Algorithms for software implementations of RSA

A. Selby
C. Mitchell

Indexing term: Algorithms

Abstract: Two new algorithms that facilitate the implementation of RSA in software are described. Both algorithms are essentially concerned with performing modular arithmetic operations on very large numbers, which could be of potential use to applications other than RSA. One algorithm performs modular reduction and the other performs modular multiplication. Both algorithms are based on the use of look-up tables to enable the arithmetic computations to be done on a byte by byte basis.

1 Introduction

The purpose of this paper is to describe two new algorithms for use in constructing software implementations of the RSA cryptosystem. These algorithms will enable software implementations to run faster than was previously possible. As is well known, RSA is a potentially extremely valuable cryptographic technique. However, it has the major disadvantage that fast implementations are notoriously difficult to construct, whether they are in hardware or software. Any new algorithms which speed up RSA implementations are therefore of great potential value.

The RSA algorithm is very simple to state. To use it one first needs to choose a value N, typically 512, which determines the security level at which the algorithm will operate. The implementation of RSA relies on computing

$$m^e \text{ (modulo } s)$$

where s has an N-bit binary representation, and

$$m, e < s$$

In fact, s is always chosen to be the product of two $(N/2)$-bit primes. When the values of the primes are known by the user, the RSA computation can be carried out by performing two computations of the type

$$a^b \text{ (modulo } c)$$

using values for a, b and c of $(N/2)$ bits each (see, for example, Quisquater and Couvreur, [1]). When the primes are not known, it is almost always true that the exponent e can be arranged to have a small number of bits (say 32); see, for example, Reference 2.

Paper 6612E (C1, C2), first received 23rd December 1987 and in revised form 17th January 1989

C. Mitchell is with Hewlett Packard Laboratories, Filton Road, Stoke Gifford, Bristol BS12 6QZ, United Kingdom

A. Selby is with Trinity College, Cambridge CB2 1TQ, United Kingdom

Therefore, in practice it is rarely if ever necessary to perform a 'full' N-bit exponentiation. However, when N is large, e.g. $N = 512$, it is still very time consuming to do the computation for $(N/2)$-bit numbers.

The values e and s form part of the encryption and decryption keys, and are therefore fixed for a number of encryption operations. The value m is dependent on the information being encrypted (or decrypted) and therefore varies from computation to computation.

The 'standard' method for doing the modular exponentiation is by using the well known 'repeated square and multiply' technique (see, for example, References 3 or 4). The 'left to right' form of this algorithm involves repeatedly squaring and multiplying by a 'local' fixed value (modulo a 'global' fixed value). The 'local' fixed value is dependent on m, i.e. is fixed for the duration of the encryption computation. The 'global' fixed value, i.e. the modulus, depends on the key and is therefore fixed for a number of encryption operations. We use this notion of local and global fixed values throughout this paper.

The algorithms described are concerned with speeding up these modular arithmetic operations. They both take advantage of local and global fixed values to precompute 'look-up' tables whose use speeds the individual calculations. They both also work on a block by block basis, i.e. they make use of the fact that computers operate on strings of several bits at a time, rather than on individual bits.

In the remainder of this paper we describe the two algorithms in more detail. For each algorithm we first give an informal, verbal description, and then give a more precise, pseudo-code formulation. The form of these descriptions has been chosen to make actual implementations easy to construct. For both algorithms we also give proofs of correctness and brief complexity analyses.

In the descriptions of the algorithms we use the following notational convention. We suppose all numbers are stored in 32-bit words, and we denote the individual words of the $32n$-bit number a by

$$a[0], a[1], \ldots, a[n-1]$$

where $a[0]$ contains the least significant word and $a[n-1]$ the most significant word. Hence

$$a = a[0] + 2^{32} \cdot a[1] + 2^{64} \cdot a[2] + \cdots + 2^{32(n-1)} \cdot a[n-1]$$

We then write $a[\ ]$ for the collection $a[0], a[1], \ldots, a[n-1]$. We also write T for $2^{32}n$. Finally, note that although both algorithms are described in terms of 8-bit bytes and 32-bit words, they would work equally well with other byte and word lengths.

