Algorithm 121

RSA KEY CALCULATION IN ADA

D. G. N. HUNTER
Standard Telecommunication Laboratories Ltd, London Road, Harlow, Essex

RSA encryption programs are inaccessible for several reasons – first they impose requirements on multiple precision arithmetic packages which are not met in general-purpose ones, especially the exponentiation operation in finite arithmetic; secondly, they are partially written in assembly code for speed, and rightly so; finally, they are being obscured by recently discovered short cuts. This paper describes one way of calculating RSA keys and includes some Ada procedures to illustrate the essential arithmetic; it is intended as a starting point for the construction of packages, especially those for 16-bit microcomputers.

1. RSA ALGORITHMS

The RSA algorithm[1] appears to be the best of the public-key encryption methods, although the computing effort needed is so high that special-purpose hardware will be essential for bulk data encryption. The algorithm deals in terms of long numbers and converts a message, M, into a cipher, C, using the equation

$$C = M^e \bmod n \tag{1}$$

and back again, using the equation

$$M = C^d \bmod n \tag{2}$$

where all numbers are typically 150 decimal digits long. It seems likely that special hardware will limit the number size to 512 bits, or 154 decimal digits, but even so it would still take three million years of computer time to break the code when the computer is fast enough to perform a million 'factorisation steps' per second. In the equations given above the values of e and n are publicly available, but d is secret. The modulus, n, is the product of two primes, p and q, which must also be secret since otherwise d could be derived from the published value of e, as we shall see.

The security of the method therefore depends on the difficulty of factorising n, and further conditions must be placed on p and q to thwart various factorising algorithms. These conditions are that p and q must differ in length by a few digits; that $p-1$ and $q-1$ must each have a large prime factor, say p' and q'; and that $p+1$ and $q+1$ must each have a large prime factor. Discussion of this last condition is deferred until the next paragraph. Moreover, in order to defeat an iterative attack, $p'-1$ and $q'-1$ must also each have a large prime factor, say p'' and q''. Such 'doubly safe'[2] primes, for example p, can be found straightforwardly in three steps: first a sequence of random odd large integers is tested for primality until p'' is found; then the sequence of numbers $2*j*p''+1$, for $j = 1, 2, 3, \ldots$, is tested until the prime, p', is found; finally the sequence of numbers $2*k*p'+1$, for $k = 1, 2, 3, \ldots$, is tested until p is found. The average number of tests needed at each step is equal to $0.5*\ln(p)$ which is much the same as the number of decimal digits in p. The computing effort for all this is about equivalent to 20 encryptions, but is rather variable since the standard deviation of the number of composite numbers rejected during a search is also $0.5*\ln(p)$. In passing it should be noted that this variability is a nuisance when constructing keys of a particular size. Some control over the size can be exerted by starting the final search from a k value larger than one.

It is scarcely feasible to seek a prime, p, such that both $p-1$ and $p+1$ have a prime factor of the order of p because of the rarity of suitable cases (1 in 10000). Williams and Schmid[3] describe a key calculation process where $p-1$ and $p+1$ have a prime factor of the order of $sqrt(p)$, thus gaining protection against one factorising algorithm at the expense of another. An equivalent process would be to find two primes, u and v, of the order of $sqrt(p)$ and such that $u-1$ and $v-1$ each have a large prime factor, and then to seek primes in the sequence $2*j*u*v+c$ for $j = 1, 2, 3, \ldots$, where c is a suitable starting value. A value for c can be found from Algorithm X[4], which terminates with $c = 1-2*u*ul$ or $c = 1-2*v*u2$ according as $u1$ or $u2$ is negative. In any case there is a 70% chance that a large number has a prime factor greater than its own square root.

The rest of the key consists of the pair of numbers d and e which are multiplicative inverses of each other in arithmetic modulo $(p-1)*(q-1)$, and can be found with much less computing effort. First d is chosen as an odd number of the same order of magnitude as n and such that both

$$\begin{aligned} &d < r \quad \text{where} \quad r = lcm(p-1,q-1) \\ &\qquad\qquad = (p-1)*(q-1)/\gcd(p-1,q-1) \quad \text{and} \\ &\gcd(d,(p-1)*(q-1)) = 1 \end{aligned} \tag{3}$$

The value of e can be found using Algorithm X where $u3 = r$, $v3 = d$ at the start, and $e = v2$ or $v2-r$ when $v3$ is found to equal 1 at the end, but since this needs an extra program to be written it is simpler to evaluate it from[5]

$$e = d^{\mathrm{phi}(r)-1} \bmod r \tag{4}$$

where phi is Euler's totient function. If r has factors $x^a y^b \ldots$ then $\mathrm{phi}(r) = (x^a - x^{a-1})*(y^b - y^{b-1})\ldots$; the factorisation of r is known from the k values found during the calculation of p and of q, hence e can be calculated with one exponentiation. However Algorithm X must be used if primes p and q have been made via u and v to satisfy both the $p-1$ and $p+1$ conditions, since it is infeasible to factorise $p-1$ or $q-1$.

