Deterministic RSA key generation (D-RSA)

José Costa
DETI Aveiro
University of Aveiro
Aveiro, Portugal
joselcosta@ua.pt

I. Introduction

RSA (Rivest–Shamir–Adleman) [1] is a public-key cryptosystem that is widely used for secure data transmission. In most cases, RSA key pairs are randomly generated for some dimension of their module. This report explains the implementation of a Deterministic RSA key pair generation from a set of parameters, which are analogous to a password that protects a private key stored in an ordinary file. This includes the explanation of pseudo-random byte generator and a RSA key pair generator describing the implementation, compilation, usage, tests and performance statistics.

The original language used for the development was the C version, with the second language being python.

II. Compilation and Usage

This section explains how to compile and run the developed programs. For each version of the programs (C and Python) an explanation is going to be provided.

Both generate the same output if the inputs are equal on both programs.

A. Primary C version

To compile the C version, Make is used with the following command inside the C_Version folder:

```
$ make
```

To execute the Pseudo-random Byte Generator, execute the following command inside the C_Version folder:

```
$ ./randgen <Password> <Confusion_String>
    <Iterations>
$ ./randgen speedtest
```

To execute the RSA Key Pair Generator, execute the following command inside the C_Version folder:

```
$ ./rsagen
```

The input to this program is a stdin sequence of bytes, for example from a file.

Possible and useful executions of these programs can be as follows:

```
$ ./randgen teste op 1 | ./rsagen
$ cat /dev/random | ./rsagen
```

The first example provided uses both of the programs in conjunction, rsagen uses the random bytes generated from the randgen function.

The second example provided generates RSA key pairs using bytes from the /dev/random source.

B. Secondary Python version

To execute the Pseudo-random Byte Generator, execute the following command inside the Python_Version folder:

```
$ ./python3 randgen.py <Password> <
    Confusion_String> <Iterations>
```

To execute the RSA Key Pair Generator, execute the following command inside the Python_Version folder:

```
$ ./python3 rsagen.py
```

The input to this program is a stdin sequence of bytes, for example from a file.

Possible and useful executions of these programs can be as follows:

```
$ ./python3 randgen.py ola ol 1 | ./
    python3 rsagen.py
$ cat /dev/random | ./python3 rsagen.py
```

The first example provided uses both of the programs in conjunction, rsagen uses the random bytes generated from the randgen function.

The second example provided generates RSA key pairs using bytes from the /dev/random source.

III. Pseudo-random Byte Generator

The objective of the Pseudo-random Byte Generator is to generate random bytes from three input sources:

- Password
- Confusion String
- Iteration Count

It generates a pseudo-random 64 byte output to stdout.

A. Explanation

The implementation of this generator uses a common, regular use mathematical pseudo-random number generator. As both versions have to generate the same result the random number generators from Python and C could not be used.

The generator chosen was the Mersenne Twister due to the impressive speed, quality combination and the common use.

A basic explanation for this algorithm is that you start with a seed (if you re-use the same seed you will obtain the same random numbers), you initialize it into a state.

Then, every time you want to obtain a random number, you transform that state with a one-way function g. [2]

When another random number is generated, the state is transformed with a one way function f, then use function g again to output a random number.

For both versions the implementation of generator was used from the following sources:

- **C Version:** https://github.com/ESultanik/mtwister [3]
- **Python Version:** https://github.com/james727/MTP [4]

The steps followed to successfully implement the generator according to the guideline were the following:

1) First Step: With the 3 inputs (Password, Confusion String, Iterations), generate a bootstrap seed by using the key derivation function PBKDF2 with SHA1, HMAC and PCKS5.

The password is used as password, the confusion string as salt and the iteration as the iteration count of PBKDF2.

2) Second Step: The second step is responsible for the generation of the confusion pattern.

Using the confusion string, the MTwister generator is seeded and used to generate an equal size confusion pattern.

3) Third Step: Seed the MTwister generator with the bootstrap seed.

4) Fourth Step: Use the MTwister generator to generate bytes until the sequence of bytes is equal to the confusion pattern.

5) Fifth Step: With the same MTwister generator from the previous step generate a new bootstrap seed and seed the generator again.

6) Sixth Step: Repeat the fourth and fifth step the same number of iterations as the iteration input.

7) Seventh Step: Generate 64 bytes and output to stdout.

