

**Frank Moore, David Milan, Hoai Nam Tran**  
**Math 918 Project**  
**Professor: Jon-Lark Kim**  
**Implementing and Cracking the RSA Cryptosystem**

## 1 Introduction

The primary goal of our project is to implement the RSA cryptosystem in Maple 8. Our implementation includes finding primes suitable for RSA, translating text messages into blocks of large numbers and back, as well as encrypting and decrypting the numerical blocks. We found that our program could quickly encrypt long messages using primes with at least 512 bits and was capable of encrypting messages in a reasonable amount of time using 1024-bit primes. According to the RSA website ([www.rsasecurity.com](http://www.rsasecurity.com)) 1024-bit primes are still considered safe for everyday use. In fact, they offer a \$200,000 reward for the factorization of a certain 2048-bit composite.

The next section of this paper will serve as a quick review of RSA. In section 3, we discuss the various properties that prime numbers should have for secure encryption as described in [4], [2], and [3]. We follow the method of [2] for generating these so-called strong primes. The final section of the paper discusses some methods for factoring  $n$  when  $p$  and  $q$  are not strong primes. The factorization algorithms we programmed were Pollard's  $p - 1$  method and William's  $p + 1$  method, taken from [6] and [7] respectively.

## 2 RSA Cryptosystem

Recall that the public key cryptosystem RSA, introduced in the paper [1], works as follows. Each user of the system first generates large random prime numbers  $p$  and  $q$  subject to the conditions for strong primes in section 3 and forms  $n = pq$ . It is also advisable that  $p$  and  $q$  not be too close together and that  $\gcd(p - 1, q - 1)$  be relatively small for reasons discussed in [1], and [3]. Next the user chooses an invertible  $e$  in  $\mathbb{Z}_{\phi(n)}$  and forms the public key pair  $(e, n)$ . The private key is then the pair  $(d, n)$  where  $d$  is the multiplicative inverse of  $e \bmod \phi(n)$ . Suppose Bob wants to send Alice a message  $\mathcal{P}$ .

Assuming he has a function that embeds plaintext into  $\mathbb{Z}_n$  (for example, the method described in [4]), he simply retrieves Alice's public key  $(e_A, n_A)$  and sends her the message  $\mathcal{C} = \mathcal{P}^{e_A}(\text{mod } n_A)$ . Alice can then decode the message by finding  $\mathcal{C}^{d_A} = \mathcal{P}^{e_A d_A} = \mathcal{P}(\text{mod } n_A)$ . Of course, we have omitted many of the important details of RSA in this description since it has already been covered extensively in class.

### 3 Finding Strong Primes

**Definition 3.1** *A strong prime  $p$  is a prime satisfying:*

1.  $p \equiv 1 \pmod{r}$ , also known as Pollard  $p - 1$  prime,
2.  $p \equiv s - 1 \pmod{s}$ , also known as Williams  $p + 1$  prime,
3.  $r \equiv 1 \pmod{t}$ , where  $r$ ,  $s$ , and  $t$  are all large, random primes of a given number of bits.

The definition of strong prime given above is taken from the proceedings of Eurocrypt [2]. Our question will be to find such  $p$ . Well, we use the following technique: we generate random primes  $s$  and  $t$  of given  $k$  bits as described in the `getKBitPrime` function in our Maple worksheet. In this function, we generate a random number between  $2^{k-1}$  and  $2^{k-1} + 2^{k-2} - 1$ , call it  $a$ . If  $a$  is even, replace  $a$  by  $a + 1$ , and check if  $a$  is prime. If  $a$  is not prime, replace  $a$  by  $a + 2i$ , where  $i$  starts at 1, and increases by 1, until we reach the first prime number. Both  $s$  and  $t$  are easily found via this process.

Once we have  $t$ , we then generate a prime  $r$  that is equivalent to 1 mod  $t$  as follows. In this part, we let  $r = 2Lt + 1$ , where  $L$  starts at  $\lfloor k/2 \rfloor$  and is incremented until a prime value of  $r$  is found.

Finally, we can construct  $p$  based on primes  $r$  and  $s$  as shown in the Maple function `getKBitStrongPrime`. By the following theorem, we know that  $p$  is of the form  $p = s^{r-1} - s^{r-1} + lrs$ , for some  $l$ . If  $p$  is not prime when  $j = 1$ , increase  $j$  by 2, and repeat the process until  $p$  is found.