2 Algorithm A

2.1 Informal description

Algorithm A is concerned with 'modular reduction'. Given a $64n$-bit number k and a $32n$-bit modulus d it outputs a $32n$-bit number k', where

$$k' \equiv k \text{ (modulo } d)$$

To do this it uses a table of values pre-computed using the modulus d. This table does not need to be re-evaluated for each computation since it is dependent on a global fixed value. In the context of RSA, algorithm A is useful when computing modular squares. Using ideas given in Reference 3, squaring can be made considerably faster than a square-mod. method which reduces modulo d as it works out the answer, so it is quicker to use a fast non-modular squaring algorithm followed by algorithm A.

The idea of the algorithm is to reduce the length of k by 8 bits (i.e. one byte) at a time. At the beginning of each step it is assumed that k is $4n + i + 1$ bytes long, (where i ranges from $4n - 1$ down to 0), together with an extra bit at the most significant end which may be 1 or 0 (this is 'left over' from the previous iteration). At each step of the algorithm, the largest multiple of d that can be safely subtracted from k is subtracted. This process uses the table *atab*, set up in advance. By safely we mean the largest multiple of d that can be subtracted to leave the result positive, given that only the most significant 9 bits of k are examined. This results in a value of k which has the byte under consideration set to either 0 or 1 (this single bit forming the 'left over' bit for the next subtraction).

In more detail, the table *atab* consists of 512 entries of the form

$$atab[i], \quad 0 \leqslant i \leqslant 511,$$

where each entry consists of an $(n + 1)$-word multi-precision integer. In the precomputation phase, $atab[i]$ is set to the unique integer multiple gd of d defined so that

$$int(gd/T) = i - 1$$

and

$$int((g + 1)d/T) = i$$

and there, as throughout this paper, $int(x)$ denotes the unique integer s satisfying

$$s \leqslant x < s + 1$$

The computation of this table may be achieved by a very simple combination of additions and comparisons; no multiplications or divisions are required.

As previously noted, in this paper we describe a version of the algorithm which operates on 8-bit bytes and thereby requires *atab* to have

$$2^{(8+1)} = 512$$

entries. If one modified the algorithm to reduce by w bits at a time then *atab* would need to contain $2^{(w+1)}$ entries.

In the main algorithm, *atab* is used to compute a value k' satisfying

$$k' \equiv k \text{ (modulo } d)$$

and

$$0 \leqslant k' \leqslant T + 2d - 1$$

As before, we suppose that k is stored in $2n$ 4-byte words, i.e. $8n$ bytes, where the bytes are labelled $0, 1, \ldots, 8n - 1$ and byte 0 is the least significant. The following two steps are repeated $4n$ times, for a value of i descending from $4n - 1$ to 0:

(*a*) Examine bytes $4n + i + 1$ and $4n + i$. Byte $4n + i + 1$ will be set to either zero or one, and thus the value j obtained by regarding the byte pair as an integer will satisfy $0 \leqslant j \leqslant 511$.

(*b*) Subtract 2^{8i} times $atab[j]$ from k. This will have the effect of clearing byte $4n + i + 1$ and resetting byte $4n + i$ to either zero or one.

2.2 Pseudo-code description

Input: $k[\ ]$ (a $2n$-word number)
$d[\ ]$ (an odd n-word constant, where $d \geqslant T/2$. Note that both the constraints on d are imposed merely to simplify the description of the algorithm. They may both be removed by making small modifications to the algorithm).

Output: $k[\ ]$ (an n-word number congruent to the original $k[\ ]$ modulo d)

Method: Prior to performing individual computations it is assumed that a $(512(n + 1))$-word array

$$atab[i][j] \quad (0 \leqslant i \leqslant 511, 0 \leqslant j \leqslant n)$$

has been set up as follows:

```
atab[0][ ] := 0;
for i := 1 to 511 do
   {Let g be the unique integer satisfying
      int(g · d/T) = i − 1   and   int((g + 1) · d/T) = i;
   atab[i][ ] := gd}
```

where, as described above, $atab[i][\ ]$ represents the number stored in the $n + 1$ words

$$atab[i][0], atab[i][1], \ldots, atab[i][n]$$

Note that, for each i, $g = int(T \cdot i/d)$; the description above is used to emphasise that no divisions are involved in the computation of *atab*.

