Efficiency in Public-Key Cryptography

Montgomery Multiplication in GF(2^k) *

Ç. K. Koç and T. Acar
Electrical & Computer Engineering
Oregon State University
Corvallis, Oregon 97331
{koc,acar}@ece.orst.edu

Abstract

We show that the multiplication operation $c = a \cdot b \cdot r^{-1}$ in the field GF(2^k) can be implemented significantly faster in software than the standard multiplication, where r is a special fixed element of the field. This operation is the finite field analogue of the Montgomery multiplication for modular multiplication of integers. We give the bit-level and word-level algorithms for computing the product, perform a thorough performance analysis, and compare the algorithm to the standard multiplication algorithm in GF(2^k). The Montgomery multiplication can be used to obtain fast software implementations of the discrete exponentiation operation, and is particularly suitable for cryptographic applications where k is large.

1 Introduction

The arithmetic operations in the Galois field GF(2^k) have several applications in coding theory, computer algebra, and cryptography. We are especially interested in cryptographic applications where k is very large. Examples of the cryptographic applications are the Diffie-Hellman key exchange algorithm [3] based on the discrete exponentiation and elliptic curve cryptosystems [7, 11] over the field GF(2^k). The Diffie-Hellman algorithm requires implementation of the exponentiation g^e, where g is a fixed primitive element of the field and e an integer. The exponentiation operation can be implemented using a series of squaring and multiplication operations in GF(2^k) using the binary method [6].

Cryptographic applications require fast hardware and software implementations of the arithmetic operations in GF(2^k) for large values of k. The most important advance in this field has been the Massey-Omura algorithm [14] which is based on the normal bases. Subsequently, the optimal normal bases were introduced [13], and their hardware [1, 2] and software [5, 15] implementations were given. While the hardware implementations are compact and fast, they are also inflexible and expensive. The change of the field in a hardware implementation requires a complete redesign. Software implementations, on the other hand, are perhaps slower, but they are cost-effective and flexible, i.e., the algorithms and the field parameters can easily be modified without requiring redesign. Recently, there has been a growing interest to develop software methods for implementing GF(2^k) arithmetic operations for cryptographic applications [15, 16].

*This research is supported in part by Intel Corporation.

In this paper, we present an algorithm for multiplication in GF(2^k), which is significantly faster than the standard multiplication, and is particularly useful for obtaining fast software implementation of the discrete exponentiation operation. The algorithm is based on Montgomery's method for computing the modular multiplication operation. We use the polynomial representation of the field GF(2^k), and show that Montgomery's technique is also applicable here. We have performed a thorough analysis of the Montgomery multiplication algorithm, and compared it to the standard multiplication algorithm in GF(2^k). We show that this operation would be significantly faster in software with the availability of a fast method for multiplying two w-bit polynomials defined over GF(2), where w is the wordsize. For example, the Montgomery multiplication is about 5 times faster for $w = 8$, and about 20 times faster for $w = 32$.

2 Polynomial Representation

The elements of the field GF(2^k) can be represented in several different ways [9, 10, 8]. We find the polynomial representation useful and suitable for software implementation. The algorithm for the Montgomery multiplication described in this paper is based on the polynomial representation. According to this representation an element a of GF(2^k) is a polynomial of length k, i.e., of degree less than and equal to $k-1$, written as

$$a(x) = \sum_{i=0}^{k-1} a_i x^i = a_{k-1}x^{k-1} + a_{k-2}x^{k-2} + \cdots + a_1 x + a_0 \ ,$$

where the coefficients $a_i \in$ GF(2). These coefficients are also referred as the bits of a, and the element a is represented as $a = (a_{k-1}a_{k-2}\cdots a_1 a_0)$. In the word-level description of the algorithms, we partition these bits into blocks of equal length. Let w be the wordsize of the computer, also assume that $k = sw$. We can write a as an sw-bit number consisting of s blocks, where each block is of length w. Thus, we have $a = (A_{s-1}A_{s-2}\cdots A_1 A_0)$, where each A_i is of length w such that

$$A_i = (a_{iw+w-1}a_{iw+w-2}\cdots a_{iw+1}a_{iw}) \ .$$

In the polynomial case, this is equivalent to

$$a(x) = \sum_{i=0}^{s-1} A_i(x)x^{iw} = A_{s-1}(x)x^{(s-1)w} + A_{s-2}(x)x^{(s-2)w} + \cdots + A_1(x)x^w + A_0(x) \ ,$$

where $A_i(x)$ is a polynomial of length w such that

$$A_i(x) = \sum_{j=0}^{w-1} a_{iw+j}x^j = a_{iw+w-1}x^{w-1} + a_{iw+w-2}x^{w-2} + \cdots + a_{iw+1}x + a_{iw} \ .$$

