D-RSA

A python implementation of deterministic RSA key generation

1st Dinis Cruz
DETI
UA
Aveiro, Portugal
cruzdinis@ua.pt

***Abstract*—This document follow the implementation and results of the second project for the course of Applied Cryptography, where the objective was to implement a deterministic RSA key generator based on a pseudo-random number generator that can take an arbitrarily time to setup with no option for parallelism.**

***Index Terms*—Deterministic RSA, Pseudo-random number generator, LFSR**

I. Introduction

The document follows the implementation of a pseudo-random number generator based on linear feedback shift registers with similarities of A5/1. The document also describes the implementation of deterministic RSA keys of arbitrary size outputted to PEM files, these keys are created with numbers generated by the previously referred pseudo-random number generator.

II. Program Structure and Usage

This project has a file (rsagen.py) that takes command line arguments and generates a key pair of RSA keys, 1 file that takes command line arguments and either outputs a stream of random bytes or outputs statistics regarding the generator and it's setup time. The generator is in a module to create structure within the code.

A. Usage

1) randgen.py: This script has 2 main flags:

- s Creates graphics with statistics about the pseudo-random number generator .
- g Outputs a streams of bytes to stdout, the argument passed must be an integer and if this integer is bellow 0 then an infinity stream is started, if a positive integer is passed those number of bytes are outputted. This options also unlocks further flags, these being:
  - p Takes a string to be the password of the generator.
  - c Takes a string to be the confusion string of the generator.
  - i Takes an integer to be the iteration count of the generator.

Example:

```
python3 rsagen.py -p password -c a
            -i 1 -s 512 -f mykey
```

2) rsagen.py: This script has 5 flags:

- p Takes a string to be the password of the generator.
- c Takes a string to be the confusion string of the generator.
- i Takes an integer to be the iteration count of the generator.
- s Takes an integer pretending the size of the key to be created.
- f Takes the prefix of the file to store the keys in PEM format.

Example:

```
python3 randgen.py -p password -c o
                    -i 1 -g 200
```

B. Pseudo-number generator module

Allows for the creation of the pseudo-random number generator and generation of bytes. Most notable methods:

- start(Password,Confusion_string,Iteration_count) - Starts the generator and the whole process this encapsulates.
- seed(Password,Confusion_string,Iteration_count) - Creates a 20 byte seed from these components with a method based on PBKDF2 key derivation.
- next() - Outputs a pseudo-random bit in integer form (0 or 1)
- nextByte() - Outputs a pseudo-random byte in integer form (0 to 256)

III. Implementation

A. Pseudo-random number generator

The pseudo-random number generator has a specific intent, takes 3 parameters (Password,confusion string and iteration count) and creates a seed for initialization, after that it must have a strategy to take an arbitrarily amount of time to setup and after, to output bytes.

1) Seed creation: The seed creation takes inspiration from the PBKFD2 method, a initial seed is created using the hmac_sha1 method. SHA-1 is used because, as will be discussed further the 20 byte output is very interesting when it comes to initializing the linear feedback shift registers and outputting a fixed amount of bytes equally distributed is a job for a hash function.

The hmac part of the algorithm comes from the need of using the password and the confusion string to create this seed. The algorithm takes the password and creates 2 64 byte pads xor'ing them with pre computed standard arrays, the message (In our case the confusion string is hashed to create this message) is then concatenated with the first of the pads and hashed, this hashed value is concatenated with the second pad and hashed to output a seed, this process isn't very complex but makes it hard to recover the password and the confusion string knowing this seed.

The process is repeated as many times as the number of iteration counts, on each iteration the seed is xord'ed with the hmac_sha1 computation using the password and the previous seed. At the end a 20 byte array is outputted.