Recent work indicates that there are faster ways of performing the algorithms than those given here, some involving special hardware. Norris and Simmons[6] describe how to compute $xy \bmod n$ in time proportional to $\ln(n)$ rather than $\ln(n)$ squared by using a precomputed value of y/n, and Quisquater and Couvreur[7] manage to avoid explicit divisions using a combination of techniques including a lookup table. They also mention a way of shortening decryption by performing two half-length exponentiations (taking one-eighth of the time each), exploiting the knowledge of p and q.

2. IMPLEMENTATION

Multiple precision arithmetic packages[8–12] have usually been constructed for scientific purposes, for example to compute accurate coefficients of a power series for inclusion in a library procedure for exp or ln, or to create benchmark numbers[13] to test a computer's floating point accuracy. By contrast, RSA algorithms only need integer working, using numbers bounded by a modulus of typically 100 or 200 decimal digits, and can therefore use a simpler internal representation for numbers. Although calculation time is a problem, since nearly 10^9 high-level language statements might be executed per key produced, data storage space requirements are modest, in contrast to other integer problems such as seeking large Mersenne primes.

The exponentiation function in finite arithmetic is peculiar. One is so accustomed to thinking about left-to-right evaluation that it is natural, when faced with the expression A^B modulo C,

to imagine that the modulo C operation gets applied to the result of A^B. However, even problems like 4444^{4444} modulo 9 often fail on systems which do this and only provide exponentiation for a small range of numbers, for example APL. Finite exponentiation is an inseparable operation involving its three inputs.

Multiple precision arithmetic

The essential point about multiple precision working is to notice that successive computer words can be regarded as holding the digits of a number in a representation using a large base, b. The choice of b is important, and the best for RSA encryption appears to be half the capacity of a computer word, that is to say one bit less than the word length, for reasons given below.

While it is true that input and output are easier if b is a power of ten, say $b = 100$, the processing speed achievable varies as the square of the number of bits used to represent b, therefore b should be made as large as possible. If no facility exists to generate double precision quantities, and a decimal base is chosen, it is necessary for a computer word to be able to hold the value of b^2, or even $3*(b^2)$ in some packages, which drastically reduces the effective processing speed.

On the other hand, if b occupies the whole word, the hardware multiply instruction on a twos-complement machine fails to provide a product containing all the required bits, but will do so if b is reduced by one bit. Thus a natural representation is one where all words are stored as positive integers and where temporary negative values have a leading word which is negative. This still leaves the problems of computing the double-length exact product of two positive integers and of finding the remainder from a double-length dividend and a single-length divisor. High-level languages have paid little attention to this requirement, and the Ada solution offered here is intended as a guide for preparing an assembly-coded external procedure in a practical program.

There is another reason for choosing a binary representation, and this is that exponentiation is particularly simple: the exponent is scanned from the left until its leading bit is found; thereafter a combined squaring and multiplication operation is done for each '1', and a squaring for each '0'. For example x can be raised to the 19th power in the following steps starting from an initial value of 1:

operation	resulting power
sq mult	1
sq	2
sq	4
sq mult	9
sq mult	19

After each squaring or multiplication the working value is brought within range of the modulus by an explicit modulus operation, although it is sufficient, during an exponentiation, to bring it within range of a normalized modulus until the final step, which saves unnecessary normalisations. An alternative algorithm[4] allows the exponent to be scanned from the right. The corresponding algorithms for an exponent which is not in binary[4] need more code and workspace, and run slower.

Although the quotient of a long division is not needed, the modulus of a double-length number with respect to a single-length one is required, and this involves forming the quotient as a by-product. There have been several proposals for division where only the leading word or two of the dividend and divisor are visible. In all these methods the idea is to obtain a 'trial divisor digit', in our case a whole computer word, and then to subtract this multiple of the quotient from the dividend. The result of dividing the first two dividend words by the leading word of the normalised divisor is a trial divisor digit which is either exact, or one too large, or two too large with probabilities of 0.6754, 0.3178 and 0.0078 respectively for a sufficiently large base,[14, 15] as is true for a computer word. The simplest algorithm would therefore add back the divisor, if necessary, once or occasionally twice after the trial subtraction. Howell[16] rounds up the leading divisor digit before use (avoiding the division entirely in the overflow case) and adjusts the answer until it is in range; Mifsud[17] uses the first two divisor digits and 'pre-adjusts' the operands before the division proper; Knuth[4] refines the trial digit by a test involving the third dividend digit and succeeds not only in making the trial digit exact in almost all cases, but also in excluding the 'two too large' case. This method is given in the appendix.