The addition of two elements a and b in GF(2^k) are performed by adding the polynomials $a(x)$ and $b(x)$, where the coefficients are added in the field GF(2). This is equivalent to bit-wise XOR operation on the vectors a and b. In order to multiply two elements a and b in GF(2^k), we need an irreducible polynomial of degree k. Let $n(x)$ be an irreducible polynomial of degree k over the field GF(2). The product $c = a \cdot b$ in GF(2^k) is obtained by computing

$$c(x) = a(x)b(x) \bmod n(x) \ ,$$

where $c(x)$ is a polynomial of length k, representing the element $c \in$ GF(2^k). Thus, the multiplication operation in the field GF(2^k) is accomplished by multiplying the corresponding polynomials modulo the irreducible polynomial $n(x)$.

3 Montgomery Multiplication in GF(2^k)

Instead of computing $a \cdot b$ in GF(2^k), we propose to compute $a \cdot b \cdot r^{-1}$ in GF(2^k), where r is a special fixed element of GF(2^k). A similar idea was proposed by Montgomery in [12] for modular multiplication of integers. We show that Montgomery's technique is applicable to the field GF(2^k) as well. The selection of $r(x) = x^k$ turns out to be very useful in obtaining fast software implementations. Thus, r is the element of the field, represented by the polynomial $r(x) \bmod n(x)$, i.e., if $n = (n_k n_{k-1} \cdots n_1 n_0)$, then $r = (n_{k-1} \cdots n_1 n_0)$. The Montgomery multiplication method requires that $r(x)$ and $n(x)$ are relatively prime, i.e.,

$$\gcd(r(x), n(x)) = 1 \ .$$

For this assumption to hold, it suffices that $n(x)$ be not divisible by x. Since $n(x)$ is an irreducible polynomial over the field GF(2), this will always be case. Since $r(x)$ and $n(x)$ are relatively prime, there exist two polynomials $r^{-1}(x)$ and $n'(x)$ with the property that

$$r(x)r^{-1}(x) + n(x)n'(x) = 1 \ , \tag{1}$$

where $r^{-1}(x)$ is the inverse of $r(x)$ modulo $n(x)$. The polynomials $r^{-1}(x)$ and $n'(x)$ can be computed using the extended Euclidean algorithm [8, 9]. The Montgomery multiplication of a and b is defined as the product

$$c(x) = a(x)b(x)r^{-1}(x) \bmod n(x) \ , \tag{2}$$

which can be computed using the following algorithm:

Algorithm for Montgomery Multiplication
Input: $a(x), b(x), r(x), n'(x)$
Output: $c(x) = a(x)b(x)r^{-1}(x) \bmod n(x)$
Step 1. $t(x) := a(x)b(x)$
Step 2. $u(x) := t(x)n'(x) \bmod r(x)$
Step 3. $c(x) := [t(x) + u(x)n(x)]/r(x)$

In order to prove the correctness of the above algorithm, we note that $u(x) = t(x)n'(x) \bmod r(x)$ implies that there is a polynomial $K(x)$ over GF(2) with the property

$$u(x) = t(x)n'(x) + K(x)r(x) \ . \tag{3}$$

We write the expression for $c(x)$ in Step 3, and then substitute $u(x)$ with the expression (3) as

$$\begin{aligned} c(x) &= \frac{1}{r(x)} [t(x) + u(x)n(x)] \\ &= \frac{1}{r(x)} [t(x) + t(x)n'(x)n(x) + K(x)r(x)n(x)] \end{aligned}$$