The individual computation now proceeds as follows:

```
for i := 4n − 1 to 0 step −1 do
   {j := int(k[ ]/(2^{8i} · T));
   k[ ] := k[ ] − 2^{8i} · atab[j][ ]}
```

To complete the process it may be necessary to subtract either d or $2d$ from the final k in order to ensure that it is less than T, which is sufficient for intermediate results in modular exponentiation. At most one further subtraction of d will ensure that the result is less than d.

Note that the two operations in each iteration of the algorithm are both very simple. The first, although apparently involving a division and an exponentiation, actually merely involves examining the two most significant bytes of k.

Finally note that during the computation of $atab[\ ][\ ]$,

$$d/T \geqslant 1/2$$

and hence

$$g < 1022 \text{ for every } i$$

Thus $g \cdot d$ (where d is stored in $d[0], \ldots, d[n - 1]$) can be stored in at most $n + 1$ 32-bit words. This explains why $atab[\ ][\ ]$ has the dimensions allocated to it.

2.3 Proof of correctness

We need to establish three points in order to prove the correctness of the algorithm. These points are as follows:

$S1$ The result of the subtraction step in the algorithm will always be positive.

$S2$ The value of j in the algorithm will always be at most 511.

$S3$ The final result is congruent to the original value of k (modulo d). In addition, the final value needs at most $2d$ subtracted from it in order to obtain a result lying in the range 0 to $T - 1$.

Before giving proofs for these three statements we first establish some inequalities. By definition

$$atab[j][\ ] = gd$$

where

$$int(gd/T) = j - 1 \quad \text{and} \quad int((g + 1)d/T) = j$$

Hence

$$gd/T < j \leqslant (g + 1)d/T$$

Therefore we have

$$atab[j][\ ] < Tj \leqslant atab[j][\ ] + d \tag{1}$$

In addition, by definition we know that

$$T/2 \leqslant d < T \tag{2}$$

We can now give proofs for $S1$, $S2$ and $S3$.

Proof of S1: We need to prove that the result of the program step

$$k[\ ] := k[\ ] - 2^{8i} \cdot atab[j][\ ]$$

will always be positive. If we denote the result of the step by k', then, by eqn. 1:

$$k - k' < 2^{8i}Tj.$$

By the previous program step

$$j = int(k/(2^{8i}T)) \leqslant k/(2^{8i}T)$$

Hence

$$k - k' < k,$$

i.e. $k' > 0$

Proof of S2: Now j is defined by the program step

$$j := int(k[\ ]/(2^{8i} \cdot T))$$

We first consider the maximum value of k in this expression. As above, if we denote the result of the program step

$$k[\ ] := k[\ ] - 2^{8i} \cdot atab[j][\ ]$$

by k', then, by eqn. 1:

$$k - k' \geqslant 2^{8i}(Tj - d)$$

By the previous program step,

$$j = int(k/(2^{8i}T)) > k/(2^{8i}T) - 1$$

Hence

$$k - k' > k - 2^{8i}T - 2^{8i}d,$$

i.e.

$$k' < 2^{8i}(d + T) < 2^{8i+1}T \quad \text{(by eqn. 2)}$$

This gives us an upper bound on the value of k in the expression for j (noting that the value of i in the bound is one more than the value of i in the expression for j). Hence

$$j = int(k/(2^{8i}T)) \leqslant k/(2^{8i}T) < 2^{8(i+1)+1}T/(2^{8i}T)$$

i.e.

$$j < 2^9 = 512$$

Proof of S3: It is straightforward to see that the final k is congruent to the original one modulo d, since at each step the value of k is modified by the subtraction of a multiple of d. Using the above argument we see that the final k satisfies the bound

$$k < 2^{8i+1}T \quad (\text{for } i = 0)$$

i.e. the final k is less than $2T$. By eqn. 2 at most $2d$ will need to be subtracted from the final k to achieve a value bounded above by T. This completes the proof of correctness.

