The RSA Encryption Scheme

Jean-Sébastien Coron

University of Luxembourg

1 Implementation of RSA

Install the Sage library, available at `http://www.sagemath.org/`. Alternatively, you can use the Sage Cell Server at `https://sagecell.sagemath.org`. Please submit a `.ipynb` file.

1. Implement the RSA key generation. You can use the function random_prime to generate a large prime, and the function ZZ.random_element to generate a random integer. You can use the inverse_mod function to compute a modular inverse. The function keyGen takes as input the bitsize n of the RSA modulus N.

```
def keyGen(n=512):
  "Generates an RSA key"
  return Nn,p,q,e,d
```

2. Implement the plain RSA encryption and decryption algorithms. You can use the powermod function. Check that decryption works on a random message.

```
def encrypt(m,Nn,e):
  pass
def decrypt(c,Nn,d):
  pass

def checkEnc():
  Nn,p,q,e,d=keyGen()
  m=ZZ.random_element(Nn)
  assert(decrypt(encrypt(m,Nn,e),Nn,d)==m)
```

3. Implement the RSA signature scheme with signature $\sigma = H(m)^d \bmod N$, where the output size of the hash function H is the same as the bit size of N, minus 4 bits. For this, we can concatenate the evaluation of a hash function h (for example, SHA-1), using an index for the message, truncating the last block:

$$H(m) = h(m\|0) \,\|\, h(m\|1)\| \cdots \,\|\, h(m\|k)$$

For the SHA-1 hash function, we use:

```
import hashlib

def sha1(m):
  h=hashlib.sha1()
  h.update(m.encode("utf-8"))
  return h.hexdigest()
```

We use the function Integer(hd,base=16) to convert an hexadecimal digest hd into an integer. The fullHash function below takes as input the message m to be signed (a string), and the length of the modulus. This length can be obtained using Nn.nbits().

```
# lN is the length of the modulus in bits
def fullHash(m,lN):
  k=ceil(lN/160)
  hf="".join(sha1(m+str(i)) for i in range(k))
  hf=hf[:lN//4-1]
  return Integer(hf,base=16)
```

Implement the signature generation and verification, and check that signature verification works. The signature algorithm should compute $\sigma = H(m)^d \bmod N$.

```
def sign(m,Nn,d):
  pass
def verify(s,m,Nn,e):
  pass

def checkSig():
  Nn,p,q,e,d=keyGen()
  m="message"
  assert(verify(sign(m,Nn,d),m,Nn,e))
```

2 RSA with CRT

Implement RSA decryption with the CRT. The key generation generates $d_p = d \bmod (p-1)$, $d_q = d \bmod (q-1)$, $a_p = q^{-1} \bmod p$, and $a_q = p^{-1} \bmod q$.

```
def keyGen_CRT(n=512):
  return Nn,e,p,q,dp,dq,ap,aq

def decrypt_CRT(c,Nn,p,q,dp,dq,ap,aq):
  pass

def checkEnc_CRT():
  Nn,e,p,q,dp,dq,ap,aq=keyGen_CRT()
  m=ZZ.random_element(Nn)
  assert(decrypt_CRT(encrypt(m,Nn,e),Nn,p,q,dp,dq,ap,aq)==m)
```

3 Fermat test

1. Implement the Fermat test of primality.
2. Write a function to generate random k-bit prime numbers, without using the random_prime function.

4 Attack on RSA

4.1 Broadcast attack

Assume that the same message m is encrypted under three moduli N_1, N_2 and N_3, with public exponent $e = 3$. Therefore, the attacker gets the ciphertexts $c_i = m^3 \bmod N_i$, for $i = 1, 2, 3$. We assume that $m < \min(N_1, N_2, N_3)$. Explain how one can recover the message m.

Let
$N_1 = 1184075491674707383364498940246110983$
$N_2 = 4807791263260690541591049475322639611 3$
$N_3 = 2330489420401247492957941549315221024 1$
$c_1 = 1029337128294509379896761354587396175$
$c_2 = 1016323139534260047141004397551260431 5$
$c_3 = 1785097104277403528957688538132924022 1$
What is the value of the message m ?

4.2 Multiple e attack

Assume that the same message m is encrypted under the two exponents $e = 3$ and $e = 5$. That is, the attacker get $c_3 = m^3 \bmod N$ and $c_5 = m^5 \bmod N$. Explain how the attacker can recover the message m.

Let
$N = 1184075491674707383364498940246110983$
$c_3 = 2592232101700861036280986445156886 96$
$c_5 = 699424913584311226961416425309170928$
What is the value of the message m ?