Furthermore, we have $n'(x)n(x) = 1 + r(x)r^{-1}(x)$ according to (1). Thus, $c(x)$ is obtained as

$$\begin{aligned} c(x) &= \frac{1}{r(x)} [t(x) + t(x)[1 + r(x)r^{-1}(x)] + K(x)r(x)n(x)] \\ &= \frac{1}{r(x)} [t(x)r(x)r^{-1}(x) + K(x)r(x)n(x)] \\ &= t(x)r^{-1}(x) + K(x)n(x) \\ &= t(x)r^{-1}(x) \bmod n(x) \\ &= a(x)b(x)r^{-1}(x) \bmod n(x) \ , \end{aligned}$$

as required. The above algorithm is similar to the algorithm given for the Montgomery multiplication of integers. The only difference is that the final subtraction step required in the integer case is not necessary in the polynomial case. This is proved by showing that the degree of the polynomial $c(x)$ computed by this algorithm is less than and equal to $k-1$. Since the degrees of $a(x)$ and $b(x)$ are both less than and equal to $k-1$, the degree of $t(x) = a(x)b(x)$ will be less than and equal to $2(k-1)$. Also note that the degrees of $n(x)$ and $r(x)$ are both equal to k. The degree of $u(x)$ computed in Step 2 will be strictly less than k since the operation is performed modulo $r(x)$. Thus, the degree of $c(x)$ as computed in Step 3 of the algorithm is found as

$$\begin{aligned} \deg\{c(x)\} &\leq \max[\deg\{t(x)\}\ ,\ \deg\{u(x)\} + \deg\{n(x)\}] - \deg\{r(x)\} \\ &\leq \max[2k-2\ ,\ k-1+k] - k \\ &\leq k-1 \end{aligned}$$

Thus, the polynomial $c(x)$ is already reduced.

4 Computation of Montgomery Multiplication

The computation of $c(x)$ involves a regular multiplication in Step 1, a modulo $r(x)$ multiplication in Step 2, and finally a division by $r(x)$ operation in Step 3. The modular multiplication and division operations in Steps 2 and 3 are intrinsically fast operations since $r(x) = x^k$. The remainder operation in modular multiplication using the modulus x^k is accomplished by simply ignoring the terms which have powers of x larger than and equal to k. Similarly, division of an arbitrary polynomial by x^k is accomplished by shifting the polynomial to the right by k places. A drawback in computing $c(x)$ is the precomputation of $n'(x)$ required in Step 2. However, it turns out the computation of $n'(x)$ can be completely avoided if the coefficients of $a(x)$ are scanned one bit at a time. Furthermore, the word-level algorithm requires the computation of only the least significant word $N'_0(x)$ instead of the whole $n'(x)$.

Recall that we need to compute $c(x) = a(b)b(x)r^{-1}(x) \bmod n(x)$, where $r(x) = x^k$. This product can be written as

$$c(x) = x^{-k}a(x)b(x) = x^{-k} \sum_{i=0}^{k-1} a_i x^i b(x) \pmod{n(x)} .$$

The product

$$t(x) = (a_{k-1}x^{k-1} + a_{k-2}x^{k-2} + \cdots + a_1 x + a_0)b(x)$$

can be computed by starting from the most significant digit, and then proceeding to the least significant, as follows:

$t(x) := 0$
for $i = k-1$ to 0
  $t(x) := t(x) + a_i b(x)$
  $t(x) := xt(x)$

The shift factor x^{-k} in $x^{-k}a(x)b(x)$ reverses the direction of summation. Since

$$x^{-k}(a_{k-1}x^{k-1} + a_{k-2}x^{k-2} + \cdots + a_1 x + a_0) = a_{k-1}x^{-1} + a_{k-2}x^{-2} + \cdots + a_1 x^{-k+1} + a_0 x^{-k} ,$$

we start processing the coefficients of $a(x)$ from the least significant, and obtain the following bit-level algorithm in order to compute $t(x) = a(x)b(x)x^{-k}$.

$t(x) := 0$
for $i = 0$ to $k - 1$
  $t(x) := t(x) + a_i b(x)$
  $t(x) := t(x)/x$

This algorithm computes the product $t(x) = x^{-k}a(x)b(x)$, however, we are interested in computing $c(x) = x^{-k}a(x)b(x) \bmod n(x)$. Following the analogy to the integer algorithm, we achieve this computation by adding $n(x)$ to $c(x)$ if c_0 is 1, making the new $c(x)$ divisible by x since $n_0 = 1$. If c_0 is already 0 after the addition step, we do not add $n(x)$ to it. Therefore, we are computing $c(x) := c(x) + c_0 n(x)$ after the addition step. After this computation, $c(x)$ will always be divisible by x. We can compute $c(x) := c(x)x^{-1} \bmod n(x)$ by dividing $c(x)$ by x since $c(x) = xu(x)$ implies $c(x) = xu(x)x^{-1} = u(x) \bmod n(x)$. The bit-level algorithm is given below:

Bit-Level Algorithm for Montgomery Multiplication
Input: $a(x), b(x), n(x)$
Output: $c(x) = a(x)b(x)x^{-k} \bmod n(x)$
Step 1. $c(x) := 0$
Step 2. for $i = 0$ to $k - 1$ do
Step 3. $c(x) := c(x) + a_i b(x)$
Step 4. $c(x) := c(x) + c_0 n(x)$
Step 5. $c(x) := c(x)/x$

The bit-level algorithm for the Montgomery multiplication given above is generalized to the word-level algorithm by proceeding word by word, where the wordsize is $w \geq 2$ and $k = sw$. Let $A_i(x)$ represent one word of the polynomial $a(x)$. The addition step is performed by multiplying $A_i(x)$ by $b(x)$ at the ith step. We then need to multiply the partial product $c(x)$ by x^{-w} modulo $n(x)$. In order to perform this step using division, we add a multiple of $n(x)$ to $c(x)$ so that the least significant w coefficients of $c(x)$ will be zero, i.e., $c(x)$ will be divisible by x^w. Thus, if $c(x) \neq 0 \bmod x^w$, then we find $M(x)$ (which is a polynomial of length w) such that $c(x) + M(x)n(x) = 0 \pmod{x^w}$. Let $C_0(x)$ and $N_0(x)$ be the least significant words of $c(x)$ and $n(x)$, respectively. We calculate $M(x)$ as

$$M(x) = C_0(x)N_0^{-1}(x) \bmod x^w \ .$$

We note that $N_0^{-1}(x) \bmod x^w$ is equal to $N_0'(x)$ since the property (1) implies that

$$\begin{aligned} x^{sw}x^{-sw} + n(x)n'(x) &= 1 \pmod{x^w} \\ N_0(x)N_0'(x) &= 1 \pmod{x^w} \end{aligned}$$

The word-level algorithm for the Montgomery multiplication is obtained as

Word-Level Algorithm for Montgomery Multiplication
Input: $a(x), b(x), n(x), N_0'(x)$
Output: $c(x) = a(x)b(x)x^{-k} \bmod n(x)$
Step 1. $c(x) := 0$
Step 2. for $i = 0$ to $s - 1$ do
Step 3. $c(x) := c(x) + A_i(x)b(x)$
Step 4. $M(x) := C_0(x)N_0'(x) \pmod{x^w}$
Step 5. $c(x) := c(x) + M(x)n(x)$
Step 6. $c(x) := c(x)/x^w$

The word-level algorithm requires the computation of the w-length polynomial $N_0'(x)$ instead of the entire polynomial $n'(x)$ which is of length $k = sw$. It turns out that the short algorithm developed for computing n_0' in the integer case [4] can also be generalized to the polynomial case. The inversion algorithm is based on the observation that the polynomial $N_0(x)$ and its inverse satisfy

$$N_0(x)N_0^{-1}(x) = 1 \pmod{x^i}$$

for $i = 1, 2, \ldots, w$. In order to compute $N_0'(x)$, we start with $N_0'(x) = 1$, and proceed as

Inversion Algorithm
Input: $w, N_0(x)$
Output: $N_0'(x) = N_0^{-1} \bmod x^w$
Step 1. $N_0'(x) := 1$
Step 2. for $i = 2$ to w
Step 3. $t(x) := N_0(x)N_0'(x) \bmod x^i$
Step 4. if $t(x) \neq 1$ then $N_0'(x) := N_0'(x) + x^{i-1}$

5 Computation of Montgomery Squaring

The computation of the Montgomery multiplication for $a(x) = b(x)$ can optimized due to the fact that cross terms disappear because they come in pairs and the underlying field is GF(2). It is easy to show that

$$a^2(x) = \sum_{i=0}^{k-1} a_i x^{2i} ,$$

and thus, the multiplication steps in the bit-level and word-level algorithms can be skipped. The Montgomery squaring algorithm starts with the degree $2(k-1)$ polynomial $c(x) = a^2(x)$, i.e.,

$$\begin{aligned} c(x) &= a_{k-1}x^{2(k-1)} + a_{k-2}x^{2(k-2)} + \cdots + a_1x^2 + a_0 \\ &= (a_{k-1}0a_{k-2}0\cdots 0a_1 0a_0) , \end{aligned}$$

and then reduces $c(x)$ by computing $c(x) := c(x)x^{-k} \bmod n(x)$. The steps of the bit-level algorithm are illustrated below:

Bit-Level Algorithm for Montgomery Squaring
Input: $a(x), n(x)$
Output: $c(x) = a^2(x)x^{-k} \bmod n(x)$
Step 1. $c(x) := \sum_{i=0}^{k-1} a_i x^{2i}$
Step 2. for $i = 0$ to $k-1$ do
Step 3. $c(x) := c(x) + c_0 n(x)$
Step 4. $c(x) := c(x)/x$

Similarly, the word-level algorithm starts with the same polynomial $c(x)$, however, then performs the reduction steps by proceeding word by word, as follows:

Word-Level Algorithm for Montgomery Squaring
Input: $a(x), n(x), N_0'(x)$
Output: $c(x) = a^2(x)x^{-k} \bmod n(x)$

Step 1. $c(x) := \sum_{i=0}^{k-1} a_i x^{2i}$
Step 2. for $i = 0$ to $s-1$ do
Step 3. $M(x) := C_0(x)N_0'(x) \pmod{x^w}$
Step 4. $c(x) := c(x) + M(x)n(x)$
Step 5. $c(x) := c(x)/x^w$

6 Examples

We take the field GF(2^4) to illustrate the Montgomery product computation. The irreducible polynomial is selected to be $n(x) = x^4 + x + 1$. Furthermore, we have $k = 4$ and $r(x) = x^4$. Since $n = (10011)$, the special field element r is (0011). The inverse of $r(x)$ modulo $n(x)$ is computed as $r^{-1}(x) = x^3 + x^2 + x = (1110)$ using the extended Euclidean algorithm. This result is easily verified by computing

$$\begin{aligned} r(x)r^{-1}(x) &= x^4(x^3 + x^2 + x) && \pmod{x^4 + x + 1} \\ &= x^7 + x^6 + x^5 && \pmod{x^4 + x + 1} \\ &= 1 && \pmod{x^4 + x + 1} \end{aligned}$$

Furthermore, we compute $n'(x)$ using the property (1) as

$$n'(x) = \frac{1 + r(x)r^{-1}(x)}{n(x)} = \frac{1 + x^4(x^3 + x^2 + x)}{x^4 + x + 1} = \frac{x^7 + x^6 + x^5 + 1}{x^4 + x + 1} = x^3 + x^2 + x + 1 \ .$$

Let $a(x) = x^3 + x^2 + 1 = (1101)$ and $b(x) = x^3 + 1 = (1001)$. In order to compute the product $c = a \cdot b \cdot r^{-1}$ in GF(2^4), we use the algorithm for the Montgomery multiplication, and compute $t(x)$, $u(x)$, and $c(x)$ as follows:

$$\begin{aligned} \text{Step 1: } t(x) &= a(x)b(x) = (x^3 + x^2 + 1)(x^3 + 1) \\ &= x^6 + x^5 + x^2 + 1 \ . \\ \text{Step 2: } u(x) &= t(x)n'(x) = (x^6 + x^5 + x^2 + 1)(x^3 + x^2 + x + 1) \\ &= x^9 + x^4 + x + 1 = x + 1 \pmod{x^4} \ . \\ \text{Step 3: } c(x) &= [t(x) + u(x)n(x)]/r(x) \\ &= [(x^6 + x^5 + x^2 + 1) + (x + 1)(x^4 + x + 1)]/x^4 \\ &= (x^6 + x^4)/x^4 = x^2 + 1 \ . \end{aligned}$$

Thus, we conclude that $a \cdot b \cdot r^{-1}$ is equal to $c = (0101)$. This result is obtained using the bit-level algorithm without computing $n'(x)$ or $r^{-1}(x)$. The bit-level algorithm starts with $c(x) = 0$, and obtains $c(x) = x^2 + 1$ using the following steps:

			Step 3		Step 4	Step 5
i	a_i	$a_i b(x)$	$c(x) := c(x) + a_i b(x)$	c_0	$c(x) := c(x) + c_0 n(x)$	$c(x) := c(x)/x$
0	1	$x^3 + 1$	$x^3 + 1$	1	$x^4 + x^3 + x$	$x^3 + x^2 + 1$
1	0	0	$x^3 + x^2 + 1$	1	$x^4 + x^3 + x^2 + x$	$x^3 + x^2 + x + 1$
2	1	$x^3 + 1$	$x^2 + x$	0	$x^2 + x$	$x + 1$
3	1	$x^3 + 1$	$x^3 + x$	0	$x^3 + x$	$x^2 + 1$