If input and output are done in hexadecimal there are no conversion problems, but a number-terminating convention is needed anyway; with grouped digits for readability, the final group can be followed by a letter. For decimal external representation, input presents no special problem but output is harder. One approach is to copy a number to the top half of a double-length number which is then regarded as a mixed integer and fraction, with the fraction in the lower half and a guard word beyond the fraction half. The mixed number is rounded and then divided repeatedly by a suitable power of ten until the integer part is seen to be zero. Finally the digits of the converted number are recovered by repeated multiplication. The power of ten is chosen to give the maximum number of decimal digits for the word size, and care must be taken to force leading zeros where they are appropriate – with or without the help of the high-level language. Alternatively, the groups of digits are calculated from the least significant end by forming the remainders when the given number is divided by the power of ten.

Primality testing

The preparation of an RSA key involves the construction of large random primes which can be found by a probabilistic test, as described below. Most of the running time for such a test is spent doing a modular exponentiation and varies as the cube of the length of the numbers. Hence, as a rule of thumb, the time to find a prime varies as the fourth power of its length, since the likelihood of an odd number being prime varies nearly inversely as its length. However, recent short cuts appear to reduce this to a third-power law.

Rivest, Shamir and Adleman suggest the Solovay and Strassen[18] (SS) test. The chance of a composite number passing the test is less than 0.5 (0.37 in practice) so that for a prime to be found with a confidence level of at least 0.999999 the SS test must be applied less than 20 times.

A simpler test is described by Knuth[4] under the name Algorithm P and is given in the appendix. The chance of a composite number passing it is, at worst, 0.25, but as low as one in ten million in practice, and hence less than 10 successful tests would show a number to be prime with the same confidence level of 0.999999; in fact 5 tests are adequate. Thanks to the good reject rate for Algorithm P, it is almost always executed only once for a composite number whereas about 1.5 SS tests are needed.

The search phase can be improved by first dividing by a few small primes, in order to weed out some composite numbers, since this takes scarcely any time. Assuming that trial division takes no time at all, it is not difficult to show how many primes must be tried in order to speed up the search phase by a given factor:

Number of trial prime divisors	Improvement factor	
3	2	
8	3	
21	4	
54	5	(reasonable choice)
141	6	
372	7	
995	8	
2697	9	
7397	10	

3. PROGRAM

The program was written in Ada, because it is readable and provides dynamic actual arrays, and it runs on an Intel iAPX432. It was developed from an Algol 60 program using the normal I/O on an Elliott 903 and assembly-coded procedures for the product and modulus operations. Numbers are represented as arrays of Ada-type short_integer, 16 bits, and the double precision operations of addition, subtraction, multiplication and division are performed using working values of type long_integer, 32 bits; it is important that a long_integer can accommodate the product of two short_integers, therefore, for example, a 24 bit long_integer will not do. Although the hardware can do all the necessary arithmetic on words twice as long as are used here, the Ada compiler does not yet allow access to these facilities. No attempt has been made to exploit the parallelism possible in Ada, but multiple processors could be used here, perhaps one per multiplier word and one per result word.

The essential procedures are listed in the appendix, but those for input/output and operator prompts are not shown. The Wichmann–Hill[19] random number generator was used. The user is expected to provide enough array space to accommodate p', given p'', or p given p', and there is no check on this. In a practical program it is worthwhile to keep the user informed of progress, for example by printing a message for every ten primality tests, as this overcomes to some extent the sense of impatience with the inherent variability of run time.

As mentioned above, the encryption and key calculation times vary, as a rule of thumb, with the third and fourth power of the number of bits in the numbers. If T is the add time for a computer with a word length of 16 bits, and which includes integer multiply and divide hardware, then the encryption and key calculation times for a program in assembly code can be expected to be TB^3 and $TB^4/30$ where there are B (B around 512) bits in the key. However, the time would be greater for a program as given in the appendix and where only the first four procedures are converted into assembly code – perhaps by a factor of two or three.

4. CONCLUSIONS

The effective speed of a computer for RSA encryption varies as the square of the bit width of its arithmetic unit. That is why the special chips under development, which are able to form the 1,024-bit product of two 512–bit numbers, are so much better than conventional computers. The best internal representation for a number on a twos-complement binary computer is an array of binary integers with the sign bit zero in all words except, temporarily, in the leading word where it is necessary to represent a negative result. Encryption time varies as the cube, and key calculation time as the fourth power, of the key length, unless recent short cuts are adopted. Key calculation time is typically equivalent to about 20 encryptions.

The program excerpt given in the appendix is largely based on Knuth's algorithms for multiple precision arithmetic. It is intended as a framework for programs to be written in assembly code, especially for 16-bit computers.

