Exponentiation cryptosystems on the IBM PC

by P. G. Comba

Several cryptosystems based on exponentiation have been proposed in recent years. Some of these are of the public key variety and offer notable advantages in cryptographic key management, both for secret communication and for message authentication. The need for extensive arithmetic calculations with very large integers (hundreds of digits long) is a drawback of these systems.

This paper describes a set of experimental programs that were developed to demonstrate that exponentiation cryptosystems can be efficiently implemented on the IBM Personal Computer (PC). The programs are organized into four layers, comprising procedures for: multiple precision integer arithmetic, modular exponentiation, prime number generation and testing, and cryptographic key generation. The major emphasis of the paper is on methods and techniques for improving execution speed. The items discussed include: the use of a specialized squaring procedure; a recursive splitting method to speed up squaring and multiplication; the computation of residues by using multiplication instead of division; the efficient encoding of residue information; and the use of thresholds to select the most effective primality testing algorithm for a given size number. Timing results are presented and discussed. Finally, the paper discusses the advantages of a mixed system that combines the superior key management capabilities inherent in public key cryptosystems with the much higher bulk-encryption speed obtainable with the Data Encryption Algorithm.

The need for data security has grown steadily over the years, paralleling growth in the use and interconnectivity of computers. Users require the protection of data from unauthorized access and alteration. System experts have drawn on the discipline of cryptography to meet the increasing needs for data security. In the simplest terms, cryptography is a technique for scrambling and disguising information so as to make it appear meaningless or unintelligible. The scrambling, or coding, process requires the use of a secret key, which is also needed to recover the original text from the scrambled version. A system that uses such a technique is referred to as a *symmetric cryptosystem.* An example is the Data Encryption Algorithm (DEA) defined in the *Data Encryption Standard,*[1] which is widely used in banking and transaction processing.

Public key cryptosystems

A revolution in cryptography took place as a result of the publication in 1976 of a paper by Diffie and Hellman[2] that introduced the concept of a *public key cryptosystem.* The novel feature of the proposed system was that different keys are used for *encrypting* and for *decrypting.* Thus, for example, an individual *A*, who is a potential recipient of encrypted messages, generates a public key as well as a private key, and publishes the public key (e.g., in a directory), while

keeping the private key secret. Another individual B, who wants to communicate secretly with A, encrypts the message with A's public key. After receiving the message, A decrypts it with A's private key. For this system to work it must be impossible to derive, from a public key, the corresponding private key.

Once the concept of a public key cryptosystem was introduced, many different specific systems were proposed and studied. References 3 and 4 describe these systems. Before addressing the technical aspects of these systems, their relevance to the key management problem is briefly discussed.

Key management. The key management problem arises from the fact that a symmetric cryptosystem, while providing protection to the data that have been encrypted, still requires that a key be exchanged secretly between any pair of communicating parties. If one visualizes a large network comprising thousands of workstations and terminals and possibly millions of messages, it becomes clear that the safe exchange of keys (and their replacement if they become compromised) can be a staggering problem. The solution made possible by a public key cryptosystem consists of a two-tiered arrangement, where the data to be communicated are encrypted using DEA (for example), and the DEA keys are encrypted using the public keys. The public keys can be freely exchanged and posted in directories.

An actual key management system may involve more complexities and possibly more tiers, but it rests on the concepts outlined above, namely separation of data-encrypting keys from key-encrypting keys, and exchange of keys via the public key system.

Outline of some exponentiation cryptosystems. Several public key cryptosystems have been proposed that involve the following elements:

- The cryptographic key, or part of it, is either a large prime number n, or the product $n = p \times q$ of two large prime numbers p and q.
- The encryption process entails raising a number to a power and taking the results modulo n, that is, dividing the results by n and using the remainder of the division.

The three systems discussed in this paper are now briefly outlined.

RSA system. The best known exponentiation cryptosystem is the one proposed in 1978 by Rivest, Shamir, and Adleman (see Reference 5) usually referred to by their initials as RSA. Under RSA, a typical size for the modulus n would be 200 decimal digits, which is equivalent to 664 bits or 42 personal computer (PC) 16-bit words. The public or encryption key is made up of two parts: the number n (but not its factors p and q) and an exponent e. A message to be encrypted is represented as a number M, which must be less than n, and the process of encryption is represented by the calculation

$$C = M^e \bmod n.$$

To decrypt, the recipient must be in possession of a suitably chosen secret decryption key d; with it the original text can be recovered by executing

$$M = C^d \bmod n.$$

Rabin system. Another similar system, attributed to Rabin,[6] uses the number 2 as the encryption exponent and the knowledge of p and q to do decryption. This system has some drawbacks for general use, but may be appropriate for special applications.

ElGamal system. Considerable interest has been generated by a public key system described by ElGamal.[7] This system requires n to be a prime number, rather than the product of two primes. The number n, as well as a properly chosen integer α, are known to all users. In addition, each individual user has a private key x and a public key y defined by

$$y = \alpha^x \bmod n.$$

To send a secret message M, one first generates a random number k and computes $K = y^k \bmod n$; the ciphertext consists then of the pair of numbers (c_1, c_2) defined by

$$c_1 = \alpha^k \bmod n, \qquad c_2 = (K \times M) \bmod n.$$

To decrypt, one computes successively

$$K = c_1^x \bmod n, \qquad G = K^{-1} \bmod n,$$
$$M = (c_2 \times G) \bmod n.$$

The challenge of exponentiation on the PC. Modular exponentiation is essential for encryption/decryption in the systems outlined above. In addition, the generation of cryptographic keys involves the search for large prime numbers; this search requires certain tests that make heavy use of modular exponentiation. Hence modular exponentiation plays a central role in the investigations reported here.