**Theorem 3.2** *Let  $r$  and  $s$  be odd primes. The prime  $p$  satisfies,*

$$p = 2jr + 1 = 2ks - 1,$$

*for some  $j$  and  $k$  if and only if  $p = s^{r-1} - r^{s-1} + lrs$ , for some  $l$ .*

**Proof:** First note that if  $p = 2jr + 1 = 2ks - 1$ , for some  $j$  and  $k$ , then it also satisfies the weaker condition:  $p = j'r + 1 = k's - 1$ , for some  $j'$  and  $k'$ , i.e.,  $p \equiv 1 \pmod{r}$  and  $p \equiv s - 1 \pmod{s}$ .

( $\Leftarrow$ ) Suppose  $p = s^{r-1} - r^{s-1} + lrs$ . Note that  $r$  and  $s$  are primes, then by Fermat's Little Theorem we have:  $r^{s-1} \equiv 1 \pmod{s}$  and  $s^{r-1} \equiv 1 \pmod{r}$ . Therefore,  $p = s^{r-1} - r^{s-1} + lrs$  implies,  $p \equiv -r^{s-1} \equiv -1 \equiv s - 1 \pmod{s}$ . Similarly,  $p \equiv s^{r-1} \equiv 1 \pmod{r}$ .

( $\Rightarrow$ ) Suppose  $p_1$  and  $p_2$  are integers such that  $p_1 = 1 + ar$ ,  $p_1 = a's - 1$ ,  $p_2 = 1 + br$ , and  $p_2 = b's - 1$ , for some  $a, b, a', b'$ . Now,

$$\begin{aligned} p_1 - p_2 &= (1 + ar) - (1 + br) \equiv 0 \pmod{r} \\ &= (a's - 1) - (b's - 1) \equiv 0 \pmod{s}. \end{aligned}$$

Since  $\gcd(r, s) = 1$ ,  $p_1 - p_2$  is multiple of  $rs$ . And since,  $s^{r-1} - r^{s-1} \pmod{rs} \equiv 1 \pmod{r} \equiv s - 1 \pmod{s}$ ,  $p_1$  and  $p_2$  must be in the form of  $p = s^{r-1} - r^{s-1} + lrs$ , for some  $l$ . ■

## 4 Computing an Example

We present an example of how one could use our Maple code to encrypt and decrypt messages in RSA.

If we want to use RSA, the first thing we should do is create our keys. So we'll need to choose  $n = pq$ . This is done by the `getKBitStrongPrime(k)` procedure. Here  $k$  is a lower bound on the number of digits in the prime. For this example we will use relatively small primes to make the output more readable.

```
p := getKBitStrongPrime(27)[5];
q := getKBitStrongPrime(32)[5];
n := p*q;
phi := (p-1)*(q-1);
```

This returns  $p = 8184762583$ ,  $q = 3433330171979$  as well as the correct values of  $n$  and  $\phi(n)$ . In this case  $\gcd(p, q) = 2$ , so our primes are acceptable for RSA. Now that we have strong primes, we can set up our public key  $(e, n)$  and private key  $(d, n)$  where  $ed \equiv 1 \pmod{\phi(n)}$ . Our algorithm works by searching for a number  $e$  such that  $\gcd(e, \phi(n)) = 1$ . We must first pick a random place to start.

```

# pick a random starting place
start:= rand((phi/2)+1..(phi-1))():
if (type(start, even)) then start:=start+1; end if:

key:= getRSA_e_d(phi,start):
e:= key[1];
d:=key[2];

```

Now we're ready to encrypt! Let's say someone wants to send us a secret message. They just need to look up our public key ( $e, n$ ) and run the encryption function.

```

message:="Hi guys. RSA is so very awesome...isn't it?
         I just wanted to say good job on your project
         and keep up the excellent work.

         Your's Truly, Ronald Linn Rivest.";

```

```

encodeResult := encodeRSA(e,n,message):
codeBlocks:= encodeResult[2]:
encodeResult[1];

```

This returns the encrypted message:

```
M4/0xU7--5 ... ar8y'
```

We can decrypt this message with the procedure `decodeRSA(codeBlocks,d,n)`. This returns the original message:

```

decodeResult := 'Hi guys. RSA is so very awesome...isn't it?
               I just wanted to say good job on your project
               and keep up the excellent work.

               Your's Truly, Ronald Linn Rivest. '

```

## 5 Pollard's P-1 Factoring Method

In 1974, Pollard [6] introduced a method of factoring integers that has been dubbed the  $p - 1$  method. The reason that it has been named this is because the primes divisors that are found using this technique are those primes such that  $p - 1$  has only small prime factors. We now give a description of the method, adapted slightly from [7].