We now illustrate the steps of the word-level algorithm to compute $c(x)$. The word-level algorithm first computes $N_0'(x)$ using the inversion algorithm. Let $w = 2$. Since $n(x) = x^4 + x + 1$, we have $N_0(x) = x + 1$. The inversion algorithm starts with $N_0'(x) = 1$, and then computes

$$t(x) = N_0(x)N_0'(x) = (x+1)(1) = x + 1 \pmod{x^2} .$$

Since $t(x) \neq 1$, the value of $N_0'(x)$ is updated as $N_0'(x) = N_0'(x) + x = 1 + x$. Therefore, we obtain $N_0'(x) = x + 1$ using the inversion algorithm. This result is easily verified since

$$N_0(x)N_0'(x) = (x+1)(x+1) = x^2 + 1 = 1 \pmod{x^2} .$$

The word-level algorithm starts with $c(x) = 0$. Since $a(x) = (1101)$, we have $A_0(x) = (01) = 1$ and $A_1(x) = (11) = x + 1$. Furthermore, $N_0'(x) = (11) = x + 1$. The steps of the word-level algorithm for computing the result $c(x) = x^2 + 1$ are given below:

$i = 0$ Step 3: $c(x) = c(x) + A_0(x)b(x) = (0) + (1)(x^3 + 1) = x^3 + 1$
Step 4: $M(x) = C_0(x)N_0'(x) = (1)(x + 1) = x + 1 \pmod{x^2}$
Step 5: $c(x) = c(x) + M(x)n(x) = (x^3 + 1) + (x + 1)(x^4 + x + 1) = x^5 + x^4 + x^3 + x^2$
Step 6: $c(x) = c(x)/x^2 = (x^5 + x^4 + x^3 + x^2)/x^2 = x^3 + x^2 + x + 1$

$i = 1$ Step 3: $c(x) = c(x) + A_1(x)b(x) = (x^3 + x^2 + x + 1) + (x + 1)(x^3 + 1) = x^4 + x^2$
Step 4: $M(x) = C_0(x)N_0'(x) = (0)(x + 1) = 0 \pmod{x^2}$
Step 5: $c(x) = c(x) + M(x)n(x) = (x^4 + x^2) + (0)(x^4 + x + 1) = x^4 + x^2$
Step 6: $c(x) = c(x)/x^2 = (x^4 + x^2)/x^2 = x^2 + 1$

Finally, we give an example illustrating the word-level Montgomery squaring algorithm. We compute $c = a \cdot a \cdot r^{-1}$ where $a = (1101) = x^3 + x^2 + 1$. The word-level Montgomery algorithm starts with $c(x) = a^2(x) = x^6 + x^4 + 1$, and performs the following steps in order to compute the final result.

$i = 0$ Step 3: $M(x) = C_0(x)N_0'(x) = (1)(x + 1) = x + 1 \pmod{x^2}$
Step 4: $c(x) = c(x) + M(x)n(x) = (x^6 + x^4 + 1) + (x + 1)(x^4 + x + 1) = x^6 + x^5 + x^2$
Step 5: $c(x) = c(x)/x^2 = (x^6 + x^5 + x^2)/x^2 = x^4 + x^3 + 1$

$i = 1$ Step 3: $M(x) = C_0(x)N_0'(x) = (1)(x + 1) = x + 1 \pmod{x^2}$
Step 4: $c(x) = c(x) + M(x)n(x) = (x^4 + x^3 + 1) + (x + 1)(x^4 + x + 1) = x^5 + x^3 + x^2$
Step 5: $c(x) = c(x)/x^2 = (x^5 + x^3 + x^2)/x^2 = x^3 + x + 1$

7 Analysis of the Word-Level Algorithm

In this section, we give a rigorous analysis of the word-level algorithm for computing the Montgomery product. We calculate the number of word-level GF(2) addition and multiplication operations. The word-level addition is simply the bitwise XOR operation which is a readily available instruction on most general purpose microprocessors and signal processors. The word-level multiplication operation receives two 1-word (w-bit) polynomials $a(x)$ and $b(x)$ defined over the field GF(2), and computes the 2-word polynomial $c(x) = a(x)b(x)$. The degree of the product polynomial $c(x)$ is $2(w - 1)$. For example, given $a = (1101)$ and $b = (1010)$, this operation computes c as

$$\begin{aligned} a(x)b(x) &= (x^3 + x^2 + 1)(x^3 + x) \\ &= x^6 + x^5 + x^4 + x \\ &= (0111\ 0010) . \end{aligned}$$

Unfortunately, none of the general purpose processors contains an instruction to perform the above operation. The implementation of this operation, which we call MULGF2, can be performed in two distinctly different ways:

- Emulation using SHIFT and XOR operations.
- Table lookup approach.