At the start of the work reported in this paper, it was not known whether the extensive calculation for the exponentiation and prime search procedures could be programmed to execute on a PC in a reasonable time. The challenge was to prove that this could be done. To meet this challenge, great effort was made to optimize the code and fine tune the algorithms. The timing results, obtained on an 8 MHz Personal Computer AT® (PC AT), are discussed throughout this paper.

A special case of division is the so-called modular division, where the quotient is not explicitly computed and the remainder is the desired result.

Four major programs that have been implemented are discussed in the following sections. Each program is dependent on the previous ones, as listed in the order given.

1. Multiple precision integer arithmetic
2. Modular exponentiation
3. Prime number search
4. Cryptographic key generation

Multiple precision integer arithmetic

The starting point of the project was the implementation of a set of arithmetic procedures to operate on very large integers, i.e., integers that contain several hundred digits. These procedures may be useful in other applications as well as cryptography; hence they have been implemented as a self-contained multiple precision integer arithmetic package (MPIA). MPIA operates on integers up to 255 words long, corresponding to 4080 bits or approximately 1220 decimal digits.

To understand the organization of MPIA, consider that computer words are manipulated similar to the digits in ordinary long-hand calculations. For example, multiplying a 3-word number by a 4-word number is analogous to multiplying a 3-digit number by a 4-digit number.

MPIA implements the standard arithmetic operations of addition, subtraction, multiplication, and division. A special case of division is the so-called *modular division*, where the quotient is not explicitly computed and the remainder is the desired result. There are also some computer-oriented operations such as movement of numbers and shifting.

Multiplication and squaring are the operations that deserve the most attention. They are important because they are major components of modular exponentiation, they are relatively time-consuming, and they lend themselves to considerable optimization. Squaring, i.e., multiplying a number by itself, is important because it is usually much more efficient to square a number by using a specialized squaring procedure than by using a multiplication procedure. The reason is that in squaring a number there are many *cross-product* terms that need to be computed only once, whereas a multiplication procedure will compute them twice. The first description of a multiple precision squaring procedure that uses this concept seems to be due to Tuckerman.[8] Our timing tests consistently show that squaring time can always be reduced to about 60 percent of multiplication time.

Apart from the cross-product procedure, the optimization techniques that are applicable to multiplication are essentially the same for squaring. For this reason, and since multiplication is the more general operation, the discussion below centers on multiplication.

Two approaches to optimization are discussed. They are called *optimization in the small* and *optimization in the large*. Optimization in the small assumes that each word of one factor must be multiplied by each word of the other factor, and endeavors to devise the most efficient possible data structure and code to carry out this basic multiplication procedure. Optimization in the large aims at reducing the total number of single precision multiplication instructions that have to be executed, by splitting the factors into smaller pieces, rearranging the pieces in various ways, and feeding them to the basic multiplication procedure. These rearrangements entail additional processing time, or overhead; hence the overall optimum is obtained when the reduction in multipli-

cations just balances the overhead. This gives rise to a threshold, denoted by P, such that only numbers of length greater than or equal to P should be rearranged. Anticipating later results, in the implementation reported here the threshold turned out to be $P = 28$.

Optimization in the small. The straightforward way to implement a multiplication procedure for two multiword numbers, or factors, is with a double loop. In the outer loop each word of the first factor, or multiplier, is successively loaded into a register; in the inner loop the contents of that register are successively multiplied by each word of the second factor, or multiplicand, and the results are added to the partial product obtained so far. The inner loop for this procedure is illustrated in Example A, which follows.

Example A

```
            PUSH  BP
            XOR   BP,BP
            .
            .
      MMUL30:
            INC   SI
            INC   SI
;fetch multiplicand word
            MOV   AX,[SI]
;preloaded multiplier word
            MUL   DI
            ADD   AX,BP
            ADC   DX,0
;add to memory
            ADD   [SI][BX],AX
            ADC   DX,0
            MOV   BP,DX
            LOOP  MMUL30
            .
            .
            POP   BP
```

The loop takes 20 bytes of code, and the time required for one execution of the loop has been measured to be 7.55 microseconds. An exact allocation of time to the individual instructions cannot be made, since the execution cycle of one instruction overlaps the fetch cycle of the next; nevertheless, numerous timing experiments lead to the following general conclusions:

- The MUL (multiply) instruction takes only about one-third of the loop time.
- The LOOP instruction is very time-consuming.
- The three memory accesses (one to fetch a multiplicand word, and two when adding to memory) are also relatively expensive.

These conclusions are the bases for the two restructuring steps that lead to the optimized basic multiplication procedure. The first step is to remove the LOOP instruction and the jump associated with it. This is done by unraveling the loop, that is, repeating the code in-line a number of times. (A macro may be used for this purpose.) If the code in Example A is unraveled, the register CX is freed up and can be used to replace the instruction ADC DX,0 by ADC DX, CX, further reducing the code length. The results are shown in Example B, where the loop is unraveled. The code takes only 16 bytes.

Example B

```
            PUSH  BP
            XOR   BP,BP
;set to 0 to shorten ADC instructions
            XOR   CX,CX
            .
            .
            INC   SI
            INC   SI
            MOV   AX,[SI]
            MUL   DI
            ADD   AX,BP
            ADC   DX,CX
            ADD   [SI][BX],AX
            ADC   DX,CX
            MOV   BP,DX
;(the above code is repeated 30 times)
            .
            .
            POP   BP
```

Referring to the threshold P mentioned above, the number of static repetitions of the code must be at least $P - 1$, since any factor of length P or longer is further broken apart. If n and m are the lengths of the factors to be multiplied, the outer loop is executed n times; for each of these calculations, the inner unraveled loop must be executed m times. Hence a jump address must be computed, so that only the last m occurrences of the unraveled loop are executed. This computation is done only once, outside the outer loop. Timing of the unraveled loop shows that each execution takes 6.03 microseconds.

