Fişa suspiciunii de plagiat / Sheet of plagiarism's suspicion	**Indexat la: 34/06**

Opera suspicionată (OS)	**Opera autentică (OA)**
Suspicious work	**Authentic work**

OS	Mang E., Mang I., "Determin large prime numbers to compute RSA system parameters", *Journal of Computer Science and Control Systems*, Vol.1, Nr. 1, p.54-57, 2008. Disponibil la: http :// electroinf . uoradea . ro / reviste %20 CSCS / documente / JCSCS _ 2008 / JCSCS _ 2008 _ 10 _ MangE _ 1 . pdf
OA	Beth, T.,Gollmann, D., "Algorithm Engineering for Public Key Algorithms", *IEEE Journal on selected areas in communications*, Vol. 7. No 4, p.458-465, May 1989.

Incidența minimă a suspiciunii / Minimum incidence of suspicion	
p.54:17s - p.57:19s	p.458:6s - p.459:37d

Fişa întocmită pentru includerea suspiciunii în Indexul Operelor Plagiate în România de la www.plagiate.ro

Algorithm Engineering for Public Key Algorithms

THOMAS BETH AND DIETER GOLLMANN

Abstract—**We will examine ways of implementing public key algorithms based on modular integer arithmetic (RSA) and finite field arithmetic (Diffie–Hellman, ElGamal). In particular, we will be concerned with architectures for VLSI implementations.**

I. Introduction

Research on public key algorithms has been concerned mainly with security aspects. The results of this research have induced sufficient confidence to apply public key cryptography on a larger scale. The ISO and CCITT have been discussing public key systems. As an example, take the CCITT directory authentication framework [12], which refers to a public key algorithm. (Precisely which algorithm will be used is left for discussion, although RSA is obviously a prime candidate.) ISDN can create new applications for public key algorithms if their implementations can meet requirements ranging from bit rate (from 64 kbits/s up to 140 Mbits/s), to storage and chip area, to physical security.

Realizations of some of the most popular public key algorithms rely essentially on efficient exponentiation. In the RSA algorithm, encryption and decryption are performed by exponentiation modulo a large integer N. Exponentiation will be decomposed into a square-and-multiply algorithm. In modular integer arithmetic, squaring is usually as difficult as multiplication. Hence, we will only deal with the second. Binary representation of numbers suggests using a shift-and-add algorithm for multiplication. Reduction modulo N is usually performed after each shift-and-add step. To perform addition efficiently, the propagation of carries has to be controlled. This can be done by carry-look-ahead techniques [33]. Because of the area required, this architecture cannot be extended to an arbitrary length of look-ahead. Alternatively, a redundant delayed-carry representation can be used [9], [29], [35]. The carries have to be resolved only at the end of a multiplication. An alternative is Bucci's method [10], which takes multiplication itself as a primitive operation, and computes the modulus by a series of multiplications, and cuts off the least significant bits (lsb's).

The Diffie–Hellman key exchange system and the ElGamal public key system are both based on exponentiation in a finite field GF(q). For q prime, the considerations for modular integer arithmetics apply again. New aspects arise in the case $q = 2^n$. Exponentiation in GF(2^n) will again be decomposed into a square-and-multiply algorithm. In GF(2^n), we have $(u + v)^2 = u^2 + v^2$; hence, squaring is a linear operation. Thus, we can hope for more efficient squaring algorithms. Different basis representations will favor multiplication or squaring, respectively. Squaring in a polynomial basis is, in general, as expensive as multiplication. The same holds for dual basis representations. In a normal basis, squaring becomes a cyclic shift and can be performed in a single clock cycle. This fact has been used in the design of the Massey–Omura multiplier [27]. However, the circuit for multiplication will require, in general, $O(n^2)$ gates and have a rather irregular structure, so that the choice of a suitable normal basis is of great importance.

Manuscript received August 1, 1988; revised January 15, 1989.
The authors are with the Fakultät für Informatik, Universität Karlsruhe, 7500 Karlsruhe 1, West Germany.
IEEE Log Number 8927171.

II. RSA

Implementations on 8-bit microprocessors, as used on chip cards (Hitachi 65901, SEEQ 72720/TMS-7000) or on circuits based on processors of the 68000 family, achieve about 10–700 bits/s (for a 512-bit RSA). Dedicated circuits built with standard techniques yield about 6800 bits/s (CYLINK Inc., RSA Inc.). This bit rate may be sufficient for authentication and signature schemes using the Fiat–Shamir protocol [19], which uses very few multiplications. However, for general security applications in ISDN, at least 64 kbits/s has to be achieved, and ultimately, several megabits per second must be achieved for broadband ISDN.

In the first part of our paper, we will therefore present a variety of algorithms that can help to facilitate better performance. Several authors have announced chips, e.g., the CT10018 microprocessor from Crypto-Technologies, or algorithms [33], [35] that can achieve ISDN data rates. Usually, efficient implementation of these algorithms will demand a dedicated hardware architecture. Thus, we will establish a close connection between full custom VLSI design and the "mathematics" of these algorithms.

A. The Algorithm

The public key algorithm most frequently referred to was proposed by Rivest, Shamir, and Adleman [32] in 1978. It is based on modular exponentiation.