Suppose that  $N$  is a number to be factored, and that  $N$  has a prime factor  $p$  such that:

$$p = \left( \prod_{i=1}^k q_i^{\alpha_i} \right) + 1 \quad (1)$$

where  $q_i$  is the  $i$ th prime and  $q_i^{\alpha_i} \leq B_1$ . Notice that this method rests *heavily* on the fact that  $q_i \leq B_1$  for all  $i$  and as such a suitable  $B_1$  must be chosen. Experimental evidence has shown that fairly large nontrivial factors of  $N$  may be found with  $B_1$  even as low as  $10^4$ . Choose  $R$  such that:

$$R = \prod_{i=1}^k q_i^{\beta_i} \quad (2)$$

and  $\beta_i$  is such that  $q_i^{\beta_i} \leq B_1$  and  $q_i^{\beta_i+1} > B_1$ . Clearly by definition of  $B_1$ ,  $p - 1 \mid R$ . By Fermat's Little Theorem, we have  $a^{p-1} \equiv 1 \pmod{p}$  when  $\gcd(a, N) = 1$ . Therefore, we also have that  $a^R \equiv 1 \pmod{p}$ , and hence  $p \mid \gcd(N, a^R - 1)$ .

Therefore, the first stage of our algorithm is to choose an  $a$  at random in the interval  $2, \dots, N-2$ . Raise this  $a$  to the  $R$  power, and check if  $\gcd(a^R - 1, N) \neq 1$ . If so, this gcd is a nontrivial factor of  $N$ . Pollard also gave in [6] a second stage of this algorithm that we also describe here.

Suppose instead of (1) above, we have:

$$p = s \left( \prod_{i=1}^k q_i^{\alpha_i} \right) + 1 \quad (3)$$

where  $s$  is a prime and  $B_1 < s \leq B_2$ . In this case we have  $p \mid (a_m^s - 1, N)$ . Our goal will be to guess this  $s$ . So, let  $\{s_j : j = 1, 2, \dots, k\}$  be the ordered set of all primes such that  $B_1 < s_j \leq B_2$  and put  $g_j = s_{j+1} - s_j$ . Next, calculate  $b_1 \equiv a_m^{s_1} \pmod{N}$  and define

$$b_{j+1} \equiv a_m^{g_j} b_j \pmod{N}.$$

Now we compute

$$G_t = \gcd\left(\prod_{i=0}^c (b_{i+1} - 1), N\right) \text{ for } t = 1, c + 1, 2c + 1, \dots, [B_2/c]c + 1$$

Since we have  $b_j \equiv a_m^{s_j} \pmod{N}$ , we see that  $p$  must divide some  $G_t$ . Since greatest common divisors are more expensive to calculate than products, you may choose  $c > 1$ . In our implementation, we choose  $c = 10$ , and this may be easily changed.

Since its development in 1974, the  $p - 1$  factoring method has found some very impressive factors. Some of these are due to a variant of the method that was developed for numbers of the form  $N = 2^p - 1$ , where  $p$  is prime, called a Mersenne number. We now describe this variant. It is known that factors of  $N$  must be of the form  $q = 2kq + 1$  where  $k$  is an integer. In order to alter the above first step, note that  $(3, N) = 1$ , by the above fact. Thus, we may compute  $a = 3^{2Rp}$  and this will be equivalent to  $1 \pmod{q}$ , since  $q - 1 \mid 2Rp$ . Therefore,  $\gcd(a - 1, N) = q$ . One factor found this way was the factor  $q = 314584703073057080643101377$ , a divisor of  $2^{2944999} - 1$ . Note that  $q = 2 \cdot 53409984701702289312 \cdot 2944999 + 1$ , and  $b = 53409984701702289312 = 2^5 \cdot 3 \cdot 19 \cdot 947 \cdot 7187 \cdot 62297 \cdot 69061$ . Note that all prime factors of  $b$  are lower than  $10^5$ , so that  $b$  is called a  $10^5$ -smooth number (the sort of numbers vulnerable to the  $p - 1$  test).