The second restructuring step is intended to reduce memory accesses, and can be visualized as follows.

Figure 1 Rectangular sweep

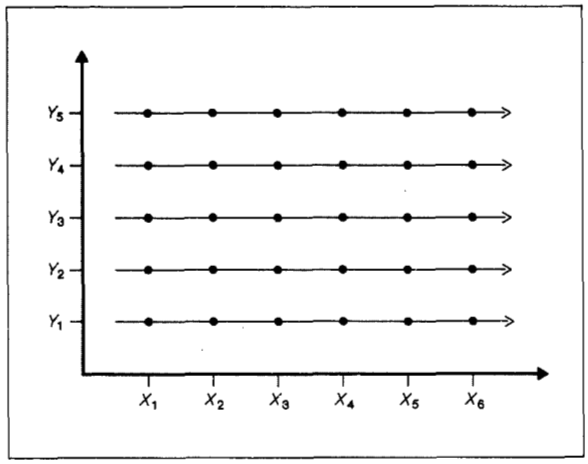
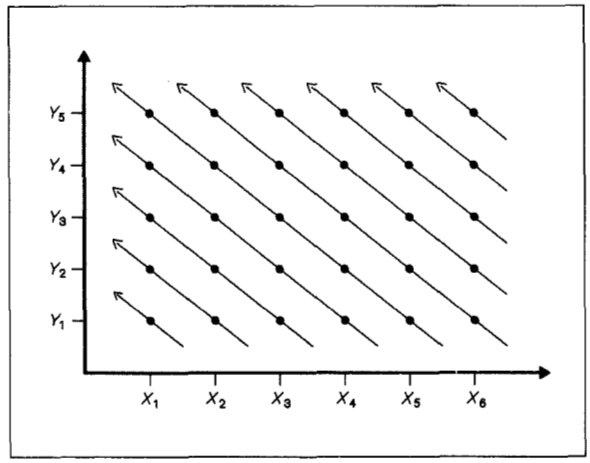


Figure 2 Diagonal sweep



If the m words of the multiplicand are represented by points on the X-axis with abscissas 1, 2, . . . , m, and the n words of the multiplier are similarly represented on the Y-axis, then the nm word-by-word products to be computed correspond to the lattice points of a rectangle in the first quadrant. The procedure described above represents a process in which the lattice points are swept row by row, in a rectangular fashion, as shown in Figure 1. In the alternative procedure, the lattice points are swept in a diagonal fashion, as shown in Figure 2. The diagonal procedure entails only two memory accesses, instead of three, for each multiply instruction: one access for each word to be multiplied. For each diagonal, the products corresponding to the individual points can be accumulated in the registers; there is no need to execute an add-to-memory instruction until the entire diagonal has been processed.

The code must now provide for the possibility of a double carry when the two-word partial products are accumulated. In the PC the second carry may be accumulated in a half register. A diagonal of length 4 is illustrated in Example C, where each short segment of code is preceded by a comment that explains its function.

Example C

```
;Z[3] = carry + X[0]xY[3]
                MOV     AX,[BX]
                MUL     WORD PTR[SI + 6]
                ADD     BP,AX
                ADC     DI,DX
                ADC     CL,CH
;Z[3] = Z[3] + X[1]xY[2]
                MOV     AX,[BX + 2]
                MUL     WORD PTR[SI + 4]
                ADD     BP,AX
                ADC     DI,DX
                ADC     CL,CH
;Z[3] = Z[3] + X[2]xY[1]
                MOV     AX,[BX + 4]
                MUL     WORD PTR[SI + 2]
                ADD     BP,AX
                ADC     DI,DX
                ADC     CL,CH
;Z[3] = Z[3] + X[3]xY[0]
                MOV     AX,[BX + 6]
                MUL     WORD PTR[SI]
                ADD     BP,AX
                ADC     DI,DX
                ADC     CL,CH
;store Z[3], shift carry
                MOV     DX,DI
                POP     DI
                INC     DI
                INC     DI
                MOV     [DI],BP
                PUSH    DI
                MOV     BP,DX
                MOV     DI,CX
                XOR     CL,CL
```

A drawback of the diagonal method is that the lengths of the diagonals vary, first increasing, then

remaining constant (if $n < m$), then decreasing. The required control code is then rather complicated and time-consuming. The problem can be avoided by a radical solution: unraveling not only the inner loop, but the outer loop as well. With this approach, code similar to Example C is repeated in-line, with the length of the diagonal (i.e., the number of occurrences of the inner loop) increasing from 1 to 30, say, and then decreasing down to 1. At the end of each of the increasing diagonals, two tests are inserted (in most cases only one is executed) to determine whether to continue with the next longer diagonal, jump to the next shorter diagonal, or repeat the same one.

As shown in Example C, the code for the inner loop is in most cases 12 bytes long. With appropriate changes in the offsets, this code will then occur in-line 900 times. Adding the code at the end of each diagonal, this results in about 12K bytes of code for the basic multiplication procedure.

The execution time for the doubly unraveled loop is estimated to be 5.03 microseconds, and the actual multiplication time for two factors of length n words is represented by the formula

$$T(n) = 5.03n^2 + 10n + 28\,. \qquad (1)$$

Optimization in the large. The procedure explained below is described by Knuth[9] as a simplification of one attributed to Karatsuba.