Acknowledgements

The author thanks STL Ltd for permission to publish this paper. Thanks are due also to Mr D. W. Davies for many helpful discussions and to Mr W. R. Gardner for assistance with the iAPX432.

REFERENCES

1. R. L. Rivest, A. Shamir and L. Adelman, A method of obtaining digital signatures and public-key cryptosystems, *Communications of the ACM* **21** (2), 120–126 (1978).
2. S. Berkovits, Factoring via superencryption, *Cryptologia* **6** (3), 229–237 (1982).
3. H. C. Williams and B. Schmid, Some remarks concerning the M.I.T. public-key cryptosystem, *BIT* **19**, 525–538 (1979).
4. D. E. Knuth, *The Art of Computer Programming, Seminumerical Algorithms*, Addison-Wesley, Reading, Mass., 2nd ed.
5. D. W. Davies, Private communication.
6. M. J. Norris and G. J. Simmons, Algorithms for high-speed modular arithmetic, *Congressus Numerantium* **31**, 153–163 (1981).
7. J. J. Quisquater and C. Couvreur, Fast decipherment algorithm for RSA public-key cryptosystems, *Electronic Letters* **18** (4), 905–907 (1982).
8. W. T. Wyatt, D. W. Lozier and D. J. Orser, A portable extended precision arithmetic package and library with Fortran precompiler, *ACM Transactions on Mathematical Software* **2** (3), 209–231 (1976).
9. J. K. Foderaro and K. L. Sklower, *The FRANZ LISP Manual*, University of California, Berkeley (1981).
10. I. D. Hill, Procedures for the basic arithmetical operations in multiple-length working (Algorithm 34), *The Computer Journal* **11** (2), 232–235 (1968).
11. P. Hammersley, Note on Algorithm 34, *The Computer Journal* **12** (1), 103–104 (1969).
12. J. L. Schonfelder and J. T. Thomason, Arbitrary precision arithmetic in Algol 68, *Software Practice and Experience* **9**, 173–182 (1979).
13. B. A. Wichmann and J. Du Croz, A program to calculate the GAMM measure, *The Computer Journal* **22** (4), 317–322 (1979).
14. G. E. Collins and D. R. Musser, Analysis of the Pope-Stein division algorithm, *Information Processing Letters* **6** (5), 151–155 (1977).
15. D. A. Pope and M. L. Stein, Multiple precision arithmetic, *Communications of the ACM* **3**, 652–654 (1960).
16. K. M. Howell, Multiple precision arithmetic techniques, *The Computer Journal* **9** (4), 383–387 (1967).
17. C. J. Mifsud, A multiple-precision division algorithm, *Communications of the ACM* **13** (11), 666–668 (1970).
18. R. Solovay and V. Strassen, A fast Monte-Carlo test for primality, *SIAM Journal of Computing* **6** (1), 84–85 (1977); **7**, 118 (1978).
19. B. A. Wichmann and I. D. Hill, An efficient and portable pseudo-random number generator, *Applied Statistics* **31** (2), 188–190 (1982).

