(Preprint for)

R. Caballero and J. Rojo
An efficient implementation of the RSA Cryptographic Technique
pp. 109-116 in
J. Tena y M.F. Blanco(eds.)
Actas de la IV Reunión Española sobre Criptología
Valladolid, 9-11 Septiembre 1996

An Efficient Implementation of the RSA Cryptographic Technique

Rafael Caballero[1] and Jesús Rojo[2]

[1]Died [2]Depto. de Matemática Aplicada a la Ingeniería, Univ. de Valladolid,

Paseo del Cauce s/n, 47011 Valladolid, Spain, e-mail: jesroj@wmatem.eis.uva.es

Abstract

We present a package of Mathematica to carry out the encoding and decoding process of the RSA cryptographic technique; this is one of the best known instances of a public cryptographic system and has a strong presence in the modern literature. We describe and implement the authentication operation (signature). Several examples are shown.

1 Public–key cryptographic techniques

Public key cryptographic techniques Diffie and Hellman [4], were born as a solution to most of the problems of the 'classical' techniques. They were so succesful that as a result the classical techniques have nearly disappeared, as Gardner states in his overview, Gardner [5]. According to the preceding authors, a public key cryptosystem is a pair of algorithms E, D ('encoding' and 'decoding'), that act over a message M producing another one. E and D are the codifying and decodifying procedures, respectively. The properties required are:

- the algorithm D is the inverse of E, that is $D(E(M)) = M$;
- the algorithms E and D are computationally simple;
- the knowledge of the encoding algorithm E does not enable anyone to find the decoding one D;
- if the key is known, both E and D may be easily computed.

The third property implies that one can make public his codifying procedure E without having problems with the security of the decodifying technique D. So everybody can send messages, using E, that only the owner of D, that is kept in secret, can read.

We will assume that it is also verified that $E(D(M)) = M$, so both algorithms are mutually inverse. This property is verified by the RSA public–key cryptographic technique, which is the one we are mainly interested in.

In practice, we can think of M as a number. In fact, there are lots of forms of transforming a text into a number. The method here presented deals with this representation of the messages.

2 RSA technique

The RSA public–key cryptographic technique (after Rivest, Shamir and Adleman [9]) can be summarized as follows: a person chooses two large prime numbers p and q, that are kept in secret, and let $n = p\,q$. After this, another large integer, d, is selected, such that d is relatively prime to $(p-1)(q-1)$. Finally an integer, e, is computed satisfying

$$e\,d \equiv 1\,[\text{mod }(p-1)(q-1)]\ ,$$

that is, e is the inverse of d modulo $(p-1)(q-1)$. (Also d is the inverse of e, for this reason both values are interchangeable.) The values of p and q can be forgotten because they will no longer be needed. The public key pair is given by (e, n) and the private one is (d, n), of which only d is secret.

In order to codify a message expressed as a number M, as we shall see later, with $M < n$, the encryption function is

$$E(M) = M^{\,e}\,(\text{mod }n)\,.$$

This operation can always be done because it only uses the public key, and it can be calculated easily because the mod n operation can be applied in each multiplication.

To decodify a number N, we set

$$D(N) = N^{\,d}\,(\text{mod }n),$$

this operation can only be carried out if one has the secret key. Euler's theorem guarantees that, if $M < n$, $D(E(M)) = M$, as it is needed to recover the original message.

To decipher this system it is necessary to factorize the number n, and so calculate the secret key. The fact that makes this technique actually unbreakable is that a fast algorithm (polynomial time) to factor integers into their primes in a reasonable amount of time is not known. A realistic computation, Knuth [7], shows that factoring a 250–digit number would take more than 3×10^{11} years.

3 Signature procedure

The aforementioned inversion property provides us with an ingenious signature procedure. A person B want to send a message to another person A, so that A is completely sure that it comes from B and that the transmission is safe from strangers. With the preceding technique everybody can send a message to A, who therefore can't be sure of its source. This authentication operation can be carried out by the following method: let M be the message to send, E_A and D_A the public and private keys of the person A, and E_B and D_B their corresponding ones of B. Person B sends the message $E_A(D_B(M)) = N$, applying first B's own private and secret algorithm and afterwards the public one of A. Then A makes $E_B(D_A(N)) = M$ employing A's private key and finally the public one of B, recovering the original message M. As can be easily understood only B can send N, because the private key of B appears, and only A can decodify N for the same reason.

