SECURITY FOR THE INTERNET OF THINGS

Setareh Nouri

Computer security and cryptography faces some complexity in arithmetic like factorization of a large integer into its prime factors. As a result, we should be able to generate random numbers and prime numbers at first.

The Disposable Mask a good symmetric encryption technique for the Internet of Things. Which satisfies three conditions: the mask is really random; it has the size of the message and it is only used once. We use the same key is used to encrypt and decrypt.
This technique is implemented in two different ways, OneTimePadSeduoRandom.py which is based pseudorandom algorithm and DisposableMask.py.

A- **OneTimePadSeduoRandom**: Consists of three following functions:
- init(): constructor of the class.
- Encryption(): applying bit by bit exclusive OR between the original message and this random sequence.
- Decryption(): the same operation, bit-by-bit exclusive OR between the cryptogram and this key.
- pseudorandom(): which produce the key randomly by algorithm.

B- **DisposableMask**: in DisposableMask class the mask is generated using different algorithms for generating Random numbers. It Consists of the following functions:

- init(): constructor of the class. The seed and key arguments are used to instantiate the random number generator. The generatorType attribute, the type of generator to produce random numbers for the mask is determined: 'lcg', 'xorshift', 'bbs', lfsr 'or' mersennetwister '.
- encrypt(): applying XOR between the original message and the generated mask. In this function it receives the msg as in input parameter and returns the encoded message and used mask.
- decrypt(): the same operation, exclusive OR between the cryptogram and the same mask. In this function it receives the encoded message and mask as inputs parameter and returns the original message.
- generateMask(): generating a mask of the same size as the message original.

C- **RandomGenerators**: in this directory there are following different classes of random generators:

I. BBS.py:
Blum Blum Shub is a Cryptographic Generators which calculates the sequence of prime numbers by iterating in this equation: $X_{n+1}=X_n^2$ mod M where M is the product of two large prime numbers p and q. These values of p and q are both congruent to 3 mod 4. This algorithm is implemented using following functions:

- init(): The constructor of the class, with three attributes: seed, which is the starting point of the generator. Start and end which is used for the generating the prime numbers p and q between start and end
- generate_primes() : Generates prime numbers between start and end
- getPrimes(): returns a list containing the prime numbers p and q satisfying the condition in which two prime numbers p and q, should both be congruent to 3 modulo 4
- generator(): Calculate the above mentioned equation for generating random numbers.

II. LCG.py :

Linear congruential generators (LCG) are defined by: $X_{n+1}=(a.X_n \ +c)\%$ m and c and m are coprime integers (their gcd is 1). For each prime number p dividing m, (a−1) is a multiple of p. m multiple of 4 ⇒(a−1) multiple of 4. (valid for c≠0):

- init(): The constructor of the class, with three attributes: seed, which is the starting point of the generator.
- generator(): Calculate the above mentioned equation for generating random numbers.

III. LFSR.py:

LFSR is a classic generator. A linear feedback shift register (LFSR) is a shift register whose input bit is a function linear from its previous state.

- init(): The constructor of the class, with two attributes: seed, which is the starting point of the generator and connection The bit indices used by the linear function which generates the input bit
- generator(): Calculate the above mentioned equation for generating random numbers.
-

IV. MersenneTwister.py:

MersenneTwister is a classic generator. Mersenne Twister is a powerful pseudo-random number generator as having a long period
(how many values it generates before repeating itself) and a statistically uniform distribution of values
(bits 0 and 1 have the same probability of appearing whatever the previous values)
https://en.wikipedia.org/wiki/Mersenne_Twister:

- init(): The constructor of the class, with three attributes: seed, which is the starting point of the generator.
- generator(): Calculate the above mentioned equation for generating random numbers.
- twist(): generate the next n values from the series x_i

V. XORSHIFT.py:

This generator has three integer parameters. We start with a 32-bit vector, which serves as a seed. To calculate the new term from the current term x:

- We replace x by the result of the exclusive or bit by bit between x and its version shifted by bits to the right (circular shift, the outgoing bits on the right return to the left).
- We replace x by the result of the exclusive or bit by bit between x and its version shifted by bits to the left (circular shift).
- We replace x by the result of the exclusive or bit by bit between x and its version shifted by bits to the right (circular shift).
-

The value obtained after these three operations is the new term of the sequence, to be produced as output. The mentioned algorithm is implemented by to functions:

- init(): The constructor of the class, with attributes: seed, which is the starting point of the generator and a, b and c: The generator parameters.
- generator(): generate the random numbers.