For the personal computer, with a word length of 16 bits, we denote by $B = 2^{16}$ the number base corresponding to one word. Assume now that u and v are two n-word numbers to be multiplied. If n is even let $m = n/2$, and if n is odd let $m = (n + 1)/2$. Also let $X = B^m$. Then u and v may be represented as $u = U_0 + U_1X$ and $v = V_0 + V_1X$, where U_0 is the low-order m-word piece of u and U_1 is the high-order piece; likewise for V_0 and V_1. One can then verify the following identity:

$$uv = U_0V_0 + (U_0V_0 + U_1V_1 - (U_1 - U_0)(V_1 - V_0))X + U_1V_1X^2 \qquad (2)$$

The operations required to compute the right-hand side of Equation 2 can now be counted, noting that the apparent multiplications by the powers of X are not actually executed as such; they just indicate that the respective coefficients are stored in a shifted position. For multiplications, there are 3 products of m-word factors. For additions (and subtractions), there are 2 additions of length m inside the double parentheses; 2 additions of length $2m$ inside the single parenthesis; and finally the coefficient of X, which is of length $2m$, has to be added into the rest of the result. Note that the first and third terms on the right-hand side do not overlap and do not require addition. The total length of the quantities to be added is thus $8m$. Restating these counts in terms of n we reach the following conclusion (which is approximate if n is odd): the Karatsuba method replaces one multiplication of factors of length n with 3 multiplications of factors of length $n/2$, plus $4n$ single precision additions.

Table 1 Performance of Karatsuba method

n	$t(n)$	$r(n)$
15	1.31	1.00
20	2.24	1.00
25	3.42	1.00
30	4.52	.93
40	7.49	.88
50	11.2	.85
60	14.7	.78
80	24.0	.73
100	35.5	.69
120	46.2	.63

For large values of n, the Karatsuba method yields a time reduction of almost 3 to 4. Also, the algorithm can be applied recursively, thus compounding the improvement with each splitting of the factors. As the length of the factors decreases, the improvement also decreases, until the "no-improvement" threshold $P = 28$ is reached.

Table 1 shows the measured multiplication time $t(n)$ in milliseconds, for two factors of length n, when the Karatsuba method is applied recursively. Also shown is $r(n)$, which is the ratio of $t(n)$ to $T(n)$; $T(n)$ is the basic multiplication time computed from Equation 1. The value of $r(n)$ represents the improvement attributable to the Karatsuba method.

The Karatsuba method is directly applicable when the factors to be multiplied are of equal size, and this is usually the case in the exponentiation calculations. However, if the factor sizes are unequal, two procedures are available to reduce the problem to the equal size case. First, if the size difference is

small, the smaller number is extended at the high end with null words. Second, if the size difference is large, the product can be visualized as a rectangle; from this rectangle the largest possible square is removed and treated as a product of equal factors; from the remaining rectangle the largest possible square is removed, etc.; continuing until either a rectangle consists only of squares, or a rectangle with a size less than P is found; the latter is then fed to the basic multiplication procedure; the results of the square multiplications are then added with appropriate shifts.

Modular exponentiation

The standard procedure for exponentiating a number, i.e., raising it to a power, in general requires many multiplications and squarings. For example, the computation of M^e is normally done by first representing e in binary notation, as a string of bits; then M is squared as many times as there are bits (excluding the leftmost bit), and the partial results are multiplied by M as many times as there are bits with the value 1. If the exponentiation is done mod n, after each squaring or multiplication the remainder mod n is computed.

For example, let us assume that the numbers M, e, and n are each 640 bits long, which is the same as 40 PC words; let us also assume that approximately half the bits of e have the value 1. Then the computation of M^e mod n requires

- 639 squarings of 40-word numbers—exactly
- 320 multiplications of two 40-word numbers—approximately
- 959 modular divisions of an 80-word number by a 40-word number—approximately

In the previous section the optimization of multiplication and squaring was discussed. Here we discuss two more techniques for speeding up modular exponentiation: a procedure to do the remainder (modulo) calculation using multiplication rather than division, and a technique for reducing the number of multiplications required. For simplicity of notation, these techniques are described for the case mentioned in the example above, where M, e, and n are each 40 words long.

Modular calculation by multiplication. As before, we let $B = 2^{16}$. Since M is a 40-word number, $M < B^{40}$. We now precompute the numbers

$$N_0 = B^{40} \bmod n$$
$$N_1 = B^{41} \bmod n$$
$$\cdot$$
$$\cdot$$
$$N_{39} = B^{79} \bmod n$$

B^{40} is 1 followed by 40 words of 0s. Hence B^{40} mod n is the remainder of the division of a 41-word number by a 40-word number; this is a fairly rapid operation, often involving only one 1×40 multiplication. To compute N_1, the previously computed N_0 is shifted left one word, then divided by n; and so on. Thus the entire precomputation is quite rapid.

Suppose now that we have to compute X mod n, where X is an 80-word number with words X_{79}, X_{78}, $\ldots$, X_1, X_0. We do this by computing the number

$$X_{40}N_0 + X_{41}N_1 + \cdots + X_{79}N_{39}, \qquad (3)$$

and adding to it the low-order half of X, consisting of X_{39}, $\ldots$, X_1, X_0. The resulting 41- or 42-word number is divided by n to obtain the remainder.