## 6 Williams's $P+1$ Factoring Method

Later, in 1982, Williams introduced in [7] a method of factorization called  $p + 1$  factorization. It is analogous in many ways to the method described above. However, in order to develop this method of factorization, we must first introduce the concept of Lucas functions. We note that this method and proof is taken *shamelessly* from the article [7], with minor adaptations.

Let  $P, Q$  be integers and let  $\alpha, \beta$  be zeros of  $x^2 - Px + Q$ . We define the Lucas functions by:

$$U_n(P, Q) = (\alpha^n - \beta^n)/(\alpha - \beta), \quad V_n(P, Q) = \alpha^n + \beta^n. \quad (4)$$

Also, we define  $\Delta = (\alpha - \beta)^2 = P^2 - 4Q$ . There is a rich and deep theory of the properties and identities of these functions, and we will need a theorem of Lehmer and several of these identities for this method.

**Remark 6.1** *Below, we will list several of the identities we will need (where the  $P, Q$  are omitted for brevity)*

$$\begin{cases} U_{n+1} = PU_n - QU_{n-1} \\ V_{n+1} = PV_n - QV_{n-1} \end{cases} \quad (5)$$

$$\begin{cases} U_{2n} = V_n U_n \\ V_{2n} = V_n^2 - 2Q^n \end{cases} \quad (6)$$

$$\begin{cases} U_{2n-1} = U_n^2 - QU_{n-1}^2 \\ V_{2n-1} = V_n V_{n-1} - PQ^{n-1} \end{cases} \quad (7)$$

$$\begin{cases} \Delta U_n = PV_n - 2QV_{n-1} \\ V_n = PU_n - 2QU_{n-1} \end{cases} \quad (8)$$

$$\begin{cases} U_{m+n} = U_m U_{n+1} - QU_{m-1} U_n \\ \Delta U_{m+n} = V_m V_{n+1} - QV_{m-1} = V_n \end{cases} \quad (9)$$

$$\begin{cases} U_{2n} = V_n U_n \\ V_{2n} = V_n^2 - 2Q^n \end{cases} \quad (10)$$

$$\begin{cases} U_n(V_k(P, Q), Q^k) = U_{nk}(P, Q)/U_k(P, Q) \\ V_n(V_k(P, Q), Q^k) = V_{nk}(P, Q) \end{cases} \quad (11)$$

*The above identities are easily verified by substitution from (4), and also using the algebraic facts that  $P = \alpha + \beta$  and  $Q = \alpha\beta$ .*

We first make a reduction of the problem. Note that if  $\gcd(N, Q) = 1$  and  $P' \equiv P^2 Q^{-1} - 2 \pmod{N}$  then if  $\alpha, \beta$  are the roots of  $x^2 - Px + Q$  then  $\alpha/\beta$  and  $\beta/\alpha$  are the roots of  $x^2 - P'x + Q'$ , where  $Q' \equiv 1 \pmod{N}$ . This fact is easily seen by substituting  $\alpha/\beta$  and  $\beta/\alpha$  into the new equation involving  $P'$  and  $Q'$ . Therefore, we have the following fact:

$$U_{2m}(P, Q) \equiv PQ^{m-1}U_m(P', 1) \pmod{N} \quad (12)$$

may be verified by substituting from (4) and using the fact about  $P$  and  $Q$  from above.

Furthermore, we include a theorem of Lehmer [5] that we will need.

**Theorem 6.2** *(Lehmer, [5]) If  $p$  is an odd prime,  $p \nmid Q$  and the Legendre symbol  $\left(\frac{\Delta}{p}\right) = \epsilon$  then*

$$\begin{aligned} U_{(p-\epsilon)m}(P, Q) &\equiv 0 \pmod{p} \\ V_{(p-\epsilon)m}(P, Q) &\equiv 2Q^{m(1-\epsilon)/2} \pmod{p} \end{aligned} \quad (13)$$

We are now able to describe the  $p + 1$  factoring method. Suppose that  $q$  is a prime divisor of  $N$ , and

$$q = \left( \prod_{i=1}^k q_i^{\alpha_i} \right) - 1, \quad (14)$$

where  $q_i$  is again the  $i$ th prime and  $q_i^{\alpha_i} \leq B$ . If we define  $R$  as in (2), we have  $q + 1 \mid R$ . So, if  $(Q, N)$ , and  $\left(\frac{\Delta}{q}\right) = -1$ , then  $q \mid U_R(P, Q)$ , by (13). So, by (6), we have that  $q \mid U_{2R}(P, Q)$ . Thus, from (12), we have that  $q \mid U_R(P', 1)$ . In other words, we lose no generality if we assume  $Q = 1$  (we will make this assumption throughout the rest of the paper). Further, by Lehmer's theorem (13), we also have

$$V_{(q-\epsilon)m}(P, 1) \equiv 2 \pmod{q}; \quad (15)$$

Hence, if  $q \mid U_R(P, 1)$ , then  $q \mid (V_R(P, 1) - 2)$ .

