Homework #3
RSA

CNS Course Sapienza

Riccardo Prinzivalle, 1904064

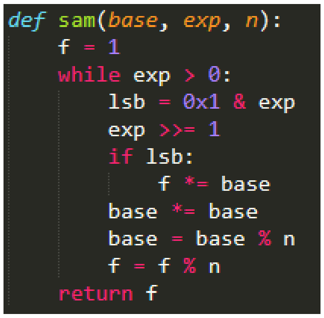
November 20, 2020

1 Homework Goal

This homework contains an implementation of RSA algorithm based on major libraries for the mathematical functions, a comparison with the insecure *PyCryptoDome* RSA and with the *PyCryptoDome* AES.

2 RSA Implementation

RSA basically is divided in two part: the initialization and the encryption.
The encryption phase is simpler than the initialization: it uses only an exponentiation and a modulo reduction. The modulo reduction is already implemented in python (used in this homework), instead what it is needed is an efficient implementation of the exponentiation. To do so, I used the proposed **Square And Multiply** in the slides of the course: at first with small values of base and exponent it worked flawlessly, but during the test with bigger messages and keys, the computation explodes, so I thought to reduce in modulo after every performed computation (since we have to do it after the exponentiation, so why don't do it at every step?) and the **SAM** computation time dropped to some seconds. The code can be seen in Fig. 1



```
def sam(base, exp, n):
    f = 1
    while exp > 0:
        lsb = 0x1 & exp
        exp >>= 1
        if lsb:
            f *= base
        base *= base
        base = base % n
        f = f % n
    return f
```

Figure 1: Improved version of Square And Multiply

Now the easy task is done; the initialization is the part which requires more attention to guarantee the security of our implementation. The operation to be performed are:

1. find two large prime numbers p and q
2. compute $n = p \cdot q$
3. compute Euler Phi function as $\Phi(n) = (p-1)(q-1)$ exploiting prime properties of p and q
4. select an exponent $e \in 1, 2, \ldots, \Phi(n)$ such that $gcd(e, \Phi(n) = 1$
5. compute d such that $d \cdot e = 1 \mod \Phi(n)$

Now every operation will be analyzed in more depth.
The first step is quite complicate: the machine, a deterministic entity, must find randomically two numbers in an unpredictable way, we have to use them in cryptography so no one must be able to exploit some not true random generations. As explained in [3], the idea is to let the computer generate a random number of the dimension we need and then test if it is a prime or not; that is possible since the dimension of the number we need for RSA encryption still guarantees to obtain a good enough probability to obtain a prime, this probability decreases as the order of the generated number increases; the same concepts can be found in chapter 7 of [2]. The problem is how to generate these random numbers: as stated in [4], we cannot simply use a pseudo random number generator, it will generate numbers that are only apparently random, and this property can be exploited by an attacker to find a pattern in these random numbers. The most secure way to generate a random number is to have a true random number generator, which uses some environment input to have a source of true randomization, the problem is that these generator are hard to design and costly o implement. This source [4] explains also that it is possible to find some services on the internet that claim to have TRNG, but for this implementation it will be used the idea of [1]: we can use cryptography secure random number generator, which uses some special source of randomization from outside the computer to initialize a random number generator. This is done by using an utility from *PyCryptoDome* that generates random numbers (in this implementation it has been specified to use os.urandom, which is cryptographically secure, but I have a doubt if it check to have enough entropy in the pool or has the same behavior specified in slide 8 about prime numbers) and then performs the primality test.
The second step perform this multiplication in order to have the number n composed only by two prime factors, and this property will be exploited in th next step.
The third step computes the Totient function by exploiting the previous step property and using simpler computations.
The fourth and the fifth step are implemented together in this work: since in the fourth step we need to compute the GCD, this can be done trough the

Extended Euclidean Algorithm needed in the fifth step. To select the exponent, it is used SystemRandom, which uses and delegates to the underlining system to compute the random number and in addition it has already implemented the possibility to add the source of randomness, as it is needed in the fourth step. As already said, the GCD test is performed by the EEA algorithm, which is needed to compute d. The EEA implementation is iterative, it is based on the slide pseudocode with some improvements, on the net I was able to find also a recursive version, maybe it has more performance, but since RSA needs big numbers, the recursive version reached the limit number of recursive executions and it is not suited for this case.
To recap, the first three steps are executed by the function *initialization*, while the other two are performed by the function *selExp*, which uses *eea* as support function.

3 RSA Comparison

This work compares the proposed implementation of RSA with a common library implementation of RSA itself and AES, the last one just to have an idea of how symmetric ciphers are in general more computational efficient than asymmetric ones. To have a fair comparison, AES uses 128 bit key, so as stated in table 1, it is necessary to have 3072 bit key or RSA. In the implementation proposed, since the key must have a length of 3072 bit, both p and q have a length of 1536 bit so their product has the correct length for the resulting key, as stated in section 7.6 of [2].

Algorithm Family	Cryptosystems	Security Level (bit)			
		80	128	192	256
Integer Factorization	RSA	1024	3072	7680	15360
Discrete Logarithm	DH, DSA, Elgamal	1024	3072	7680	15360
Elliptic Curves	ECDH, ECDSA	160	256	384	512
Symmetric key	AES, 3DES	80	128	192	256

Table 1: Key length comparison in public key and symmetric key algorithm

4 Conclusion

After this brief introduction on elliptic curves, it is obvious why they have been widely adopted in many cases of asymmetric encryption: they use less bits for the same level of security, so are more efficient to compute and their base concepts are easier to visualize. As suggested on section ??, EC can be easily used in hybrid encryption scheme, in the key exchange phase.

References

[1] Can i generate authentic random number with python? `https://stackoverflow.com/a/22891612`.

[2] C. Paar and J. Pelzl. *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[3] I. Pamuditha. How to generate big prime numbers — miller-rabin. `https://medium.com/@prudywsh/how-to-generate-big-prime-numbers-miller-rabin-49e6e6af32fb`, 2019.

[4] A. Prudhomme. True random number generator functions? `https://medium.com/swlh/random-functions-a4f36b1dfd8f`, 2018.