APPENDIX

```
package RSA_BCS is
   type vector is array ( short_integer range<> ) of short_integer;
-- MASK returns the logical AND of two short_integers
   function MASK(A,B: short_integer) return short_integer;
-- RANDOM is described in reference 19. X,Y,Z initially set by user.
   X: short_integer:=540; Y: short_integer:=955; Z: short_integer:=68;
   function RANDOM return float;
-- These four procedures provide the equivalent of the normal
-- facilities seen by a programmer working in assembly code,
-- except that BORROW is always positive.
   procedure CODE_ADD(CARRY,RESULT: in out short_integer;
                      X: short_integer);
   procedure CODE_SUBTRACT(BORROW,RESULT: in out short_integer;
                      X: short_integer);
   procedure CODE_MULTIPLY(HI,LO: out short_integer;
                      MAND,MIER: short_integer);
```

```
        procedure CODE_DIVIDE(QUOT, REMAINDER: out short_integer;
                              HI,LO,DIVISOR: short_integer);
--      These procedures operate on numbers of type vector.
        procedure CLEAR(A: out vector);
        procedure DOUBLE(A: in out vector);
        function  EQUAL_ZERO(X: vector) return boolean;
        procedure HALVE(A: in out vector);
        procedure GENERATE(X: in out vector; N: vector); -- see ref. 19.
        procedure NORMALISE(A: in out vector; K: out short_integer);
--      These multiply or divide a vector by a short_integer.
        procedure MPY(A: in out vector; SCAL: short_integer);
        procedure DIV(A: in out vector; SCAL: short_integer);
--      These provide the basic vector operations. ADD, SLEW_ADD and
--      SUBTRACT can deal with negative numbers.
        procedure ADD(A: in out vector; B: vector);
        procedure SLEW_ADD(A: in out vector;
                           B: vector; S: short_integer);
        procedure SUBTRACT(A: in out vector; B: vector);
        procedure MULTIPLY(A: in out vector;                    -- see ref. 4 (M)
                           B,C: vector);
        procedure MODULUS(X,A,B: in out vector);                -- see ref. 4 (D)
--      These are directly used by the RSA algorithms.
        procedure GCD(X: in out vector; U,V: vector);
        procedure POWER(X: in out vector;
                        M,E: vector; N: in out vector);
        function  PRIME(N: vector) return boolean;              -- see ref. 4 (P)
        procedure BUILD_PRIME(W: short_integer; P: in out vector);
        procedure BUILD_SUCCESSOR(W: short_integer; P: vector;
                                  PP: in out vector; FP: out short_integer);
        procedure BUILD_KEY(W: short_integer; P,Q: in out vector;
                            FP,FQ: short_integer;
                            DD,EE: out vector; N: in out vector);
    end RSA_BCS;
-- Package body follows.
    with text_io, unchecked_conversion; use text_io;
    package body RSA_BCS is
        type long_integer is new integer;
        BITS_IN_USE: constant short_integer: = 15;
        HALF: constant short_integer: = 2**(BITS_IN_USE - 1);
        MAX_INT: constant short_integer: = 2*(HALF - 1) + 1;
        TRIALS: short_integer;
        WORD: constant long_integer: = long_integer(HALF + HALF);
        function MASK(A,B: short_integer) return short_integer is
        -- returns the logical AND of two short_integers
        type bool_array is array ( 1..1 + BITS_IN_USE) of boolean;
        AA,BB,C: bool_array;
        function INWARD is new unchecked_conversion
                                  (short_integer, bool_array);
        function OUTWARD is new unchecked_conversion
                                  (bool_array, short_integer);
        begin
            AA: = INWARD(A); BB: = INWARD(B); C: = AA and BB; return OUTWARD(C);
        end;
        function RANDOM return float is
        -- See reference 19. X,Y,Z are assigned to before first use.
        W: float;
        begin
            X: = 171 * (X mod 177) - 2 * (X/177);
            Y: = 172 * (Y mod 176) - 35 * (Y/176);
            Z: = 170 * (Z mod 178) - 63 * (Z/178);
            if X<0 then X: = X + 30269; end if; if Y<0 then Y: = Y + 30307; end if;
            if Z<0 then Z: = Z + 30323; end if;
            W: = float(X)/30269.0 + float(Y)/30307.0 + float(Z)/30323.0;