These steps are all summarized and illustrated in Fig. 1:

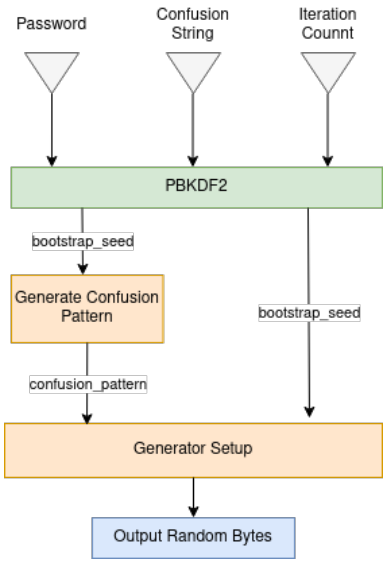

Fig. 1.

It is possible to generate more random bytes other than 64, but for this project in specific, only 64 are generated.

B. Versions Comparison

To verify that the Pseudo-random Byte Generator implemented in both languages is coherent and produces the same results for the same inputs, the following test was conducted:

1) First test:

Input: Password="teste", Confusion String="op", Iteration=2
C command: ./randgen teste op 2 > test
Python command: python3 randgen.c teste op 2 > test
Random bytes generated in hex:(test file):

```
9a68 666d 9d91 0ce7 d43d 5043 7c08 0de1
ce0a 3664 98e9 5b85 76fe 339b bd9a c9f5
fddd ab63 4bc5 e067 17b6 15f2 f7de 3124
de74 fb8e 6dcc 8bc9 ded8 c95d dd9e a798
```

Fig. 2.

C. Performance Testing

On the C version of this application, tests were conducted to verify the impact of the confusions string size and number of iterations on the time necessary to generate random numbers using the Pseudo-random Byte Generator developed. When the C version is executed in the following manner:

```
$ ./randgen speedtest
```

It will perform a test that for each confusion string size of 1 to 3 bytes, with 19 iterations and print a table with the results.

On the tests illustrated results in figures 7,8,9 the iteration number varies between 1 and 19 for a confusion string size of 1 to 3 bytes.

An additional test was also performed to verify the impact of the confusion string size, for a single iteration, presented on figure 10.

Taking into account the results from figure 10, the impact from the size of the confusion string is much larger when compared to the number of iterations.

With the results from figure 7,8,9 for the same confusion string size, as expected with each iteration the elapsed time grows in a semi linear manner on every test.

IV. RSA Key Pair Generator

The objective of the RSA Key Pair Generator is to generate a 1024 bit public and private RSA keys using a stdin 64 byte stream input as a "Password".

A. Explanation

To implement the RSA Key Pair Generator, parameters from the RSA cryptosystem were calculated based on the stdin input. All of the following steps explain how to the key generation works:

1) First Step: The first parameters to be calculated are the safe prime numbers p, q and multiply them to get modulus N.

$$N = p * q$$

To generate both p and q, the next safe prime number approach is used. This consists in finding the next prime, where both itself (p) and p*2+1 are prime.

To guarantee that the modulus N is 1024 bits, both p and q must have a minimum size of 512 bits.

Before finding the next prime safe prime p, the stdin value last 2 bits are set to 1 to guarantee the 512 bit minimum size, and the next safe prime is calculated. To calculate q, the same approach is conducted, but insted on using the stdin value as the start input, the already calculated p value is used.

2) Second Step: With p,q and N calculated, and using the public exponent e (65537), the z value can be calculated with

$$z = (p - 1) * (q - 1)$$

and the d value can be calculated by the modular multiplicative inverse of e modulo z.

$$d = e^{-1} \pmod{z}$$

3) Third Step: Calculate the dmp1, dmq1 and iqmp.

$$dmp1 = d \pmod{p - 1}$$

$$dmq1 = d \pmod{q - 1}$$

$$iqmp = q^{-1} \pmod{p}$$

4) Fourth Step: With all values calculated, use openssl C library or cryptography Python library to generate both a public and private RSA keys in the PEM format.

The public key contains the public modulus N and the public exponent e.

The private key contains the modulus N, the public exponent e, the private exponent d, both prime numbers (p and q), the prime numbers d mod (p-1), d mod (q-1) and the coefficient.

For both versions, the final files will be named private.pem and public.pem .

