it to initialize the generation of bytes.

All three parameters (password, confusion string and number of iterations) are given as input to a PBKDF2[1] method, implemented by Python's *cryptography* library, to generate a 64-byte bootstrap seed which is used to initiate the generation itself. The confusion string is used as salt and the number of iterations is also given as argument to the digest. Finally, the hash is used to digest the password, returning the bootstrap seed as result.

Furthermore, the confusion string is processed to produce the pattern of bytes which will have to be found in each round of the byte generator. The string is digested by SHAKE256 hash method, implemented by Python's *cryptography* library. The resulting byte sequence has the same length as the string given as input.

This sub-module can handle the textual input parameters (password and confusion string) both in the form of a string or an array of bytes.

2.2 MainGen

It performs the generation of bytes as many times as defined by the number of iterations input parameter. Each time, it generates a segment of the stream and adds it to the result.

For each round, it extracts an initialization vector (IV) and a key from the seed, respectively by extracting the bytes with odd and even indices. Those values, along with the previously computed byte sequence, are then used to initiate (or re-initiate) the underlying sub-module (*StreamGen*) which produces the new stream segment and a new seed to be used in the next iteration.

2.3 StreamGen

This sub-module uses the underlying one, giving it a key and an IV as input, to produce a sequence of bytes, one byte at a time. The sequence ends when the previously computed pattern is found. This is achieved by deleting a byte from the pattern each time it comes up, and stopping when there are no bytes left in the pattern. When this happens, the output sequence is finished but the generation goes on, so that a new seed is obtained to be used in the next round.

2.4 ByteGen

This is the lowest-level sub-module of the generator. It takes the key and the IV as input and uses Advanced Encryption Standard (AES) in OFB mode in order to generate a stream cipher. It actually uses the AES implementation of Python's *cryptography* library in ECB mode, which is turned to OFB mode by being used to encrypt the IV and giving it as input to the next round of encryption[2]:

```
...
# Cipher mode ECB:
cipher = Cipher(algorithms.AES(self.key), modes.ECB(), backend=default_backend())
self.cipher = cipher.encryptor()
...
def __get_block(self):
    return self.cipher.update(self.iv)
...
# Cipher mode ends up being OFB:
if self.block == b'':
    self.block = self.__get_block()
    self.iv = self.block
...
```

This ends up being in fact an implementation of AES in OFB mode without the actual encryption stage.

Every time the generator is called, the leftmost byte is extracted from the currently stored block. When the current block gets empty a new round of the cipher is executed to obtain the next one.

3 Key Pair Generator

This is the second core module of the project. As already mentioned, it takes a stream of bytes as input and then extracts from it two large prime numbers which it uses to compute the new key pair. It is composed of two sub-modules, which are explained below.

3.1 PrimeGen

This sub-module takes as input a sequence of bytes and extracts the two large primes (p and q) from it. This is done by the following process:

1. A first number is computed by picking 128 bytes from the sequence, starting with index 0 and proceeding with a step equal to the integer division of the length of the sequence by 128;
2. A second number is computed by picking bytes the same way but starting k bytes in advance, k being equal to the remaining of the previously performed integer division;
3. Each of the new numbers is tested for primality and if the test fails it is incremented by one and tested again until a prime number is found. The primality test implemented is the Miller-Rabin [3] test, with the recommended 40 rounds[4].

3.2 KeyGen

This sub-module inherits from the previous one. It adds the remaining computations needed to generate the RSA key pair [5]:

- Calculate n by multiplying p by q;
- Calculate the Euler Totient function of n [5]
- Perform the extended euclidean algorithm to obtain the value of d[6] [7];

The value of e is 65537, as required.

The public and private keys are returned in Privacy-Enhanced Mail (PEM) format, ready to be saved in this type of file. This is achieved by using Python's *PyCrypto* library.

4 Applications

Two applications were developed, as required, to demonstrate the implementation of the above modules: *randgen* and *rsagen*.

4.1 Randgen

This application uses the byte generator. In its regular mode of operation it takes the three necessary arguments from the command-line and outputs the stream to *stdout*, in base64 format.

It also has the option of testing the time taken to create the byte sequence, depending on the confusion string and the number of operations. This mode can be executed by calling the application without any command-line arguments. It will output a chart portraying the results obtained for different numbers of rounds and confusion string lengths.