More technical details and a good introduction to cryptography can be found in Akritas [1]. An excellent reference to the problem of factoring integers, that includes a paragraph on the RSA method, more specific than in our case, is Knuth [7]. There are several books about algorithms that include a section about the RSA technique, such as Brassard and Bratley [2], Cormen, Leiserson and Rivest [3], Manber [8], which describe exponentiation procedures. The book Wagon [10] deals with the same matter related to Mathematica.

4 Implementation

Here we present the different functions developed to work with the RSA technique. The use of Mathematica is highly recommended due to its exact precision in the work with very long integers. The functions here programmed require a great deal of work in the usual programming language, that can be saved using our symbolic system. This package runs with 2.0 , 2.1 and 2.2 versions.

A function has been programmed to help the user in the selection of the primes p and q: this function yields the first prime greater than a given integer. The function is

```
NextPrime[a_Integer]:= a /;PrimeQ[a];
NextPrime[a_Integer]:= NextPrime[a+2]/;OddQ[a];
NextPrime[a_Integer]:= NextPrime[a+1]/;EvenQ[a];
```

It is necessary to look for approximately $\ln n$ numbers to find the following prime number of the integer n. For instance, A computes his numbers p, q and n by means of

```
In[1]:= pa=NextPrime[83294656751339022856204186320263 1\
20806543513465016749806531321654065403219840465465406 5\
4984654326543]

Out[1]= 83294656751339022856204186320263120806543513 46\
5016749806531321654065403219840465465406549846543270 61

In[2]:= qa=NextPrime[36549806456219780654621346846032 1\
64890519518743749109512623795165121621621984845763702 5\
9654654210319]

Out[2]= 36549806456219780654621346846032164890519518 74\
3749109512623795165121621621984845763702596546542103 77

In[3]:= na=pa qa
```

```
Out[3]= 304440358309870155842169300690541367214440941 2\
79297333501375334840521114994335372938106217899400977 5\
749124689032934348385860461362860570610292326992618689\
7078206387761188786017307526546333415258111997
```

Another function allows A to compute the integer d so that $\mathrm{GCD}(d,(p-1)(q-1))$=1:

```
NextNumber[a_Integer,p_Integer,q_Integer]:=
  a /;GCD[a,(p-1)(q-1)] == 1;
NextNumber[a_Integer,p_Integer,q_Integer]:=
  NextNumber[a+1,p,q];
```

For example, person A calculates

```
In[4]:= da=NextNumber[598321984987321654981320654064 98\
406213203210656879089048707810847092873418271849398475\
49340458869697,pa,qa]

Out[4]= 598321984987321654981320654064984062132032106 5\
687908904870781084709287341827184939847549340458869699
```

Finally the integer e needs to satisfy $e\,d \equiv 1\,[\mathrm{mod}\ (p-1)(q-1)]$. This number can be found with the function

```
RSA[d_Integer,p_Integer,q_Integer]:=
  PowerMod[d,-1,(p-1) (q-1)];
```

This function can be used with the previous values to give,

```
In[5]:= ea=RSA[da,pa,qa]

Out[5]= 288270582369002582987033010643813993061701580 1\
510140665454439654288061725102141882942837123058645631\
444262786580239064510916832906556225640733628253487999\
8120264094811433882244502929816800339813046059
```

The number e can also be calculated solving a diophantine equation, because if $e\,d \equiv 1\,[\mathrm{mod}\ (p-1)(q-1)]$ then there exists an integer y that satisfies the equation $e\,d - y(p-1)(q-1) = 1$. To calculate e and y a continued fraction technique can be used, as is described in Henrici [6]. The function that summarizes this method is

```
Diophantine[k_Integer,l_Integer]:=
  Module[{a,su},
  a=Drop[Reverse[Floor[FixedPointList[1/(#1-Floor[#1]) &,
  k/l,SameTest->(Floor[#2] == #2 &)]]],1];
  su=Fold[1/#1+#2 &,a[[1]],Drop[a,1]];
  If[OddQ[Length[a]],Return[
  {Denominator[su],Numerator[su]}],
  Return[{l-Denominator[su],k-Numerator[su]}] ];
  ]
```