Our first step in the algorithm is to find a  $P$  such that  $(P^2 - 4, N) = 1$ . If we find a  $P$  such that this does not hold, we have found a factor of  $N$  and so we are done. Note that for the theorem to hold, all we need is for  $\left(\frac{P^2-4}{q}\right) = -1$ . However we obviously cannot check this beforehand, since we do not know  $q$ . However, there is a 50% chance that this will hold for each choice of  $P$ , giving a probabilistic method for finding  $q$ , for our method is guaranteed to work if  $q$  satisfies the above hypotheses and  $\left(\frac{P^2-4}{q}\right) = -1$ . We then calculate  $P_R = V_R(P) \pmod{N}$  using the following rules:

$$\begin{cases} V_{2f-1} \equiv V_f V_{f-1} - P \\ V_{2f} \equiv V_f^2 - 2 \\ V_{2f+1} \equiv P V_f^2 - V_f V_{f-1} - P \pmod{N} \end{cases} \quad (16)$$

These rules, along with a binary representation of  $R$  allow us to compute  $P_R$  very rapidly, described as follows (we omit the mention of  $P$  for brevity again). Note first that  $V_0 = 2$  and  $V_1 = P$ , which is easily seen from the definition of  $V_n$ . In addition note that we need only  $V_f$  and  $V_{f-1}$  to compute any of the above identities. Suppose  $R = (\zeta_{k-1} \zeta_{k-2} \dots \zeta_0)_2$  be a binary representation for  $R$ , and define  $R_i = (\zeta_{k-1} \zeta_{k-2} \dots \zeta_{k-i})_2$ . Now we have the rule on pairs  $(V_{R_i}, V_{R_{i-1}})$  from (16):

$$(V_{R_{i+1}}, V_{R_{i+1}-1}) = \begin{cases} (V_{2R_i}, V_{2R_i-1}) & \text{when } \zeta_{k+1} = 0 \\ (V_{2R_i+1}, V_{2R_i}) & \text{when } \zeta_{k+1} = 1 \end{cases} \quad (17)$$

such that  $V_{R_k} = V_R = V_R(P)$ . Lastly, we compute  $d = \gcd(V_R - 2, N)$ . If  $d \neq 1$ , then  $d$  is a non-trivial factor of  $N$ .