B. Versions Comparison

To verify that the RSA Key Pair Generator implemented in both languages is coherent and produces the same results for the same inputs, the 2 tests were performed:

1) First test:

Input: Payload File
Payload Path: C_version/tests/payload
C command: cat payload | ./rsagen
Python command: cat payload | python3 rsagen.py
Keys generated by both versions:

```
-----BEGIN PRIVATE KEY-----
MIICdgIBADANBgkqhkiG9w0BAQEFAASCAmAwggJcAgEAAoGBAOo/uPVVrWHEiA9W
XCX9FrJRsimCh6snYdZg21YJnJNKyZRm/WMDH9Aa5naJd5a2YJs2tyMRJZuHJY4J
Hpd04f0h1CdcLaLKrRzJFyZ2uH09q6zPQBvK7LGFMhXvxWRPPWoMeO9mD1we2E37
RG9nbubUVV30+6XCZGFAojhJJlGBAgMBAAECgYAywGGbTdfFweXUpNpMJLxhIBAg
rDYmn96zqsJ6rpRs2Rw5eeLWK9wdo2ZhMkFlttfRaRZek515R9ucHV+yYtF/M1NN
ELJkZYMm1dMl9xsqoSDdTnyDLh0ElpdLqATOSKXkOcKG6Xda5vCOBEEqdAq+0qVH
HURYEw870rZRr2xvUQJBAPTiEXr/o+iPeNa5zfOTLsUZLG3Y1P1itNJtwH6v/pGF
9n4NlcElB02lnx0VmYA2tn1d/1lLkEjICu1a9+yM8sUCQQD04hF6/6Poj3jWuc3z
ky7FGSxt2NT9YrTSbcB+r/6RhfZ+DZXBJQdNpZ8dFZmANrZ9Xf9ZS5BIyArtWvfs
j9+NAkBG565oDt1G2UQD6IjZniRMrZ1PsRj6xTnWNDEofY0YVdGpbgWOh0o2EV6g
7UAk/7KXWkA1esusRz/iuhRPSgCBAkEA2YJPaoPUr8V33Knuz7HksfbZKryU+neJ
CvunJh/mqpVlyrMAWC0unn5EpQ3BaUuvrg0muMbQYoUkh4zsd7eXJQJAF7Gm6A9Z
/drdlhh0dhZv04kFZH9Apcv/mOPrQMqJumqURkcHx108hnIdFz+mYeRze45zmCIO
sDEL2R+vhF+nUA==
-----END PRIVATE KEY-----
```

Fig. 3.

```
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDqP7j1Va1hxIgPVlwl/RayUbIp
goerJ2HWYNtWCZyTSsmUZv1jAx/QGuZ2iXeWtmCbNrcjESWbhyWOCR6XdOH9IdQn
XC2iyq0cyRcmdrh9Pausz0AbyuyxhTIV78VkTz1qDHjvZg9cHthN+0RvZ27m1FVd
9PulwmRhQKI4SSZRgQIDAQAB
-----END PUBLIC KEY-----
```

Fig. 4.

2) Second test:

Input: randgen
C command: ./randgen teste op 2 | ./rsagen
Python command: python3 randgen.py teste op 2 | python3 rsagen.py
Keys generated by both versions:

```
-----BEGIN PRIVATE KEY-----
MIICdgIBADANBgkqhkiG9w0BAQEFAASCAmAwggJcAgEAAoGBALpV+QYSZphUdp3V
E7fWzjwNA7qZhU8CPW+kQQas7DMsMVMgFcmT92PxodnFeqXAd9q6PmLRf9IPwWM4
Fki5r/ilQmKtn7eCkw2vNW0n2DQHAREFyKTFTGxXPPZUz/ckEaXwrYTjJz7Z4rhc
1KJ4ijUYrapyXlZ8ZS27OkKO9/OdAgMBAAECgYEAgDHlwWenbeWuNagCiNgfyvCp
QBeev1yrJ8O5eP4/AnXE+Yt5cdV6viffUhOq7Z4c1z6K7TF03vA3/Cr6Kc6WYS3K
92hoCuKoi1hBmJwG7apwL9iS5mI+o2XmJI/pX8HsJ83Kf+FEeNNT53dCwXTxgdsA
KMEL0YErFgmOWBQRx4ECQQDaaGZtnZEM59Q9UEN8CA3hzgo2ZjjpW4V2/jObvZrJ
9f3dq2NLxeBnF7YV8vfeMSTedPuObcyLyd7YyV3dnv6jAkEA2mhmbZ2RDOfUPVBD
fAgN4c4KNmSY6VuFdv4zm72ayfX93atjS8XgZxe2FfL33jEk3nT7jm3Mi8ne2Mld
3Z+ovwJATYeB1Kd6NfqVfK5hRkYVp3JcbbKIFU5HslXKPB6yM77+vxF3zg4P+oiQ
OnplQppvXAhqw8R2N6kRUiMpDuLBwQJAQS93oq+sYxWuGr6h5ZrhycscS8SFFGDk
bKdh79GfKTfqFQbaE/tzBlHlXpetsvsn7d0sV04eBnqxsDIpecP1MQJALuNsE0nI
kOuaYeKJp9MltfkMaM44CTrgMQTUBU5MXI+m1Y7rerXeEcIvtupC9QDTxC4HtIWP
XDqd0tKm0NhT9A==
-----END PRIVATE KEY-----
```