            return W + 0.5 - float(short_integer(W));
        end;
        procedure CODE_ADD(CARRY,RESULT: in out short_integer;
                           X: short_integer) is
        -- generates sum with carry;
        TEMP: long_integer;
        begin
            TEMP: = long_integer(RESULT) + long_integer(X);
            if TEMP > = WORD then
                TEMP: = TEMP - WORD; CARRY: = CARRY + 1;
            end if;
            RESULT: = short_integer(TEMP);
        end;
        procedure CODE_SUBTRACT(BORROW,RESULT: in out short_integer;
                                X: short_integer) is
-- generates difference with borrow;
TEMP: long_integer;
begin
    TEMP: = long_integer(RESULT) - long_integer(X);
    if TEMP < 0 then
        TEMP: = TEMP + WORD; BORROW: = BORROW + 1;
    end if;
    RESULT: = short_integer(TEMP);
end;
procedure CODE_MULTIPLY(HI, LO: out short_integer;
                        MAND, MIER: short_integer) is
-- generates accurate product;
TEMP: long_integer;
begin
    TEMP: = long_integer(MAND) * long_integer(MIER);
    HI: = short_integer(TEMP/WORD);
    LO: = short_integer(TEMP rem WORD);
end;
procedure CODE_DIVIDE(QUOT, REMAINDER: out short_integer;
                      HI, LO, DIVISOR: short_integer) is
-- generates quotient and remainder;
TEMP: long_integer;
begin
    TEMP: = long_integer(HI)*WORD + long_integer(LO);
    QUOT: = short_integer(TEMP/long_integer(DIVISOR));
    REMAINDER: = short_integer(TEMP rem  long_integer(DIVISOR));
end;
procedure CLEAR(A: out vector) is
-- sets A to zero;
begin
    for J in A'range loop A(J): = 0; end loop;
end;
procedure DOUBLE(A: in out vector) is
-- doubles A;
OCARRY,ICARRY: short_integer: = 0;
begin
    for J in reverse A'range loop
        CODE_ADD(OCARRY,A(J),A(J)); A(J): = A(J) + ICARRY;
        ICARRY: = OCARRY; OCARRY: = 0;
    end loop;
end;
function EQUAL_ZERO(X: vector) return boolean is
-- true if X = 0;
begin
    for J in X'range loop if X(J) /= 0 then return false; end if;
    end loop;
    return true;
end;
procedure GENERATE(X: in out vector; N: vector) is
-- generates a random non-zero X less than N;
TEMP: float: = float(MAX_INT);
begin
    for J in X'range loop
        X(J): = short_integer(TEMP*RANDOM);
    end loop;
    for J in N'range loop
        if N(J) = 0 then X(J): = 0;
                    else
            while X(J) > = N(J) or (J = N'last and X(J) = 0) loop
                TEMP: = float(N(J)); X(J): = short_integer(TEMP*RANDOM);
            end loop;
            exit;
        end if;
    end loop;
end;
procedure HALVE(A: in out vector) is
-- halves A;
IBIT, OBIT, TEMP: short_integer: = 0;
begin
    for J in A'range loop
        TEMP: = A(J)/2;
        if A(J) = 2*TEMP then OBIT: = 0; else OBIT: = HALF; end if;
        A(J): = TEMP + IBIT; IBIT: = OBIT; OBIT: = 0;
    end loop;
end;
procedure NORMALISE(A: in out vector; K: out short_integer) is
-- normalises A so that leading word > = base/2;
KK: short_integer: = 0;
begin
    while A(1)<HALF loop
        KK: = KK + 1; DOUBLE(A);
    end loop;
    K: = KK;
end;
procedure MPY(A: in out vector; SCAL: short_integer) is
-- vector times word;
CARRY: short_integer: = 0; HI, LO: short_integer;
begin
    for J in reverse A'range loop
        CODE_MULTIPLY(HI,LO,A(J),SCAL); CODE_ADD(HI,LO,CARRY);
        A(J): = LO; CARRY: = HI;
    end loop;
end;
procedure DIV(A: in out vector; SCAL: short_integer) is
-- vector divided by word;
REMAINDER: short_integer: = 0; HI, LO: short_integer;
begin
    for J in A'range loop
        CODE_DIVIDE(HI, LO, REMAINDER, A(J), SCAL);
        A(J): = HI; REMAINDER: = LO;
```

```
   end loop;
end;
procedure ADD(A: in out vector; B: vector) is
-- vector addition;
ICARRY, OCARRY: short_integer:=0;
begin
    for J in reverse 2..B'last loop
        CODE_ADD(OCARRY,A(J),B(J)); CODE_ADD(OCARRY,A(J),ICARRY);
        ICARRY:=OCARRY; OCARRY:=0;
    end loop;
    A(1):=A(1) + B(1) + ICARRY;
end;
procedure SLEW_ADD(A: in out vector; B: vector;
                   S: short_integer) is
-- vector addition with offset;
ICARRY, OCARRY: short_integer:=0;
begin
    for J in reverse 2..B'last loop
      CODE_ADD(OCARRY,A(J+S),B(J)); CODE_ADD(OCARRY,A(J+S),ICARRY);
      ICARRY:=OCARRY; OCARRY:=0;
    end loop;
