CRYPTOGRAPHY #03.1

RSA

Jacek Tchórzewski, jacek.tchorzewski@pk.edu.pl

1. RSA basic parameters

Rivest Shamir Adleman is a certificated asymmetric block cipher. It is approved by NIST, ISO/IEC and also by RFC. RSA uses a public key for ciphering purposes and private key for deciphering purposes – it means that everyone can create a cryptogram but only person which posses private key can decrypt it. RSA allows to create digital signatures (DSs) which work in a reverse way: DSs can be only created with usage of a private key, however, everyone who has a public key can verify it.

- Digital Signature – a block of additional code is added to the original message. Appropriately generated code can verify the integrity of a message as well as its' origin authenticity.

RSA key generations:

1) Find two distinct prime numbers: *p, q*
2) Calculate *n = p * q*
3) Calculate totient of *p* and *q*: *f = (p - 1) * (q - 1)*
4) Find *e,* such that: *1 < e < f* and *GCD(f, e) = 1*
5) Calculate $d \equiv e^{-1} mod\ f$

Note that:
- bitlength of *n* shouldn't be smaller than 2048
- *p* and *q* should differ in bitlength (let's assume a minimum 128 bits of difference between them), however, both should be longer than ¼ bitlength of *n*.
- *e* should be at least 512b long

A private key is a pair *(n, d)* and a public key is a pair *(n, e)*.
Ciphering scheme:

$$c \equiv m^e\ mod\ n$$

NOTE that message *m* must be lower than modulus *n*: *m < n*

Deciphering scheme:

$$m \equiv c^d\ mod\ n$$

Message *m* is interpreted here as a number, same with a cryptogram. Notation RSA – X denotes bitlength of modulus *n*. For example RSA-2048 means that *n* has 2048 bits.

Homomorphism – all computations are done on ciphertext matches computations done on the plaintext. If this rule applies only for a chosen mathematical operations, a cipher is **partially homomorphic.**

RSA is homomorphic only for '*' operator. Thus:

RSA(m1) * RSA(m2) = RSA(m1 * m2)

Note that: m1 * m2 < n

2. Mask Generation Function 1

The algorithm is defined in RFC-8017. MGF1 allows mapping a message *m* to the bit string of a given length. MGF1 is currently the only one standardized masking function. It also assumes the usage of a hashing function. MGF1 allows to extend the message size as well as squeeze it.

Input parameters:
m – bytes of a message
len – desired length of a message in bytes

Assumptions:
H – chosen certificated hashing function
hLen – the length of a hash returned by *H* (in bytes)

Output:
output – modified message

Algorithm:

1) for INTEGER i = 0,…,$\left\lceil \frac{len}{hLen} \right\rceil - 1$ do step 2, 3
2) temp = H(m|| i)
3) output = output || temp
4) return *len* leading bytes of output.

Note that:

- Hash is calculated from a message concatenated with a counter.
- Leading means the most significant.
- In each iteration, *the output* is extended by *hLen* bytes.
- Notation *mgf1(a, b)* means that we are using mgf1 masking function to extend the message *a* into *b* bytes.

Exercise 1:

Write a program that will be generating appropriate RSA keys. Modulus *n* should have 2048 bits (256B). Follow ALL security rules mentioned in chapter 1. Write a function that will be ciphering messages with RSA usage and will return bytes of the cryptogram. Write the second function which will be deciphering cryptogram and will return bytes of the original message. Functions structure:

1) *byte[] RSAEncode(byte[] m, [e], [n]);*
2) *byte[] RSADecode(byte[] crypt, [d], [n]);*

Verification in *main* function:

```
m = "message_from_keyboard"
byte[] crpyotgram = RSAEncode(bytearray(m));
byte[] decoded = RSADecode(cryptogram);
print(String(decoded));
```

Exercise 2:

Write a function that will be returning mask of a message accordingly to the chapter 2. Assumptions:

1) *H* – is a SHA-256 hashing function. You can find it in *java.security.MessageDigest* (or in *hashlib* in python)
2) Accordingly to the previous point, *hLen* = 32 (Bytes).

Function structure:

1) *byte[] mgf1(byte[] message, int len)*

Verification in *main* function:

```
m = "message_from_keyboard"

byte[] mask = msgf1(bytearray(m), 56);
```

Check your function for many different *len* parameters. *mask* have to have always *len* length.