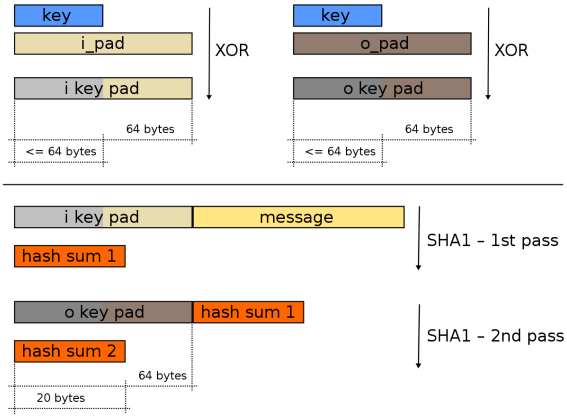

Fig. 1. HMAC-SHA1 graphical explanation taken from Wikipedia

2) Linear feedback shift registers configuration: Based on the A5/1 generator 3 linear feedback shift registers were chosen, these however have lengths that when combined make 2 important features:

- The sum of bits composing the 3 LFSR's must be 160 so that the full seed can be used to initiate the generator, this means that all lengths must come to 160.
- All of them must be of the maximum length for periodicity and use only 2 bits to generate the next so computations are easier.

This leads to the choosing of LFSR'S with length and taps(Numbers looked at when there is a need to create the output of the LFSR) such as:

Length	Taps
79	[79,70]
58	[58,39]
23	[23,18]

With these LFSR'S, when a new bit is requested each of them outputs 1 bit, the xor between the 3 is done and that is the result outputted. The LFSR's don't all update after the bit is outputted, out of the 3 bits created (one for each LFSR) there will be a majority, either 0 or 1, the LFSR's who created bits equal to the majority are updated, the other is not. This leads to not all registers moving at the same pace and with information that is never leaked to the output.

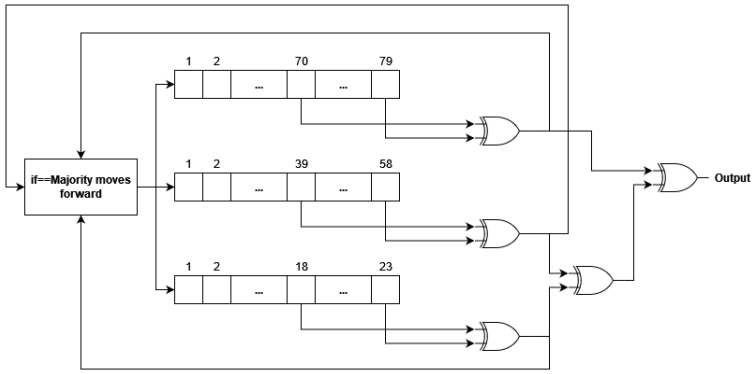

Fig. 2. Graphical representation of the pseudo-number generator created, based on A5/1

3) Starting the generator: Having a seed created the setup time of the generator is controlled by the confusion string length and the number of iteration counts. The strategy is to initiate the generator with this seed and after that iterate over the generator trying to find the first x bytes of the confusion pattern. This confusion pattern is created by hashing the confusion string and capturing it's length of bytes counting from the start. This means that with a confusion string of size 2 we will hash it, producing 20 bytes, we take the first 2, start generating bytes with the generator and after we find those 2 bytes we generate an additional 20 to create a new seed and initiate the generator again. Repeating this process for the number of iterations inputted by the user. To be noted that this process does not allow paralelism since to make since of the current iteration we must account for the ones before because the generator was initated with previous information.

B. D-RSA

Creating deterministic RSA Keys, revolves around the creation of (very) large prime numbers, and this is done by creating large random numbers and seeing if they are primes, since primes are fairly common. The deterministic part is guaranteed considering the generator if initiated with the same parameters will output the same bits, which will lead to the same primes and eventually to the same keys. Security is also configurable with the parameters passed to the generator, while we can create a small keys in a matter of seconds we can also create a large key with a generator that takes minutes or hours to setup. This can be done with the usage of large iteration counts or bigger confusion strings.

1) Generation of primes: In an infinite loop, n bits (Number inputted by the user cut in half) are generated and concatenated, cast to an integer and a verification is made whether the number is prime or not, if it is, great we can return it, if not we gotta repeat the generation process.