    A(S+1):=A(S+1) + B(1) + ICARRY;
end;
procedure SUBTRACT(A: in out vector; B: vector) is
-- vector subtraction;
BOR1, BOR2: short_integer:=0;
begin
    for J in reverse 2..B'last loop
      CODE_SUBTRACT(BOR2,A(J),B(J)); CODE_SUBTRACT(BOR2,A(J),BOR1);
      BOR1:=BOR2; BOR2:=0;
    end loop;
    A(1):=A(1) - B(1) - BOR1;
end;
procedure MULTIPLY(A: in out vector; B,C: vector) is
-- vector multiplication;
HI, LO, CARRY: short_integer;
begin
    for J in A'range loop A(J):=0; end loop;
    for J in reverse B'range loop
        CARRY:=0;
        for I in reverse C'range loop
            CODE_MULTIPLY(HI,LO,B(I),C(J)); CODE_ADD(HI,A(I+J),LO);
            CODE_ADD(HI, A(I+J), CARRY); CARRY:=HI;
        end loop;
        A(J):=CARRY;
    end loop;
end;
procedure MODULUS(X,A,B: in out vector) is
-- X:= A modulo B, where A is twice as long as B;
BOR1, BOR2, FACT, HI, K, LO: short_integer;
U, V: vector(1..3);
begin
    NORMALISE(B,K);
    if B(1) <= A(1) then SUBTRACT(A,B); end if;
    if A(1) < 0 then ADD(A,B); end if;
    for L in B'range loop
        if B(1) = A(L) then FACT:=MAX_INT;
                       else CODE_DIVIDE(FACT,LO,A(L),A(L+1),B(1));
        end if;
        U(1):=A(L); U(2):=A(L+1); V(1):=0; V(2):=B(1);
        if B'last > 1 then U(3):=A(L+2); V(3):=B(2);
                      else U(3):=0; V(3):=0; end if;
        MPY(V,FACT); SUBTRACT(U,V);
        if U(1) < 0 then FACT:=FACT-1; end if;
        BOR1:=0;  BOR2:=0;
        for J in reverse B'range loop
            CODE_MULTIPLY(HI,LO,B(J),FACT);
            CODE_SUBTRACT(BOR1,A(L+J),LO); CODE_ADD(BOR2,HI,BOR1);
            CODE_SUBTRACT(BOR2,A(L+J-1),HI); BOR1:=BOR2; BOR2:=0;
        end loop;
        if BOR1 = 1 then
            A(L):=0; A(L+1):=
                short_integer(long_integer(A(L+1)) - WORD);
            SLEW_ADD(A,B,L);
        end if;
    end loop;
    for J in B'range loop X(J):=A(J+B'last); end loop;
    for J in 1..K loop
        HALVE(B); SUBTRACT(X,B);
        if X(1) < 0 then ADD(X,B); end if;
    end loop;
end;
procedure GCD(X: in out vector; U,V: vector) is
-- X:= GCD(U,V);
UU, VV: vector(U'range); K: short_integer;
begin
    UU:=U; VV:=V;
    while not EQUAL_ZERO(VV) loop
        K:=0; X:=UU;
        if X(1) > MAX_INT then
            NORMALISE(VV,K); SUBTRACT(X,VV);
            if X(1) < 0 then ADD(X,VV); end if;
        else
<<LEFT_SHIFT>>
            SUBTRACT(X,VV);
            if X(1) < 0 then ADD(X,VV);
                        else DOUBLE(VV); K:=K+1; goto LEFT_SHIFT;
            end if;
        end if;
        for J in 1..K loop
            HALVE(VV); SUBTRACT(X,VV);
            if X(1) < 0 then ADD(X,VV); end if;
        end loop;
        UU:=VV; VV:=X;
    end loop;
    X:=UU; return;
end;
procedure POWER(X: in out vector; M,E: vector;
                                  N: in out vector) is
-- X:= M**E modulo N;
K: short_integer;
C: vector(M'range); D: vector(1..2*M'last);
FINISH: boolean:=false; ONE_BIT: boolean;
begin
    CLEAR(X); NORMALISE(N,K);
    for J in E'range loop
        if E(J) /= 0 then
            X(J):=HALF;
            while MASK(X(J),E(J)) = 0 loop
                X(J):=X(J)/2;
            end loop;
            goto FOUND;
        end if;
    end loop;
    X(X'last):=1; return;
<<FOUND>>
    CLEAR(C); C(C'last):=1; ONE_BIT:=true;
    while not FINISH loop
        if ONE_BIT then
            MULTIPLY(D,C,M); MODULUS(C,D,N);
        end if;
        if X(X'last)=1 then
            for J in 1..K loop
                HALVE(N); SUBTRACT(C,N);
                if C(1) < 0 then ADD(C,N); end if;
            end loop;
            X:=C; FINISH:=true;
                  else
            MULTIPLY(D,C,C);MODULUS(C,D,N);HALVE(X);ONE_BIT:=false;
            for J in X'range loop
                ONE_BIT:=ONE_BIT or MASK(X(J),E(J)) /= 0;
            end loop;
        end if;
    end loop;
end;
function PRIME(N: vector) return boolean is
-- Algorithm P preceded by a few trial divisions;
J,K: short_integer;
TRIAL: array(1..20) of short_integer:=
        (  3,  5,  7, 11, 13, 17, 19, 23, 29, 31,
          37, 41, 43, 47, 53, 59, 61, 67, 71, 73);
Q, X, ONE, Y, NN, NM1: vector(N'range);
Z: vector(1..N'last+1); D: vector(1..2*N'last);
begin
    NN:=N;
    for I in TRIAL'range loop