2.4 *Complexity analysis*

There are three main elements to an analysis of complexity for Algorithm A: the space required for the tables, the time required to compute the tables, and the time taken to perform one iteration of the algorithm. We consider these aspects in turn.

2.4.1 Space for tables: As we discussed above, the tables for Algorithm A require a total of $2048(n + 1)$ bytes.

2.4.2 Computation time for tables: Suppose that the time taken to add together two 32-bit words is h. Suppose also that the time taken to compare two 32-bit words is e (we use this notation throughout). Since we observed that

$$g < 1022 \quad \text{for every } i$$

the computation of the table requires at most

$1021(n + 1)$ word additions.

Similarly, generating the table requires at most

512 word comparisons.

This is because

$$atab[i + 1] - atab[i]$$

is always either d or $2d$, and to decide which requires at most one comparison of the most significant word.

Hence the total time required to generate the tables is bounded above by approximately

$$2^9(2(n + 1)h + e).$$

2.4.3 Computation time for algorithm: Each step of the algorithm requires a single 'long' subtraction, i.e. $(n + 1)$ word subtractions. Since there are $4n$ steps, the time complexity of the algorithm is

$$4n(n + 1)h$$

where it is assumed that a word subtraction takes the same amount of time to perform as a word addition.

3 Algorithm *B*

3.1 *Informal description*

Algorithm B is concerned with modular multiplication by a fixed value. Suppose we wish to compute

$$a \cdot b(\text{modulo } d),$$

where a, b and d are all $32n$-bit numbers. The algorithm uses a precomputed table dependent on a and d. This approach is useful since, in the context of RSA, the value a can be made a local constant through use of the 'left to right' form of the square and multiply algorithm. Algorithm B is a generalisation of an algorithm due to Quisquater and Couvreur [1].

The basic idea behind the algorithm is to precompute a table of multiples of a (modulo d), which can then be added together according to the value of b. We start by examining the contents of the pre-computed table *btab*. This table contains

$$(32n/c)(2^c - 1)n \text{ words}$$

with entries of the form

$$btab[i][j], \quad 0 \leqslant i < 32n/c, 0 \leqslant j < 2^c - 1$$

and where each entry contains n words. The value of c dictates the size of the table and can be chosen to be any integer dividing 32. The choice of c will be implementation dependent. During the precomputation phase $btab[i][j]$ is set to

$$2^{ci}(j + 1)a \quad (\text{modulo } d).$$

To explain the algorithm's use of *btab* in more detail it is perhaps simplest to examine the case $c = 1$ first. In this case $btab[\ ][\ ][\ ]$ simply contains $2^i \cdot a$ (modulo d) for every i in the range $0, 1, \ldots, 32n - 1$. The algorithm consists of looking at every bit of b (in its binary representation) in turn, and if it is 1 then adding the appropriate multiple of a (from $btab[\ ][\ ][\ ]$) to the partial product. This case of the algorithm corresponds to procedure MODMULCO of Quisquater and Couvreur[1].

This idea can be generalised to the case where more than one bit of b is processed at a time. This is precisely what the above algorithm does, and the value given to c is equal to the number of bits of b processed in one iteration of the algorithm.

From this it can be seen that the larger the value chosen for c, the faster the main part of the algorithm will run, although the precomputation of $btab[\ ][\ ][\ ]$ will both take longer and require more storage. For RSA implementations, since a is only a local constant, this means that the time and space required for the precomputation has to be balanced very carefully against the time taken for the main algorithm to run.

3.2 Pseudo-code description

Input: $a[\ ]$ (an n-word constant)
$b[\ ]$ (an n-word number)
$d[\ ]$ (an n-word constant)

Output: $p[\ ]$ (an n-word number congruent to $a \cdot b$ (modulo d) with the property that $0 \leqslant p < d$). Note that during the computations array $p[\ ]$ will need to be up to $n + 1$ words long.

Method: Prior to performing individual computations it is assumed that a $((32n/c) \cdot (2^c - 1) \cdot n)$-word array

$$btab[i][j][k] \quad 0 \leqslant i < (32n/c), \quad 0 \leqslant j < 2^c - 1, 0 \leqslant k < n$$

has been set up as follows. Note that c is initially chosen to be a small integer dividing 32.