Below is an example of the command used to call the application this way and an example of the output:

```
python randgen.py
```

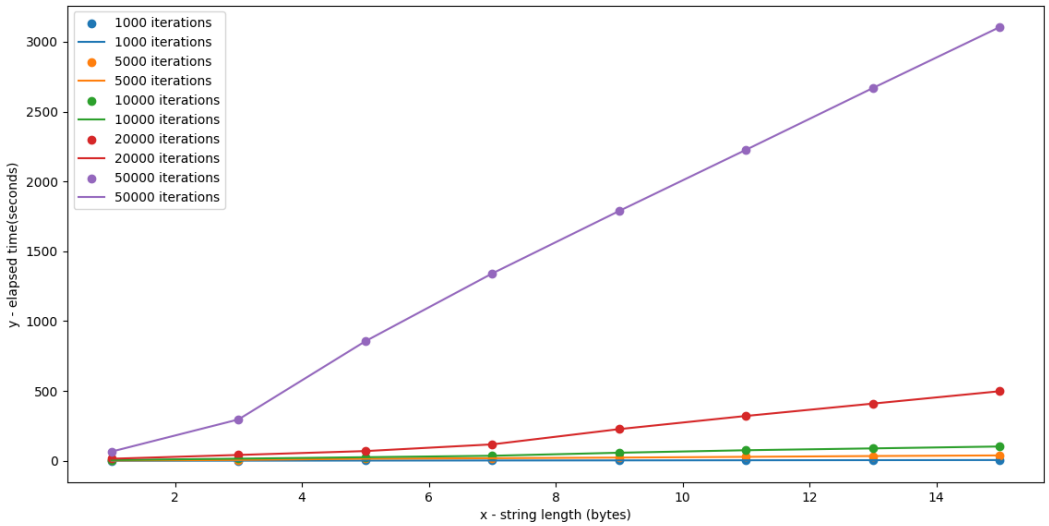

Figure 1: Results obtained by executing randgen in testing mode

The longer it takes to generate the stream, the more resilient the system will be to dictionary, brute-force or random table attacks, or similar ones.

The parameters used for the tests are as follows:

- A single, random password, used in all tests;
- An array of random confusion strings with lengths ranging from 1 to 15, with a step of 2, each one used for every number of iterations;
- A predefined array with values for the number of iterations.

The results obtained in the tests show that 50000 rounds is relatively secure, even with short confusion strings (2 or 3 bytes would already be enough to take minutes to generate a stream of bytes). 20000 rounds would also be safe, requiring strings with at least 5-6 characters. The minimum value advisable for the iterations would be 10000, requiring long strings, with no less than 10 characters. This would ensure at least one minute to create the streams in my laptop. More powerful machines would require larger strings and numbers of rounds.

4.2 Rsagen

This second application takes as input a byte-array in base64 format, from stdin. Then it converts it to raw bytes and uses the key generation module to extract the big prime numbers from it and create the key pair. The result is saved into two .pem files. The name of this files can be given as argument to the application, but default ones are used if no names are defined. The two applications developed can be piped in the command-line, which allows *randgen* to input the stream of bytes directly to *rsagen*. Bellow is an example of the command:

```
python randgen.py password conf 10000 | python rsagen.py prv.pem pub.pem
```

References

[1] Unknown, "Pbkdf2." "https://en.wikipedia.org/wiki/Pbkdf2".

[2] unknown, "The difference in five modes in the aes encryption algorithm." "https://www.highgo.ca/2019/08/08/the-difference-in-five-modes-in-the-aes-encryption-algorithm/".

[3] unknown, "Miller–rabin primality test." "https://en.wikipedia.org/wiki/Miller%E2%80%93Rabin_primality_test#Miller%E2%80%93Rabin_test".

[4] unknown, "Python implementation of the miller-rabin primality test." "https://gist.github.com/Ayrx/5884790".

[5] unknown, "Rsa (cryptosystem)." "https://en.wikipedia.org/wiki/RSA_(cryptosystem)#Key_generation".

[6] unknown, "Implementing rsa in python from scratch (part 1)." "https://coderoasis.com/implementing-rsa-from-scratch-in-python/".

[7] unknown, "Extended euclidean algorithm." "https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm#Description".