The procedure described can be further improved by doing the multiplications indicated in Equation 3 in parallel. This means that in the inner loop of the code, the word X_{40} is multiplied by the ith word of N_0 and the result is saved in a pair of registers; then the word X_{41} is multiplied by the ith word of N_1 and the result is added into the same pair of registers, etc. This technique is similar to the one used in the diagonal code for multiplication, previously described, and it requires only two storage accesses per single precision multiplication, rather than the three required by the conventional approach. A further speedup can be obtained by unraveling the inner loop.

Reducing the number of multiplications. In computing M^e, instead of executing one multiplication for each 1 bit of e, as explained above, it is possible to take the bits in small groups or *nibbles*, for examples of 5 each. The following quantities are then precomputed:

$$M_2 = M^2 \bmod n$$
$$M_3 = M^3 \bmod n$$
$$\cdot$$
$$\cdot$$
$$M_{31} = M^{31} \bmod n$$

This precomputation requires 15 squarings, 15 multiplications, and 30 modular divisions. In the exponentiation procedure the bits of e are then scanned by nibbles instead of individually, and if a nibble has the value j, the current partial result is multiplied by M_j.

A further improvement is obtained by working with *floating nibbles*, i.e., those having a 1 in the high order position. This reduces the precomputation to 4 squarings, 15 multiplications, and 19 modular divisions, and also reduces the number of nibbles encountered during the scan.

The net result of the optimization is shown in the following reduced operation count:

- 639 squarings of 40-word numbers
- 121 multiplications of two 40-word numbers
- 760 modular divisions done by optimized multiplication

Table 2 shows the actual time for modular exponentiation for selected lengths of the operands. The time varies approximately as the power 2.77 of the length. A straightforward implementation would yield a power of 3; the reduction is due to the use of the Karatsuba method and the other optimization devices.

Prime number search

The problems of (a) deciding whether a number is prime, and (b) if not, finding its prime factors, have occupied generations of mathematicians. Recent discussions of these problems and their relevance to public key cryptography, together with ample bibliographies, can be found in Knuth,[9] Dixon,[10] and Riesel.[11] Riesel's book is the most detailed and contains numerous Pascal programs that are functionally similar to some of the procedures described in this paper.

Both the RSA and the Rabin cryptosystems require that the modulus n be the product of two prime numbers, which are usually denoted by p and q. In both systems the number n is part of the public key, while the factors p and q are kept secret. The security of both systems rests on the extreme difficulty of factoring a large number containing large prime factors. To make sure that the factorization is difficult, certain subsidiary conditions are imposed on p and q; for example $p - 1$ must contain a large prime factor, denoted by p_1; and $p_1 - 1$ must contain a large prime factor, denoted by p_2. Similar conditions apply to q.

In the ElGamal system a single prime number $n = p$ is generated, together with a primitive element α. A secret key x is then chosen by each user, and the corresponding public key $y = \alpha^x$ is computed. The security of the system depends on its being very difficult to infer x from a knowledge of y; this is the so-called discrete logarithm problem which, in the general case, is believed to be of the same order of difficulty as the factorization problem.[3] The same subsidiary conditions apply here; for example $n - 1$ must contain a large prime, etc. Thus the prime search procedure plays an essential role in all these cryptosystems.

The prime search is organized so as to take advantage of the connection between p, p_1, and p_2 (and between q, q_1, and q_2 if applicable). Two main techniques are used: a sieve procedure to eliminate numbers containing small prime factors, and a so-called strong pseudoprime test. These techniques are explained below. For concreteness assume, in the RSA case, that the goal is to generate a modulus of length 200 digits, or 42 words, and that this is achieved by finding primes p and q of length 20 and 22 words, respectively.

The sieve procedure. Because of the relationship between the primes p, p_1, and p_2, the search must be done in reverse order. Thus, from the goal of producing p of length 20 words, the program establishes the subgoal of producing p_1 of length four-fifths of 20, or 16 words, then the further subgoal of producing p_2 of length 13.

An odd pseudorandom number B of length 13 is now generated, and a sieve table $S2$ and a residue table $R2$ are initialized. B is the base for the search, and $S2$ is a set of flags that indicate the results of applying primality tests to B and its odd successors: $B + 2, B + 4, \ldots$. The sieve procedure is carried out with the aid of the Small Prime Table, which is a fixed table containing all the odd primes 3, 5, 7, ..., 65 521 that fit in one PC word; the table has 6541 entries. The number B is divided successively by each prime in the Small Prime Table (or a subset thereof) and the remainder is stored in the table $R2$; also, using the remainder, those flags in $S2$ are turned on that correspond to numbers that are divisible by that particular prime; thus the odd successors of B that are found to be nonprime are flagged.

The numbers that survive the sieve procedure are considered prime candidates and are subjected to the strong pseudoprime test (explained below); the first survivor of that test is taken as the prime p_2.

The search for p_1 is now initiated. Since p_1 must be of length 16, a pseudorandom even number A of length 3 is generated, and the search is carried out among the integers of the form $1 + (j + A) \times p_2$ where $j = 0, 2, 4, \ldots$. A new sieve table $S1$ is now created, but unlike the table $S2$, it does not have to be computed from scratch. Instead, using the values of P_2, B, A and the residues saved in $R2$, the sieve computation for $S1$ is faster than for $S2$. Again, the survivors of the sieve are run through the strong pseudoprime test, and the first survivor of that test is taken as p_1.

The strong pseudoprime test is probabilistic.

The calculation of p from p_1 is very similar to the calculation of p_1 from p_2. The calculation of the qs is also analogous.

The strong pseudoprime test. The strong pseudoprime test used in the prime search procedure is described in References 9 and 10; the details of the algorithm are not repeated here, except to note that, given a number $n = 1 + 2^k m$ to be tested, where m is odd, the test requires the computation

$$x^m \bmod n,$$

which is an exponentiation.