$btab[0][0][\ ] := a[\ ];$

for $j := 1$ to $2^c - 2$ do

$btab[0][j][\ ] := btab[0][0][\ ] + btab[0][j - 1][\ ];$

for $i := 1$ to $32n/c - 1$ do

$\{btab[i][0][\ ] := btab[i - 1][0][\ ]$
$+ btab[i - 1][2^c - 2][\ ];$

for $j := 1$ to $2^c - 2$ do

$btab[i][j][\ ] := btab[i][0][\ ] + btab[i][j - 1][\ ]\}$

where all the additions take place modulo d. At any stage this only ever involves subtracting a single multiple of d from the sum.

Note that the above routine has the effect of setting

$$btab[i][j][\ ] = 2^{ci} \cdot a \cdot (j + 1)(\text{modulo } d)$$

for each i and j. The importance of the above algorithm is that it uses only add and subtract operations, and hence $btab[\ ][\ ][\ ]$ can be computed quickly.

The individual computation now proceeds as follows (where $(b_{32n-1}b_{32n-2} \cdots b_1b_0)_2$ is the binary representation of b):

$p[\ ] := 0$

for $i := 0$ to $32n/c - 1$ do

$\{j := (b_{ci+c-1}b_{ci+c-2} \cdots b_{ci})_2 - 1;$

if $(j \geqslant 0) p[\ ] := p[\ ] + btab[i][j][\ ]\}$

Finally, reduce $p[\ ]$ modulo d using iteration $i = 0$ of algorithm A (noting that, although its exact length will depend on c, the final value of $p[\ ]$ from the above algorithm will usually not contain more than $4n + 1$ 8-bit bytes, since for most applications $32n/c < 512$).

3.3 Proof of correctness

Suppose

$$y_i = (b_{ci+c-1}b_{ci+c-2} \cdots b_{ci})_2 - 1$$

Then it should be clear that

$$b = \sum_{i=0}^{32n/c-1} (y_i + 1)2^{ci}. \tag{3}$$

By examining the description above it then follows that the value computed by algorithm B is

$$p = \sum_{i=0}^{32n/c-1} btab[i][y_i]$$

$$= \sum_{i=0}^{32n/c-1} 2^{ci}a(y_i + 1) \text{ (modulo } d)$$

$$= a \cdot b \text{ (by eqn. 3).}$$

3.4 Complexity analysis

As with algorithm A, we give an analysis of complexity for algorithm B in three parts: the space required for the tables, the time required to compute the tables, and the time taken to perform one iteration of the algorithm. We consider these aspects in turn.

3.4.1 Space for tables: As we discussed above, the table for algorithm A requires a total of

$$(32n/c)(2^c - 1)n \text{ 32-bit words,}$$

i.e. a total of

$$128n^2(2^c - 1)/c \text{ bytes}$$

3.4.2 Computation time for tables: Using the algorithm described above, it is straightforward to see that it takes a total of

$$128n^2(2^{c-1} - 1)/c \text{ word additions}$$

to compute; hence the total time to compute the tables is bounded above by

$$2^{c+6}n^2h/c.$$

3.4.3 Computation time for algorithm: Each step of the algorithm requires a single 'long' addition, i.e. n word additions. Since there are $32n/c$ steps, the time complexity of the algorithm is

$$2^5n^2h/c.$$

4 References

1 QUISQUATER, J.-J., and COUVREUR, C.: 'Fast decipherment algorithm for RSA public-key cryptosystem', *Electron. Lett.*, 1982, **18,** pp. 905–907

2 ZIMMERMAN, P.: 'A proposed standard format for RSA cryptosystems', *IEEE Comput. Magazine*, 1986, pp. 21–34

3 KNUTH, D.E.: 'The art of computer programming, Volume 2: Seminumerical algorithms' (Addison-Wesley, USA, 1981, 2nd edition)

4 BEKER, H.J., and PIPER, F.C.: 'Cipher systems' (van Nostrand, London, 1982)