The emulation approach is usually slower than the table lookup approach, particularly for $w \geq 8$. The following function can be used to compute the 2-word result H,L given the inputs A and B. The MULGF2 algorithm given below requires $2w$ SHIFT and w XOR operations.

```
H := 0 ; L := 0
for j=w-1 downto 0 do
    L := SHL(L,1)
    H := RCL(H,1)
    if BIT(B,j)=1 then L := L XOR A
```

On the other hand, a simple method for implementing the table lookup approach is to use 2 tables, one for computing H and the other for computing L. The tables are addressed using the bits of A and B, and thus, each table is of size $2^w \times 2^w \times w$ bits. We obtain the values of H and L using two table reads. Other approaches are also possible. However, we note that these tables are different from the tables in [5, 16], which are used to implement GF(2^w) multiplications. Here we are using the tables to multiply two $(w-1)$-degree polynomials over the field GF(2) to obtain the product polynomial which is of degree $2(w-1)$.

In Table 1, we give the the number of MULGF2 and XOR operations required in each step of the word-level Montgomery multiplication algorithm.

Table 1: Operation counts for the word-level Montgomery multiplication algorithm.

	MULGF2	XOR
for i=0 to s do	-	-
C[i]:=0	-	-
for i=0 to s-1 do	-	-
for j=0 to s-1 do	-	-
MULGF2(H,L,A[j],B[i])	s^2	-
C[j]:=C[j] XOR L	-	s^2
C[j+1]:=C[j+1] XOR H	-	s^2
MULGF2(H,M,C[0],N0')	s	-
MULGF2(P,L,M,N[0])	s	-
for j=1 to s-1 do	-	-
MULGF2(H,L,M,N[j])	$s^2 - s$	-
C[j-1]:=C[j] XOR L XOR P	-	$2s^2 - 2s$
P:=H	-	-
C[s-1]:=C[s] XOR P XOR M	-	$2s$
C[s]:=0	-	-
	$2s^2 + s$	$4s^2$

We now compare the word-level Montgomery multiplication algorithm to the standard GF(2^k) multiplication using the polynomial representation. The standard GF(2^k) multiplication can be accomplished in several different ways. We select the word-level interleaving and reduction method

for comparison since this algorithm is very similar to the word-level algorithm for the Montgomery multiplication in terms of its data structure and the general flow. This algorithm computes $c = a \cdot b$ using the polynomial representation by computing $c(x) = a(x)b(x) \pmod{n(x)}$.

Word-Level Standard Multiplication Algorithm
Input: $a(x), b(x), n(x)$
Output: $c(x) = a(x)b(x) \bmod n(x)$
Step 1. $c(x) := 0$
Step 2. for $i = s - 1$ downto 0 do
Step 3. $c(x) := c(x)x^w$
Step 4. $c(x) := c(x) + B_i(x)a(x)$
Step 5. $c(x) := c(x) \pmod{n(x)}$

In Step 5, the modular reduction is performed by aligning the most significant word of $n(x)$ with the most significant word of $c(x)$, and then by performing a series of bit-level right shift and polynomial additions until the most significant word of $c(x)$ becomes zero. Table 2 gives the number of MULGF2, XOR, and SHIFT operations required in each step of the word-level standard multiplication algorithm.

Table 2: Operation counts for the word-level standard multiplication algorithm.