The strong pseudoprime test is probabilistic, in the following sense: of the two possible outcomes of the test, one outcome, "fail" guarantees that n is not prime; the other outcome, "pass" does not guarantee that n is prime, but indicates that the assertion "n is prime" is very probably true. For this reason, a number that passes the test is called a pseudoprime. To increase the confidence that n is indeed prime, the test may be applied repeatedly, with different values for x. The test is characterized as strong because it is an improvement over several tests that had been used before, in that it has a lower probability of letting nonprime numbers pass.

The use of a probabilistic test is generally considered necessary and adequate, in view of the fact that it would be prohibitively time-consuming to execute a deterministic test that identifies prime numbers with mathematical certainty.

As mentioned above, the pseudoprime test is applied, as part of the prime search, to the numbers that pass the sieve procedure. The first time that the test is applied to a prime candidate, it is convenient to choose $x = 3$, since this makes the exponentiation calculation faster (by 15 to 20 percent) than with a multiword value for x. For subsequent applications, x should be a different pseudorandom number each time. The number of such subsequent applications is governed by several parameters whose significance is discussed further in the following subsection.

Some refinements and thresholds. Some of the complexities in the prime number search, as described above, are intended to thwart any attempts at factoring the product $p \times q$, or solving the discrete logarithm problem, by using certain known algorithms. Another algorithm that must be considered is one which is effective when $p + 1$ (or $q + 1$) consist entirely of small prime factors. To safeguard against this possibility, after p is generated, the number $p + 1$ is divided by the Small Primes and any exact divisor is factored out; if the remaining number is too short (according to some *ad hoc* threshold), p is rejected and a new p is generated, starting from p_1.

In the RSA case, further subsidiary requirements must be met:

- The numbers $p - 1$ and $q - 1$ must have a small greatest common divisor; this is almost always the case, but the condition must be tested for, and a new q generated if necessary.
- The ratio $\frac{p}{q}$ must not be close to the ratio of two small integers.

In the ElGamal case, the question of additional conditions on n has been the object of recent studies;[12] it may well be that new requirements will be discovered that must be met in order to insure the difficulty of the discrete logarithm problem.

From a performance viewpoint, an important threshold is determined by the relative efficiency of the sieve procedure and the pseudoprime test. Suppose for definiteness that we are testing a number n of length 20 words. If we divide n by an entry in the Small Prime Table, say s, the probability of finding that n is nonprime is $1/s$, while the cost of the test, determined by timing the division procedure, is about 102 microseconds; thus, over a large series of tests with that particular s, it would take an average of $102s$ microseconds to identify one nonprime number. On the other hand, the pseudoprime test is essentially always effective, but it takes a time equal to 80 percent of the exponentiation time (see Table 2), or 1 376 000 microseconds. Equating the two times and solving for s, we find that approximately $s = 13\,500$. This implies that the optimal policy, when testing a number of length 20, is to use the sieve procedure for primes $< 13\,500$, then switch to the pseudoprime test.

In the program the situation is a little more complicated, because the tests for p_2, p_1, and p are interlinked; however, many timing trials have proved that there is indeed an optimum threshold for switching between the sieve procedure and the pseudoprime test, and its value is approximately the one calculated above. These empirical tests are essential, in addition to the theoretical calculations, in order to tune the program for good performance.

Another set of thresholds regulates the sizes of the flag tables and the residue tables. The performance of the program is not very sensitive to these parameters, since the computations are quite fast. Hence it is convenient to make the tables large enough so the probability of overflow is small. Nevertheless, the probability cannot be reduced to zero; therefore the program must make provisions for its occurrence and be able to restart the corresponding section of the sieve procedure. The computation of the overflow probability is interesting in its own right, since it sheds some light on the performance of the prime search. Consider again the case where the number being tested is of length 20 words, i.e., 96 decimal digits, and note that the frequency of primes among odd numbers of this length is approximately $2/\ln 10^{96} = 0.0090$. If the flag table is taken to be of length 1000, the probability of overflow is about $0.991^{1000} = 0.00011$, or slightly over one in ten thousand.

The frequency of primes can now be used to estimate how many times the pseudoprime test is executed in each prime search. Assuming that the sieve procedure is done, as indicated above, using the small primes $< 13\,500$, the fraction of survivors in the flag table is given approximately by the product

$$\prod_{s_j < 13\,500} \left(1 - \frac{1}{s_j}\right) = 0.118$$

where s_j is the jth odd prime. Thus on average the sieve procedure leaves 118 survivors out of 1000, of which only 9 are primes. Hence the pseudoprime test has to be executed an average of 13 times before a prime is found. Also, because the probability of finding a prime on any given occurrence is so small, the number of executions of the pseudoprime test varies a great deal between one prime search and another. So, while it is possible to optimize the average performance of the prime search, it is impossible to guarantee that the procedure will terminate within a given time.

Table 2 Exponentiation time

Length (*words*)	Time (*ms*)
10	302
15	795
20	1,700
30	5,100
40	11,300
60	35,400
80	77,000
100	143,000

The calculation shown above for the case of a prime of length 20 can be repeated for other lengths, so as to obtain the various thresholds as a function of that length. For example, if one seeks a prime of length 40 words (192 digits), it turns out that the sieve procedure should be carried out with the small primes up to 51 000, resulting in about 104 survivors per thousand; since there are only on average 4.5 primes per thousand odd numbers of this length, the pseudoprime test will be executed an average of 23 times before a prime is found.