        if N'last=1 and N(1)=TRIAL(I) then return true; end if;
        for L in N'range loop Z(L):=N(L); end loop;
        Z(Z'last):=0; DIV(Z,TRIAL(I));
        if Z(Z'last)=0 then return false; end if;
    end loop;
    GENERATE(X,N); CLEAR(ONE); ONE(ONE'last):=1;
    NM1:=N; SUBTRACT(NM1,ONE); Q:=NM1; K:=0;
    while MASK(Q(Q'last),1)=0 loop
        K:=K+1; HALVE(Q);
    end loop;
    POWER(Y, X, Q, NN); J:=0;
    while J < K loop
        if J=0 then
            if Y=ONE then return true; end if;
        end if;
        if Y=NM1 then return true; end if;
        if J > 0 and Y=ONE then return false; end if;
        J:=J+1;
        if J < K then
            MULTIPLY(D,Y,Y); MODULUS(Y,D,NN);
        end if;
    end loop;
```

```
        return false;
    end;
    procedure BUILD_PRIME(W: short_integer; P: in out vector) is
    - - finds random prime;
    NT: short_integer; SUCCESS: boolean: = false;
    TEMP: float: = float(MAX_INT);
    begin
        while not SUCCESS loop
            for J in P'range loop
                P(J): = short_integer(TEMP*RANDOM);
            end loop;
            P(P'last): = MASK(P(P'last),MAX_INT - 1) + 1; NT: = 1;
            while NT < = TRIALS loop
                SUCCESS: = PRIME(P);
                if SUCCESS then NT: = NT + 1; else exit; end if;
            end loop;
        end loop;
    end;
    procedure BUILD_SUCCESSOR(W: short_integer; P: vector;
                PP: in out vector; FP: out short_integer) is
    - - finds successor prime;
    NG: short_integer: = 0; NT: short_integer;
    SUCCESS: boolean: = false; ONE: vector(1..W);
    begin
        CLEAR(ONE); ONE(ONE'last): = 1;
        while not SUCCESS loop
            NG: = NG + 1; PP: = P; MPY(PP,2*NG); ADD(PP,ONE); NT: = 1;
            while NT < = TRIALS loop
                SUCCESS: = PRIME(PP);
                if SUCCESS then NT: = NT + 1; else exit; end if;
            end loop;
        end loop;
        FP: = 2*NG;
    end;
    procedure BUILD_KEY(W: short_integer; P,Q: in out vector;
                                FP,FQ: short_integer;
                                DD,EE: out vector; N: in out vector) is
- - Generates D,E and N
    PQ, SHORT_ONE: vector(1..W);
    PHI, R, D, E, LONG_ONE: vector(1..2*W);
    TEMP1,TEMP2: long_integer; SUCCESS: boolean: = false;
    function T(MM: long_integer) return long_integer is
- - Euler's Totient function
    TM, TMOLD: long_integer: = 1; M: long_integer: = MM;
    F: long_integer;
    begin
        F: = 2;
        while F*F < MM or M > 1 loop
            while M mod F = 0 loop
                M: = M/F; TM: = TM * F;
            end loop;
            if TM /= TMOLD then TM: = TM - (TM/F); TMOLD: = TM; end if
            F: = F + 1;
        end loop;
        if TM = 1 then return (MM - 1); else return (TM); end if;
    end;
    begin
        CLEAR(SHORT_ONE); SHORT_ONE(SHORT_ONE'last): = 1;
        CLEAR(LONG_ONE); LONG_ONE(LONG_ONE'last): = 1;
        MULTIPLY(N,P,Q); SUBTRACT(P,SHORT_ONE);
        SUBTRACT(Q,SHORT_ONE); MULTIPLY(PHI,P,Q); GCD(PQ,P,Q);
        for J in 1..PQ'last - 1 loop
            if PQ(J) /= 0 then
                PUT("BAD SEED CHOICE"); NEW_LINE; return;
            end if;
        end loop;
        R: = PHI;
        if PQ(PQ'last) /= 1 then DIV(R,PQ(PQ'last)); end if;
- - Equation 3. Given R now seek a D such that GCD(D,(P - 1)*(Q - 1)) = 1
        while not SUCCESS loop
            GENERATE(D,R); D(D'last): = MASK(D(D'last),MAX_INT - 1) + 1;
            GCD(E,D,PHI); SUCCESS: = E = LONG_ONE;
        end loop;
        DIV(P,FP); DIV(Q,FQ);
- - Totient function value now is (P' - 1)*(Q' - 1)*T(FP*FQ/GCD(P - 1,Q - 1))
        SUBTRACT(P,SHORT_ONE); SUBTRACT(Q,SHORT_ONE);
        MULTIPLY(PHI,P,Q); TEMP1: = long_integer(FP);
        TEMP2: = long_integer(FQ); TEMP1: = TEMP1*TEMP2;
        TEMP2: = long_integer(PQ(W)); TEMP1: = T(TEMP1/TEMP2);
- - Must factorise TEMP1 and offer each factor in turn to MPY
        TEMP2: = 2;
        while TEMP1 > 1 loop
            while TEMP1 mod TEMP2 = 0 loop
                TEMP1: = TEMP1/TEMP2; MPY(PHI,short_integer(TEMP2));
            end loop;
            TEMP2: = TEMP2 + 1;
        end loop;
        SUBTRACT(PHI,LONG_ONE); POWER(E,D,PHI,R);
        DD: = D; EE: = E;
    end;
end RSA_BCS;
```