We start with a short description of the RSA algorithm. The private information of a user consists of two primes p and q. Security considerations suggest using two primes with up to 100 decimal digits. From this private information, the user computes the public key, consisting of the product $N = p \cdot q$ and a number $e > 1$ which is coprime to $p - 1$ and $q - 1$. To transmit a message, the sender divides the message into blocks m_i where m_i are numbers in the interval $[1, N-1]$. To encipher a block m, the sender uses the public numbers N and e to form

$$m \rightarrow m^e \text{ MOD } N.$$

The receiver knows the factors of $N = p \cdot q$ and is thus able to compute in advance the (secret) deciphering key d, $d \in [1, (p-1)(q-1)]$, uniquely defined by

$$e \cdot d \equiv 1 \text{ MOD } (p-1)(q-1).$$

Elementary number theory guarantees, for $c \equiv m^e$ MOD N,

$$c^d \equiv m \text{ MOD } N.$$

The security of RSA depends on the difficulty of factoring "hard" large numbers. Most numbers can be factored easily, so the primes p and q have to be chosen carefully to give a "hard" number N. This proves to be not too difficult. On the other hand, it still seems impossible to factor hard numbers with 200 decimal digits, even using supercomputers or other advanced computer architectures (cf. Caron and Silverman [11] and Davis, Holdridge, and Simmons [16]).

It should be noted that RSA or similar systems can be used to implement security services, including key management, encipherment, signatures, and authentication. For the latter applications, new zero-knowledge techniques (cf. Goldreich, Micali, and Wigderson [15] and Berger, Kannan, and Peralta [2]) have been proposed by Fiat and Shamir [19].

B. Exponentiation MOD N

All of the above applications require in one way or another algorithms for computing the modular exponentiation

$$m \rightarrow m^e \text{ MOD } N.$$

At this point, we will make no further assumptions on the structure of N. Exponentiation can be implemented by a square-and-multiply algorithm (Knuth [23]). There are two ways this can be done. Starting from the lsb of the exponent, we get

$$m^e = m^{e_0}(m^2)^{e_1}(m^4)^{e_2} \cdots (m^{2^{n-1}})^{e_{n-1}}$$

(n denotes the length of the binary representation of N). Multiplying the intermediate result with $(m^{2^k})^{e_k}$ and updating m^{2^k} to $m^{2^{k+1}}$ can be done in parallel. Hence, we call this the parallel square-and-multiply algorithm. Starting from the most significant bit (msb) of the exponent, we get

$$m^e = \left(\cdots\left(\left(m^{e_{n-1}}\right)^2 \cdot m^{e_{n-2}}\right)^2 \cdots \cdot m^{e_1}\right)^2 \cdot m^{e_0}.$$

The intermediate result has to be squared before it can be multiplied by m^{e_k}. Hence, we call this the serial square-and-multiply algorithm. We have now identified the two basic operations: "square MOD N" and "multiply MOD N." These are fixed-point long-integer arithmetic operations. As N is odd, we can divide by 2 MOD N. In a binary number representation, this is just a shift and possibly an addition. The product $a \cdot b$ can thus be computed by

$$a \cdot b = \frac{(a+b)^2 - a^2 - b^2}{2}$$

using only additions and squaring. Hence, we assume that squaring is about as expensive as multiplication and examine only the latter.

C. Multiplication MOD N

Binary number representations suggest using a shift-and-add algorithm for multiplication.

$$\begin{aligned} a \cdot b &= \left[\sum_{i=0}^{n-1} a_i 2^i\right] \cdot b \\ &= a_0 b + 2a_1 b + 4a_2 b + \cdots + 2^{n-1} a_{n-1} b \end{aligned}$$

has a structure similar to that of the parallel square-and-multiply algorithm. It is obvious how a serial shift-and-add algorithm could be defined. Multiplication by 2 MOD N comprises a shift and, when the result is larger than N, also an addition by $-N$. Addition MOD N is thus the essential basic operation in our decomposition of exponentiation MOD N. The following topics will be addressed in the search for efficient multiplication algorithms.

- *Control of the Carries:* Simple carry-ripple adders would take too much time. This disadvantage has to be faced when implementing RSA on a standard microprocessor.
- *Computation of Residues MOD N:* Intermediate results need not be reduced MOD N after each step; some time can be gained by allowing some overflow and adding an appropriate multiple of $-N$. This gain in time has to be balanced against the additional space for storing the multiples of $-N$.
- *Step-Width in Shift-and-Add:* The factor a need not be processed bit by bit as suggested in the above shift-and-add algorithm. The time for multiplication may be reduced by dealing with substrings of a, either of fixed or arbitrary length. Treating a as a sequence of runs of 0's and 1's is an example for the second case.

There exist different propositions to handle these problems. These propositions will be discussed in the following paragraphs.

D. Brickell's Algorithm

Timing problems due to carry propagation can, of course, be controlled by avoiding carries for as large a part of an addition as possible. This can be achieved by using a redundant representation of numbers that ensures that carries can propagate only by one bit. Longer ripples will occur only when converting the redundant representation back to some standard binary format. The well-known carry-save adder was modified by Norris and Simmons to a "delayed-carry" adder [29]. Brickell [9] has extended this technique to a multiplication algorithm that