Finally, we discuss briefly the parameters that control the number of additional times that the pseudoprime test is applied to a number n after n has passed the pseudoprime test with $x = 3$. First, it must be noted that these parameters, while affecting the time required for the prime search, are not thresholds that can be varied to tune the performance of the pro-

Table 3 Average key generation time

Modulus Length		Average Time	
		RSA	ElGamal
(*digits*)	(*words*)	(*min:s*)	
125	26	0:35	2:15
160	33	1:06	5:00
200	42	2:07	11:00
250	52	4:05	25:00
320	66	9:00	60:00
400	85	19:00	125:00
500	104	42:00	N.A.

gram; instead they affect the probability (extremely small in any case) that the program will accept as prime some numbers that are not prime. Second, we note that in our experience, including an analysis of hundreds of thousands of tests, the first application of the strong pseudoprime test has always been decisive: not a single instance has been recorded of a number that passed the test with $x = 3$ and failed it later with a different x. Third, we briefly summarize the applicable mathematical knowledge, making use of the following terminology introduced by Rabin:[6] if the pseudoprime test applied to n, using a particular value of x, indicates that n is a pseudoprime, when n is in fact not prime, then that x is called a *false witness*. For the vast majority of values of n the false witnesses are extremely rare or nonexistent, while for a very few values of n the false witnesses may be as many as 25 percent of the possible x s. Unfortunately the "vast majority" and the "very few" have not been quantified by mathematical analysis. Hence there are no firm guidelines. A further consideration, in the context of cryptographic key generation, is that the essential goal is to produce the primes p and q, while the primality of p_1, q_1, p_2, q_2 is of lesser importance. In view of the above, the number of additional executions of the pseudoprime test, as currently set in the program, are as follows:

- 0 for p_2 and q_2
- 1 for p_1 and q_1
- 9 for p and q

Cryptographic key generation

Once the required primes have been obtained, the process of key generation can be swiftly completed. In the case of the Rabin cryptosystem, where the public key is the product $n = p \times q$, all that remains to be done is multiply the two numbers.

For the ElGamal system, an integer $\alpha < n$ must be found which is primitive mod n. This condition is easy to check, since the factorization of $n - 1$ is known. Thus a random α is generated and exponentiated with each factor of $n - 1$; if any of the results are unity, α is discarded. Very few trials are needed in most cases. Each individual user can then arbitrarily choose his private key x and compute the corresponding public key y by exponentiation.

For the RSA system, several additional numbers must be produced. The first of these is $\phi = (p - 1)(q - 1)$, which must be kept secret. The public encryption exponent e may be chosen next, and it must be a number relatively prime to ϕ. Since a smaller e tends to result in faster encryption, a small prime may be a good choice. The number 3 was once recommended,[9] but it has since been shown that it is vulnerable in the so-called broadcast situation (i.e., if the same message is encrypted with three different moduli but with $e = 3$ in all cases, and if an attacker can intercept and analyze the three ciphertexts). The number $2^{16} + 1 = 65\,537$ is now advocated by some as being sufficiently large to avoid this problem.

The computation of the secret decryption exponent d, which must satisfy the equation $ed \bmod \phi = 1$, is accomplished by means of a straightforward and fairly fast procedure that resembles a greatest common divisor calculation. If e is chosen first, as indicated, then d will usually turn out to be almost as large a number as n, causing the decryption process, represented by the equation $M = C^d \bmod n$, to be quite slow. The situation can be remedied in part by computing four auxiliary numbers defined by the equations

$$A_p = q^{p-1} \bmod n, \qquad A_q = n + 1 - A_p,$$
$$d_p = d \bmod (p - 1), \qquad d_q = d \bmod (q - 1);$$

decryption is then done by computing

$$M = (A_p \times ((C \bmod p)^{d_p} \bmod p) + A_q \times ((C \bmod q)^{d_q} \bmod q)) \bmod n.$$

In this calculation, the modular exponentiations are done with the moduli p and q, which are about half the length of n; the time required is usually reduced by about 70 percent. The auxiliary numbers, like all the other quantities except n and e, must be kept secret.

To verify that the computations have been done correctly, the program now generates a random

number of the same length as n, encrypts it and decrypts it, and verifies that the end result agrees with the original. As an additional test for RSA, another random number is generated, decrypted and encrypted, then compared with its original.

For experimental and testing purposes, the key generation program was written so it could generate keys with moduli between 60 and 600 digits long (12 to 125 words), although in actual practice the extremes of the range are not useful: less than 160 is probably not secure enough, and more than 320 seems to be overkill. The average key generation time is shown in Table 3. The variability in the prime search procedure, previously discussed, causes the key generation time to be also highly variable in individual cases, especially for longer moduli; it is not uncommon for a key generation run to take anywhere between half to twice the stated average time.

Speed and cryptosystems design

The figures in Table 3 show that for moduli in the range of 200 to 250 digits, which is generally considered an adequate length, the average RSA key generation time on a PC AT is two to four minutes. This time seems quite reasonable when one considers that in a public key system the keys should rarely be changed (just like telephone numbers).

It is apparent that, for a given modulus length, the average ElGamal key generation time is larger by a factor of 4 to 7; the reason is that a single large prime is required, rather than two smaller ones. However, the difference may not be very significant, in that with the ElGamal system the same modulus is shared by many users; hence very few modulus generations are needed.

A different picture emerges when one compares any public key encryption/decryption time with the corresponding time for the Data Encryption Algorithm. This comparison is first made with regard to the RSA system, then some observations are presented pertaining to the ElGamal system. The first consideration, already mentioned, is that RSA encryption can be made quite fast by choosing the number 3, or another small prime, as the public key; but since each encrypted message has to be decrypted in order to make sense, it is really the average of encryption and decryption time that gives a meaningful measure of performance. Under DEA the two times are essentially the same.