D- **Rapid Exponentiation**

Various operations in cryptography require calculating a multiple or a power. These operations must therefore be able to be carried out efficiently, especially since the objects considered here (the wireless sensors) are not very powerful and on battery.
This goal is implemented in two ways:

I. RapidExpoRec(X, e):
- if e is even, we calculate (recursively) $X^{e/2}$, which we will multiply by itse
- otherwise, we calculate (recursively) X^{e-1} , which we will multiply by X.

II. RapidExpo(X, e):
Since the above technique is recursive, its implementation in a sensor may be problematic. But as the operations to be done are determined by the parity of the number following a division by 2, or has a division by 2 after a deletion of 1, we therefore deduce that the binary writing of the number contains exactly the operations to be done.

E- **Obtaining Large Prime Numbers**

Most current security approaches boil down to the problem of finding the prime factors of a large integer: since Antiquity, attempts have been made to do so effectively, and no solution has yet been found. We must therefore be able to find two large original prime numbers, and make their product.

I. Primes.py:

A class with the following functions for generating prime numbers:
- obtain_prime_numbers (start, end): takes two integers as parameters, start and end and returns the list of prime integers between these two parameters.
- Eratosthene(): Takes an integer as input then list of prime numbers less than limit.
- **FermatTest()**: check the primality of input parameter using Fermat's theorem: $a^n \equiv$ a [n]
- RevWitness(): this function used by Miller Rabin and returns True if a number is prime
  Miller_Rabin() : give the input and check the primality of it calculating the equation: $a^{p-1} \equiv 1$ (mode p)

F- Asymmetric encryption algorithms

Two type of key is used for cyphering. Public key is shared while the private key is kept secret.

I. RSA.py:

RSA algorithm is an asymmetric algorithm, in which public key [n,c] and d is private key and . P1 and P2 are two large prime numbers close to each other, the message t is to be cyphered, the mask will be t^{cd} and $(t^c)^d \equiv$ t [mode n]

- init(): constructor of the class which initiates P1 and P2 then calls the key () function that generates the keys
- key(): function for generating the private and public keys, n,c,d by calling bezout() based on P1 and P2.
- encrypt(): based on the equation the original message is encrypted using mask.
- decrypt(): based on the equation the original message is unmasked.
- bezout() : Take two parameters a and b as input and return their Bezout coefficients.

II. El GAMAL.py

The ElGamal algorithm is an asymmetric cryptography algorithm based on discrete logarithms as following:

- P and g are prime numbers, and g primitive root modulo p.
- a is chosen from {0,...,p-2} , A= g^a hence the public key is (p,g,A).
- b is chosen from {0,...,p-2} , B= g^b. For encrypting massage m, (B,c) is send, where c= $A^b * m$.
- And for decryption m= (B^{p-1-a}*c)

And implemented by these functions:

- generatekey(): This function will generate a pair of keys (public, private) used for encryption and decryption. The public key contains three integers (p, g, A).
- init(): constructor of the class which initiates p and g then and then calculates the two private keys and public
- encrypt(): input message and receiver Public Key and returns the cryptogram.
- decrypt(): input cryptogram and sender Public Key and returns the message.

III. BLUMGOLDWASSER.py

The BlumGoldWasser algorithm is an asymmetric cryptography algorithm which is combined to pseudo random Blum Blum Shub bit generator.BlumGoldWasser includes three algorithms; a probabilistic key generation algorithm that produces a public key and a private key, an algorithm probabilistic encryption and a deterministic decryption algorithm. Implemented as following:

- P and g are prime numbers, and g primitive root modulo p. p, q are two large distinct prime numbers satisfying the congruence condition at 3 mod 4. (p, q) is private key, n= pq is public key and h is the block size

.

- init(): the constructor creates the BBS generator which is a pseudo-random number generator for generating the keys, then initiates the values p, q, n and h and calls the generateKey() function.
- generateKey(): this function will generate a pair of keys (public, private) used for encryption and decryption.
- encrypt(): input the message M to be encrypted, and returns the cryptogram in the form c, the cryptogram is all the c_i values plus the final x_{t+1}value: (c_1 , c_2,..., c_t, x_{t+1})
- decrypt(): input the cryptogram (c_1 , c_2,..., c_t, x) and deciphers by private key(p,q).
- bezout() : Take two parameters a and b as inputs and return their Bezout coefficients.