Fig. 5.

```
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQC6VfkGEmaYVHad1RO31s48DQO6
mYVPAj1vpEEGrOwzLDFTIBXJk/dj8aHZxXqlwHfauj5i0X/SD8FjOBZIua/4pUJi
rZ+3gpMNrzVtJ9g0BwERBcikxUxsVzz2VM/3JBGl8K2E4yc+2eK4XNSieIo1GK2q
cl5WfGUtuzpCjvfznQIDAQAB
-----END PUBLIC KEY-----
```

Fig. 6.

REFERENCES

[1] RSA (cryptosystem), Available: https://en.wikipedia.org/wiki/RSA_(cryptosystem) .

[2] How does the Mersenne's Twister work?, Available: https://www.cryptologie.net/article/331/how-does-the-mersennes-twister-work/.

[3] The MTwister C Library, Available: https://github.com/ESultanik/mtwister.

[4] MTP - The Mersenne Twister Pseudo-random number generator, Available: https://github.com/james727/MTP.

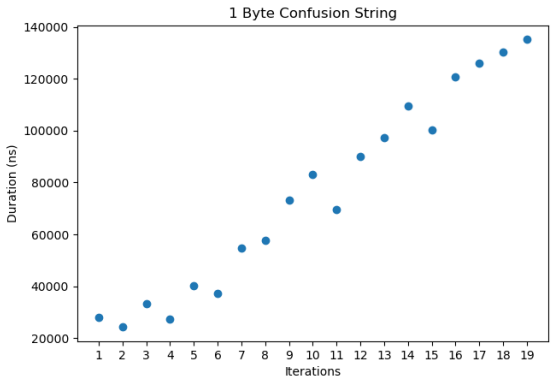

Fig. 7.

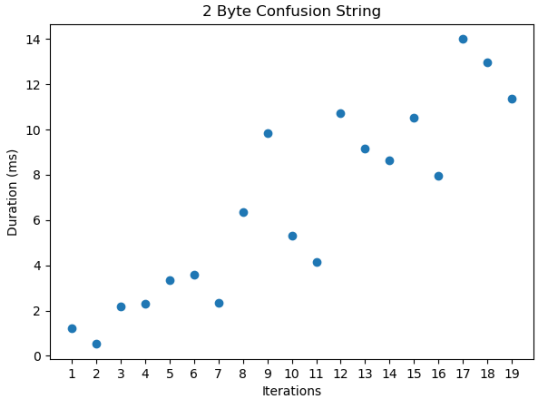

Fig. 8.

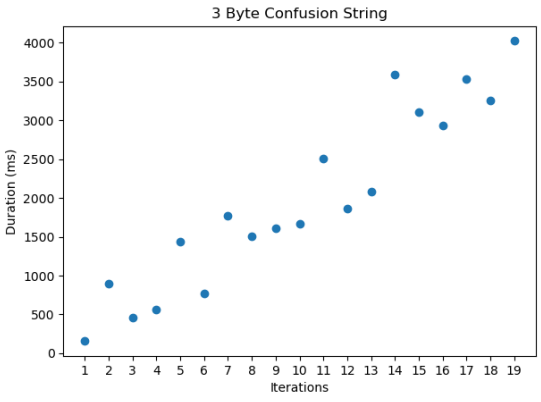

Fig. 9.

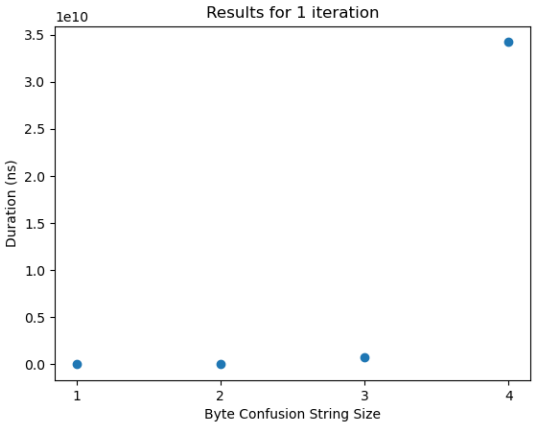

Fig. 10.