Table 4 RSA and DEA block encryption time

Modulus Length (digits)	Modulus Length (words)	Enc. Time (ms)	Dec. Time (ms)
125	26	23	1,210
160	33	35	2,220
200	42	50	4,010
250	52	73	7,240
320	66	115	13,940
400	85	170	26,040
500	104	250	47,950
DEA		0.306	0.308

Table 5 RSA and DEA byte encryption time

Modulus Length (digits)	Modulus Length (words)	Enc. Time (ms)	Dec. Time (ms)
125	26	0.44	23
160	33	0.53	34
200	42	0.61	48
250	52	0.70	70
320	66	0.87	105
400	85	1.02	157
500	104	1.20	230
DEA		0.0383	0.0386

The second point is that the comparison can be made in two ways:

- By comparing the time needed to encrypt/decrypt a single block—Under DEA the block, or minimum unit of encryption, is 8 bytes; under RSA it is equal in length to the modulus. This comparison is appropriate if the intended application requires the encryption of short items, such as DEA keys, or individual database fields.
- By comparing the average time per byte—This is appropriate if the application involves bulk encryption of large files.

Table 4 and Table 5 show the results of the two comparisons. The times for the DEA were obtained by using a highly optimized implementation of the algorithm.

It is apparent that the speed of the DEA, compared to the average encrypt/decrypt speed of a 200-digit RSA, is better by almost three orders of magnitude at the byte level, and almost four at the block level. Such enormous differences are bound to affect the role that DEA or RSA can play in any integrated

Table 6 ElGamal block encryption time

Modulus Length (*digits*)	(*words*)	Av. Enc./Dec. Time (*s*)
125	26	5.1
160	33	10.6
200	42	19.2
250	52	35.
320	66	72.
400	85	135.
500	104	235.

cryptosystem. While DEA is likely to remain the algorithm of choice for routine message and file encryption, the increasing recognition being given to the importance of key management, and the difficulty of accomplishing it with DEA, have drawn attention to the unique advantages of the public key systems. If a public key algorithm is used only for the specialized task of exchanging other cryptographic keys, the performance indicated in the tables—several seconds for each key exchange—is quite acceptable in many environments.

Under the ElGamal system, encryption/decryption is slower than under RSA, as can be seen by comparing Table 6 and Table 4. Here encryption requires two modular exponentiations and it cannot be significantly speeded up by choosing a small key; decryption entails an exponentiation with a modulus that cannot be factored. Another disadvantage of ElGamal is that each block of plaintext produces two blocks of ciphertext.

On the plus side, one can list two advantages:

- Encryption involves a random number, so if a given plaintext is encrypted on two different occasions the corresponding ciphertexts are different; this is a protection against possible replay attacks.
- Under RSA, if the plaintext is assumed to be chosen from a small set of candidates, an attacker can encipher each candidate with the public key and compare the results with the ciphertext; under ElGamal this attack does not work.

Conclusion

This paper has stressed a variety of techniques for increasing the speed of the numerical calculations that constitute the core of exponentiation cryptosystems. The results that can be obtained by using these techniques have been illustrated. A PC implementation of the major components of these algorithms yields a very reasonable performance. The solutions made possible by these algorithms are likely to become increasingly important as the interconnectivity of computers grows.

Cited references and note

1. *Data Encryption Standard*, Federal Information Processing Standard (FIPS) Publication 46, National Bureau of Standards, Washington, DC (January 1977).
2. W. Diffie and M. Hellman, "New Directions in Cryptography," *IEEE Transactions on Information Theory* **IT-22,** 644–654 (November 1976).
3. D. E. R. Denning, *Cryptography and Data Security*, Addison-Wesley Publishing Co., Reading, MA (1982).
4. W. Patterson, *Mathematical Cryptology*, Rowman & Littlefield, Totowa, NJ (1987).
5. R. L. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Communications of the ACM* **21,** 120–126 (February 1978).
6. M. O. Rabin, "Digitalized Signatures and Public-Key Functions as Intractable as Factorization," MIT/LCS/TR-212, Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, MA (January 1979).
7. T. ElGamal, "A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms," *IEEE Transactions on Information Theory* **IT-31,** 469–472 (July 1985).
8. B. Tuckerman, "The 24th Mersenne Prime," *Proceedings of the National Academy of Science* **68,** 2319–2330 (1970).
9. D. E. Knuth, *The Art of Computer Programming*, Volume 2 (second edition), Addison-Wesley Publishing Co., Reading, MA (1981).
10. J. D. Dixon, "Factorization and Primality Testing," *American Mathemathics Monthly*, 333–352 (June-July 1984).
11. H. Riesel, *Prime Numbers and Computer Methods of Factorization*, Birkhauser, Boston, MA (1985).
12. Don Coppersmith, private communication.

Paul G. Comba *IBM Cambridge Scientific Center, 101 Main Street, Cambridge, Massachusetts 02142.* Dr. Comba joined IBM in 1960 and has since worked on a variety of advanced technology projects, mainly in software and application development, including programming languages, simulation, graphics, and database management. In the last seven years, he has worked primarily in the area of software implementation of cryptographic systems. Dr. Comba received his Ph.D. in mathematics from the California Institute of Technology in 1951. Before joining IBM he was assistant professor, then associate professor of mathematics at the University of Hawaii from 1951 to 1960.

Reprint Order No. G321-5416.