## 7 Conclusion and Code Listing

Implementing the RSA Cryptosystem is not too hard in the sense that we do not need to use advanced technological equipment in order to implement it. Indeed, we are able to implement it in a software package as readily available as Maple 8. It is even possible to break RSA if the primes  $p$  and  $q$  are not chosen appropriately. Our included Maple worksheet illustrates a method of choosing our primes in order to avoid possible attack. In addition, the methods we used to attack RSA when the primes are not chosen carefully are Pollard's  $p - 1$  method and William's  $p + 1$  method, which are also included in our Maple worksheet. We conclude our paper with the code listing of the Maple worksheet that performs the RSA encryption/decryption algorithms and the factoring techniques described above. The worksheet may be downloaded from <http://www.math.unl.edu/~fmoore> .

```
prodFirstPrimes := proc(N)
  # This function returns the product of the primes less than N
  local x,i,myPrime;
  i := 1:
  x := 1:
  myPrime := 2:
  while(myPrime < N) do
    i := 1:
    while (myPrime^i <= N) do i := i + 1: od:
    i := i - 1:
    x := x * myPrime^i:
    myPrime := nextprime(myPrime):
  od:
  x
end proc;

pollardPMinusOneMethod := proc(n, B1, B2)
  # This function uses the Pollard P-1 method of factoring to try and factor n
  # This method works only when n has a prime factor p such that p-1 has small prime
  # factors
  local a,k,b,d,i,myPrime,testFactor,currentB,t,c,index,gap,randomNumber:
  i := 0:
  d := 1:
  k := prodFirstPrimes(B1):
  randomNumber := rand(2..n-2):
  while ((d = 1) and (i < 10)) do
    # these first 4 lines are what is called the p-1 first stage in the literature
```

```

# relatively simple
a := randomNumber():
b := a&k mod n:
d := gcd(b - 1, n):
i := i + 1:
# this if block is the second stage p-1 factoring method.  A little more
# complex, yet is a more sophisticated method.
if (d = 1) then
  myPrime := nextPrime(B1):
  t := 1:
  currentB := (b&^(myPrime)) mod N:
  c := 10:
  while ((t <= floor(B2/c)*c + 1) and (d = 1)) do
    testFactor := 1:
    for index from 0 to c do
      testFactor := testFactor * (currentB - 1):
      gap := nextPrime(myPrime) - myPrime:
      currentB := (b&^(gap)*currentB) mod N
    od:
    d = igcd(testFactor, N):
    t := t + c:
  od:
end if:
od:
d
end proc;

lucasVFunction := proc(P,N,r)
# this function computes V_r(P) mod N where V_n is the Lucas Function
# defined by a^n + b^n where a,b are roots of the polynomial
# x^2 - px + 1
local i, V_Zero, V_bigtemp, V_smalltemp, V_big, V_small,numBits,binaryR:
# this is initialized to v_0
V_small := 2:
# this is initialized to v_1
V_big := P:

# convert r to binary and then to a string so we may access the bits individually
binaryR := convert(convert(r, binary), string);
numBits := evalf(log(r)/log(2)):
if (ceil(numBits) > numBits) then numBits := numBits + 1: end if:
# this loop calculates V_r(P) using an algorithm that is polynomial
# in the number of bits of r, using the following recurrences:
# V_(2f-1) = V_f*V_(f-1) - P mod N
# V_(2f) = V_f*V_f - 2 mod N

```

```

#  $V_{(2f-1)} = P \cdot V_f^2 - V_f \cdot V_{(f-1)} - P \pmod N$ 
for i from 2 to numBits do
  V_smalltemp := V_small:
  V_bigtemp := V_big:
  if (binaryR[i] = "1") then
    V_big := (P*V_bigtemp^2 - V_bigtemp*V_smalltemp - P) mod N:
    V_small := (V_bigtemp^2 - 2) mod N:
  else
    V_big := (V_bigtemp^2 - 2) mod N:
    V_small := (V_bigtemp*V_smalltemp - P) mod N:
  end if:
od:
V_big
end proc;

williamsPPlusOneMethod := proc(n,B)
  # This function uses the William's p+1 method of factoring an integer n.
  # This method works when n has a prime divisor p such that p+1 has small
  # prime divisors. This function returns a factor.
  local k,pnot,loopCount, result:
  loopCount := 1:
  # this finds the pnot such that  $(pnot^2 - 4) = 1$ 
  pnot := 3:
  while (igcd(pnot^2 - 4, n) <> 1) do
    pnot := pnot + 1:
  od:
  k := prodFirstPrimes(B):
  result := igcd(lucasVFunction(pnot,n,k) - 2, n):
  # loop until we have found a factor. Don't perform this
  # loop more than 10 times though.
  while ((result = 1) and (loopCount < 10)) do
    loopCount := loopCount + 1:
    # get the next pnot
    pnot := pnot + 1:
    while (igcd(pnot^2 - 4, n) <> 1) do
      pnot := pnot + 1:
    od:
    result := igcd(lucasVFunction(pnot,n,k) - 2, n):
  od:
  result
end proc;

getKBitPrime := proc(k)
  # This function returns a prime number of at least k bits
  local a,i:

```

```

i := 0:
a := rand(2^(k-1)..2^(k-1) + 2^(k-2) - 1)():
if (type(a, even)) then a := a + 1; end if;
while (not(isprime(a))) do
  i := i + 1:
  a := a + 2:
od:
# note the return value is a pair. i is the number of times isprime() is called
# and a is the prime of at least k bits
[i,a]
end proc;

getKBitStrongPrime := proc(k)
# This function returns a prime number of at least k bits that is deemed
# a strong prime. This means that p is such that t = p-1 has a large prime
# factor, p+1 has a large prime factor, and t's large prime factor has
# another large prime factor.
local r,s,t,p,pnot,L,i,retList,j:
# get a large prime s of at least floor(k/2) bits
retList := getKBitPrime(floor(k/2)):
s := retList[2]:
i := retList[1]:
# get a large prime t of at least floor(k/2) bits
retList := getKBitPrime(floor(k/2)):
t := retList[2]:
i := i + retList[1]:
# this algorithm creates a prime r that is equivalent to 1 mod t (i.e. t divides r-1)
L := floor(k/2):
r := 2*L*t + 1:
while (not(isprime(r))) do
  i := i + 1:
  L := L + 1:
  r := 2*L*t + 1:
od:
# The paper states that a prime that such that p+1 has factor s and p-1 has factor r
# must be of the form p = pnot + 2*j*r*s where pnot is defined below.
pnot := (s^(r-1) - r^(s-1)) mod r*s:
if (type(pnot, even)) then pnot := pnot + r*s end if:
j := 1:
p := pnot + 2*j*r*s:
while (not(isprime(p))) do
  i := i + 1:
  j := j + 1:
  p := pnot + 2*j*r*s:
od:

```

```

    # Note that the return value is again a list,
    # with the last value is the desired prime p
    [i,r,s,t,p]
end proc;

getRSA_e_d := proc(phi,start)
    local x,i,e,d:
    for i from start to phi-1 do
        x := i:
        if (igcdex(i,phi,'d','blah') = 1) then
            e := i:
            break:
        end if:
    od:
    [e mod phi,d mod phi]
end proc;

encoder := (P,e,n) -> (P&^e) mod n:

decoder := (C,d,n) -> (C&^d) mod n:

encodeRSA := proc(e,n,message)
    local blockSize, encMessage, messageLength, numBlocks, encMessage2, theBlocks,
        theEncodedBlocks, encodedMessage, i, j, morbledMessage;
    blockSize := floor(evalf(log(sqrt(n))/log(256)))-1;
    encMessage := convert(message, bytes); messageLength := nops(encMessage);
    numBlocks := ceil(messageLength/blockSize);
    encMessage2 := array(1..blockSize*numBlocks);
    for i from 1 to messageLength do
        encMessage2[i] := encMessage[i]:
    od:
    for i from messageLength+1 to blockSize*numBlocks do
        encMessage2[i] := 32:
    od:
    theBlocks := array(1..numBlocks);
    for i from 1 to numBlocks do
        theBlocks[i] := encMessage2[(i-1)*blockSize+1]:
        for j from 1 to blockSize-1 do
            theBlocks[i] := theBlocks[i] + (256^j)*encMessage2[(i-1)*blockSize+1+j]:
        od:
    od:
    theEncodedBlocks := convert(map(encoder, theBlocks,e,n),array);
    encodedMessage := map(convert,map(convert,theEncodedBlocks,base,256),bytes);
    morbledMessage := '';
    for i from 1 to numBlocks do

```

```

        morbledMessage := cat(morbledMessage, encodedMessage[i]):
    od:
    [morbledMessage, theEncodedBlocks]
end proc;

decodeRSA := proc(theEncodedBlocks, d, n)
    local blockSize,decodedMessage, demorbledMessage,i;
    blockSize := max($op(2,theEncodedBlocks()));
    decodedMessage :=
        map(convert,map(convert,map(decoder,theEncodedBlocks,d,n),base,256),bytes);
    demorbledMessage := '';
    for i from 1 to blockSize do
        demorbledMessage := cat(demorbledMessage, decodedMessage[i]):
    od:
    demorbledMessage
end proc;

```

## References

- [1] L. M. Adleman, R. L. Rivest, and A. Shamir, *A method for obtaining digital signatures and public-key cryptosystems*, Communications of the ACM, **21** (1978), 120-126.
- [2] J. A. Gordon, *Strong primes are easy to find*, Advances in Cryptology, Proceedings of Eurocrypt 1984, Springer, 1985, 216–223.
- [3] D.E. Knuth, *The Art of Computer Programming Volume 2, Seminumerical Algorithms*, 3rd edition, Addison-Wesley, 1999.
- [4] Neal Koblitz, *A Course in Number Theory and Cryptography*, 2nd edition, Springer-Verlag, GTM 114, 1994.
- [5] D. H. Lehmer, *An extended theory of Lucas' functions*, Ann. of Math (2), **31** (1930), 419-448.
- [6] J. M. Pollard, *Theorems on factorization and primality testing*, Proc. Cambridge Philos. Soc., **76** (1974), 521–528.
- [7] H. C. Williams, *A  $p+1$  method of factoring*, Mathematics of Computation, **39** (1982), 225-234.