To compute the number e we can also set

```
In[6]:= ea=Diophantine[da,(pa-1)(qa-1)][[1]]
Out[6]= 28827058236900258298703301064381399306170158O1\
510140665454439654288061725102141882942837123058645631\
444262786580239064510916832906556225640733628253487999\
8120264094811433882244502929816800339813046059
```

Of course, B computes and stores the numbers pb, qb, nb, db and eb in the same form, and publishes the numbers eb and nb that constitute the public key. Therefore the calculations of B will be

```
In[11]:= pb=NextPrime[49274837459378567382919189378437\
568393958875738393984775849393935753694529739517528415\
44156566516359];
```

```
In[12]:= qb=NextPrime[124903852093485203956802389467O2\
394857023456237495467346235906234973428918911293877243\
78439580249837];
```

```
In[13]:= nb=pb qb;
```

```
In[14]:= db=NextNumber[4893935785696977857634368596964\
736489506583773219643400765914612004074213453983215672\
215739397576943,pb,qb];
```

```
In[15]:= eb=RSA[db,pb,qb];
```

Now we will see how to represent a text by a number. Traditionally to each letter is assigned a two digit number, a $\rightarrow$ 01, b $\rightarrow$ 02, . . . , z $\rightarrow$ 26, and other more typographic characters are introduced. But as we are working with computers we will use the ASCII code. The function that implements this conversion is

```
AsciiToCode[x_Integer]:=
  If[ x<132 && x>31, x-31, 1];
```

We subtract 31 to get only two digit numbers. The number corresponding to a string results from joining the numbers of each character. The function that performs this is

```
StringToNumber[a_String]:=
  Fold[100 #1+#2 &,0,Map[AsciiToCode,
  EliminateSpaces[ToCharacterCode[a]]]]
```

For example, we have

```
In[7]:= number=StringToNumber["How are you?"]
```

```
Out[7]= 418088016683700190808632
```

Also there exist the inverse of these functions; they are

```
CodeToAscii[x_Integer]:=
  If[ x<100 && x>0, x+31, 32];

NumberToString[a_Integer]:=
  FromCharacterCode[Reverse[Map[CodeToAscii[#1-100
  Floor[#1/100]] &,
  Drop[FixedPointList[(Floor[#1/100] &),a],-2]]]];
```

For example, we have

```
In[8]:= NumberToString[number]

Out[8]= How are you?
```

Finally, we need some functions to carry out the codifying and decodifying operations employing the RSA technique. For example, we suppose that B sends a message to A. B has the function:

```
Codify[a_String,e_Integer,n_Integer]:=
  PowerMod[StringToNumber[a],e,n]/;StringToNumber[a] < n;
Codify[a_String,e_Integer,n_Integer]:=
  Print["Message too long"];
```

Every text of interest has a length that exceeds the number n, so we have programmed a function that codifies a message of arbitrary length, breaking it up into pieces of length smaller than n. This function is

```
StringTakeRSA[a_String,i1_Integer,i2_Integer]:=
  StringTake[a,{i1,i2}] /; i2 <= StringLength[a];
StringTakeRSA[a_String,i1_Integer,i2_Integer]:=
  StringTake[a,{i1,StringLength[a]}]
  /; i2 > StringLength[a];

CodifyText[a_String,e_Integer,n_Integer]:=
  Module[{b,i},
  b=Table[StringTakeRSA[a, i, i+Floor[N[Log[10,n]]/2]-1],
  {i,1,StringLength[a],Floor[N[Log[10,n]]/2] } ];
  Return[ Table[Codify[b[[i]],e,n],{i,1,Length[b]}] ];
  ]
```