2) Checking if a number is prime: The method used is the Rabin-Miller algorithm taken from here and it is complemented by simple checks like:

- is the number odd? If not the only prime not odd is 2 so we can throw the number out
- Check if it is divisible by the first primes up until 997 so that any number divisible by such low primes can be discarded

An important part of RSA primes is to make sure that, the prime subtracted by one does not have low prime factors, so this aspect is also checked at the end of the checking process of a number being a prime. Because it is not only important that the number is prime, it must be a suitable rsa prime.

3) RSA Algorithm: After having 2 large primes numbers we can multiply them to obtain the modulo (N), with e being a prime pre-selected (65537) the inverse can be calculated easily in python with the function pow that takes 3 arguments: the base, the exponent and the modulo to divide. This follow the equation:

$$d = e^{-1} MOD\ N \tag{1}$$

After that all components of an RSA key are generated and there is only the need to store them in a standard format. The code makes this process more easily understandable:

```
def makeRSAKey(n,f):
    p = getRSAPrime(n//2)
    q = getRSAPrime(n//2)

    n = p*q

    e = 65537
    d = pow(e, -1, (p-1)*(q-1))

    key_obj = {
        "n": n,
        "e": e,
        "d": d,
        "p": p,
        "q": q,
    }

    writeToPEM(key_obj,f)

    return e,d,n
```

4) Writing to PEM format: Most keys generated in python are done using libraries and these libraries have their own format for storing this information. With the python module rsa (here) which allows for the creation of private and public keys objects using only the integers that make up the keys we can then export them to PEM format.

IV. Results

A. Uniformity of numbers generated

To achieve pure random number generation we must ensure that from 0-255 all numbers are being generated an equal amount so for 10 Million bytes generated this was the result.

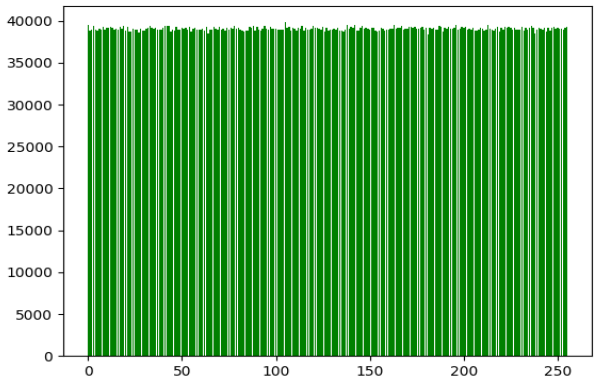

Fig. 3. Uniformity in results produced from the generator

B. Non pattern generation

Uniformity means nothing if the numbers generated can be predicted based on a pattern for that an image with 1024*1024 pixels was created using random pixels created by the generator to show no pattern formation.

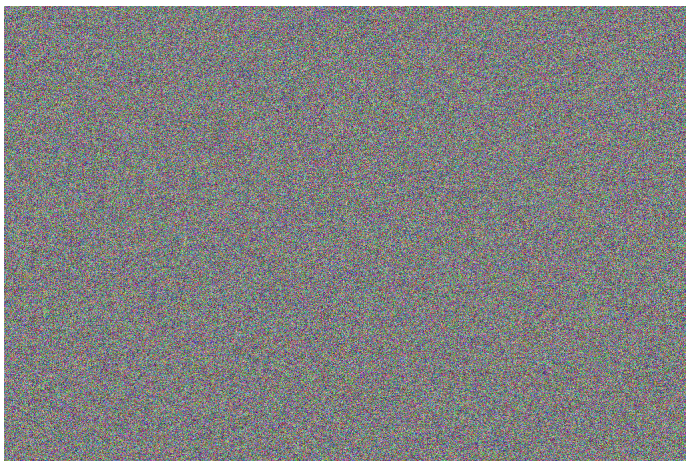

Fig. 4. Uniformity in results produced from the generator

C. Changing iteration count

The main objective was making the steps and time configurable before the generation of random numbers could take place. One way to control this is the iteration count. For this we can see 2 graphs that show the impact of different iterations (1,5,20,50,100,200) on the time it takes to setup the generator with 2 different confusion string lengths. The results show a clear definition of what the iteration count makes to the time of setup. It is a linear function.