This function returns a list of numbers, each one corresponding to a part of the message. This operation can be done by everybody, because only the public key is used. For instance, B sends a message to A setting

```
In[9]:= list=CodifyText["Hello A, we have a meeting \
today at 8 p.m., in the usual place. We will deal \
with an important matter for our society. B.",ea,na]
```

```
Out[9]=
{197066004175391169528678282382798190731054979882217 71\
26248967997703167418582692263275660537101802339836 8541\
66659653716595470831759540169725238588600315291908 7594\
92067217397090546947505247671037618007 6,52460777395005\
86038442918676857199703779995296945667630250301798 4399\
22312663193520423435840481511751348526400429486231 1164\
70434437419209671376491875057869380182732225459820 0679\
7102731703029338403707 1}
```

Finally, if B wants to send a message that other persons can't send, it is possible to use the signature technique already given. The function that performs this is

```
CodifyTextWithSignature[a_String,d1_Integer,n1_Integer,
  e2_Integer,n2_Integer]:=
  PowerMod[CodifyText[a,d1,n1],e2,n2];
```

It is worth noticing that one generally has several `CR` (Carriage Return) in each text. We have done a small modification and substitute `CR` by a space, and several consecutive ones are eliminated, resulting in only one, as other good programs do.

The necessary functions to decodify the number or the number list resulting from the previous process are

```
Decodify[a_Integer,d_Integer,n_Integer]:=
  NumberToString[PowerMod[a,d,n]];

DecodifyList[a_List,d_Integer,n_Integer]:=
  StringJoin[Table[Decodify[a[[i]],d,n],{i,1,Length[a]}]];

DecodifyListWithSignature[a_List,e1_Integer,n1_Integer,
  d2_Integer,n2_Integer]:=
  DecodifyList[PowerMod[a,d2,n2],e1,n1];
```

The first function is an auxiliar one, which will be needed for the rest of them. For example, to decodify the previous list of numbers, we use

```
In[10]:= DecodifyList[list,da,na]

Out[10]= Hello A, we have a meeting today at 8 p.m., \
in the usual place. We will deal with an important \
matter for our society. B.
```

Let us see now how to use the signature technique with a practical example. Person B wants to send a message to A that he can read and be sure that it comes from B. B does not know da and B does not know db. Person B must write the following command

```
In[16]:= listsigned=CodifyTextWithSignature["Hello A, \
we have a meeting today at 8 p.m., in the usual place. \
We will deal with an important matter for our society. \
B.",db,nb,ea,na];
```

The person A, to read this message, must do the following:

```
In[17]:= DecodifyListWithSignature
  [listsigned,eb,nb,da,na]

Out[17]= Hello A, we have a meeting today at 8 p.m., \
in the usual place. We will deal with an important \
matter for our society. B.
```

Finally we remark that the package has three 'non–standard' aspects, that is to say, facts that are not included in the general theory of the method. The first is the conversion of letters into numbers; different ways exist to perform this operation. Another aspect is the partition of the string in substrings, which is carried out trying to produce the minimum number of them, adjusting its length to the number n. The third aspect is the sustitution of `CR` by a space and the elimination of repeated spaces. These three aspects can be modified by each user of the package.

References

[1] Akritas, A.G. 1989: *Elements of Computer Algebra with Applications.* John Wiley & Sons.

[2] Brassard, G. and Bratley, P. 1988: *Algorithmics.* Prentice-Hall.

[3] Cormen, T.H., Leiserson, C.E. and Rivest, R.L. 1990: *Introduction to Algorithms.* MIT-Press.

[4] Diffie, W. and Hellman, M. 1976: New Directions in Cryptography. *IEEE Transactions on Information Theory* **22** (1976), 644-654.

[5] Gardner, M. 1989: *Penrose Tiles to Trapdoor Ciphers.* Freeman.

[6] Henrici, P. 1977: *Applied and Computational Complex Analysis*, volume 2. John Wiley & Sons.

[7] Knuth, D.E. 1981: *The Art of Computer Programming*, volume 2, Seminumerical Algorithms, 2d ed. Addison-Wesley.

[8] Manber, U. 1989: *Introduction to Algorithms.* Addison-Wesley.

[9] Rivest, R.L., Shamir, A. and Adleman, L.M. 1978: A method for obtaining digital signatures and public–key cryptosystems. *Communications of the ACM* **21** (1978), 120-126.

[10] Wagon, S. 1991: *Mathematica in Action.* Freeman.