For the confusion string with length 1 (estimated time and relation taken from the graph):

$$time = 0,005 * iterations \quad (2)$$

For the confusion string with length 2(estimated time and relation taken from the graph):

$$time = 1,25 * iterations \quad (3)$$

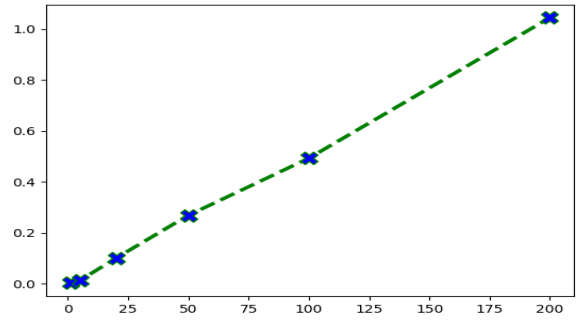

Fig. 5. Iteration change for confusion string of length 1

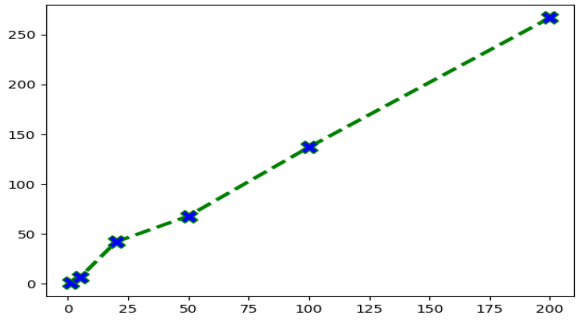

Fig. 6. Iteration change for confusion string of length 2

D. Changing confusion string length

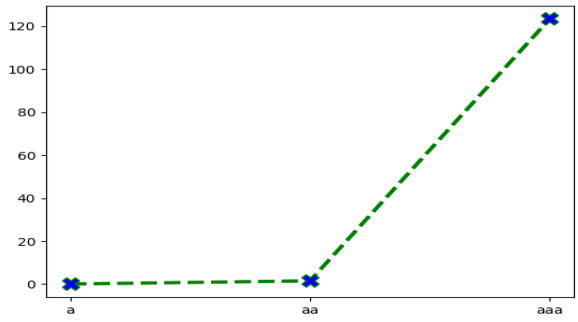

Fig. 7. Uniformity in results produced from the generator

Finding a pattern of bytes, in contrast to the iteration count, is a exponential problem, since finding 1 byte has a probability of 1 in 256 but finding 2 has a probability of 1 in 256*256 and so on. This is also shown in the graph where finding a confusion pattern of 3 already takes 2 minutes.

E. Time to generate RSA Keys

It is also interesting to see how much time it takes to generate RSA keys as the process it self. Using a simple password (”password”), a confusion string of length 1(”a”) and an iteration count of 1 to make the process of setup basically invisible we get the following results:

Size(bits)	Time(s)
4096	14.90
2048	3.33
1024	3.22
512	0.43

V. Analysis of the results

With these results we can ensure that if a user chose to use our application to create secure but deterministic RSA Keys he could do it with a good amount of certainty about the secrecy he is getting. We can get setup times up to hours which the user shouldn’t mind since he only needs to create these keys one time while a possible attacker would have to compute and waste a several amount of resources to try and crack the generation. If the user is looking for a exponential amount of increase he can try and make the confusion string bigger, if he is content with linear growth he can make the iteration count bigger or even mix the 2 approaches. The pseudo-random number generator can have other applications as it is uniform and pattern free(to a certain tested point of generated bytes). The usage of python has it’s implementation advantages but it is much slower than other languages, suing a lower level language like C or a compiled language like java we could get much more iterations and bigger confusion strings with less time making the process even harder on the attacker if a language like these is not used for the attack.

VI. Conclusion

The work developed was completed with the expected specifications and offers what it is intended to offer. My amount of knowledge about pseudo-random number generator as also greatly increased with his project as well as the creation of large primes.