	MULGF2	XOR	SHIFT
`for i=0 to s do`	-	-	-
`  C[i]:=0`	-	-	-
`for i=s-1 downto 0 do`	-	-	-
`  P:=0`	-	-	-
`  for j=s-1 downto 0 do`	-	-	-
`    MULGF2(H,L,A[j],B[i])`	s^2	-	-
`    C[j+1]:=C[j] XOR H XOR P`	-	$2s^2$	-
`    P:=L`	-	-	-
`  C[0]:=P`	-	-	-
`  for j=s downto 1 do`	-	-	-
`    U[j]:=SHL(N[j],w-1) XOR SHR(N[j-1],1)`	-	s^2	$2s^2$
`  U[0]:=SHL(N[0],w-1)`	-	-	s
`  for j=w-1 downto 0 do`	-	-	-
`    if DEGREE(C)>=DEGREE(U) then`	-	-	-
`      for k=0 to s do`	-	-	-
`        C[k]:=C[k] XOR U[k]`	-	$sw(s+1)/2$	-
`    for k=0 to s-1 do`	-	-	-
`      U[k]:=SHR(U[k],1) XOR SHL(U[k+1],w-1)`	-	s^2w	$2s^2w$
`    U[s]:=SHR(U[s],1)`	-	-	sw
	s^2	$3s^2(w/2+1)+$ $sw/2$	$2s^2(w+1)+$ $s(w+1)$

As can be seen from Tables 1 and 2, the word-level Montgomery multiplication algorithm performs about twice as many MULGF2 operations as the standard algorithm, however, it requires much fewer XOR operations and no SHIFT operation. Thus, if the MULGF2 operation can be performed in a few clock cycles, the word-level Montgomery multiplication algorithm would be significantly faster. In Table 3, we tabulate the total number of operations for the Montgomery and standard multiplication algorithms for $w = 8, 16, 32$ for comparison purposes.. According to the operation counts seen in Table 3, the word-level Montgomery multiplication algorithm is only slightly faster than the standard multiplication algorithm if the MULGF2 operation is emulated using SHIFT and

XOR operations. However, with a fast implementation of MULGF2, the Montgomery multiplication algorithm is significantly faster.

Table 3: Comparing the Montgomery and standard multiplication.

	Emulation			Table Lookup		
w	Standard	Montgomery	Speedup	Standard	Montgomery	Speedup
8	$57s^2 + 13s$	$52s^2 + 24s$	1.09	$34s^2 + 13s$	$6s^2 + s$	5.67
16	$109s^2 + 25s$	$100s^2 + 48s$	1.09	$62s^2 + 25s$	$6s^2 + s$	10.33
32	$213s^2 + 49s$	$196s^2 + 96s$	1.09	$118s^2 + 49s$	$6s^2 + s$	19.67

We have also implemented these two algorithms in C, and obtained timings on a 100-MHz Intel 486DX4 processor running the NextStep 3.3 operating system. We summarize the experimental speedup values in Table 4.

Table 4: Experimental speedup values.

w	Method $k \rightarrow$	64	128	256	512	1024
8	Table Lookup	4.51	3.82	3.17	2.94	2.83
8	Emulation	1.25	1.15	1.13	1.10	1.06
16	Emulation	1.27	1.10	1.02	0.98	0.92
32	Emulation	2.35	1.69	1.34	1.18	1.08

As was mentioned, the table lookup approach can be implemented using 2 tables each of which is of size $2^w \times 2^w \times w$ bits. For $w = 8$, each of the tables is of size 64 Kilobytes. For $w = 16$, the table size increases to $2^{16} \times 2^{16} \times 16$ bits, or 8 Gigabytes. Thus, we have implemented the table lookup MULGF2 algorithm for only $w = 8$.

8 Conclusions

We have described the bit-level and word-level algorithms for computing the Montgomery product $a \cdot b \cdot r^{-1}$ in the field GF(2^k). It turns out that this operation would be significantly faster in software with the availability of a fast method for multiplying two w-bit polynomials defined over GF(2), where w is the wordsize. This can be achieved using a table lookup approach when the wordsize is small; another method is to implement an instruction on the underlying processor for performing this operation which is much simpler than the integer multiplication due to the lack of carry propagation.

The Montgomery multiplication can be used to obtain fast software implementation of the exponentiation over the field GF(2^k). Let the field element a and the m-bit positive integer e be given. In order to compute $c = a^e \in$ GF(2^k), we can use the binary method [6]. The algorithm first computes $\bar{c} = 1 \cdot r$ and $\bar{a} = a \cdot r$ using the standard multiplication, and then proceeds to compute c using only Montgomery squarings and multiplications.

for $i = m - 1$ downto 0 do
  $\bar{c} := \bar{c} \cdot \bar{c} \cdot r^{-1}$
  if $e_i = 1$ then $\bar{c} := \bar{c} \cdot \bar{a} \cdot r^{-1}$
$c := \bar{c} \cdot 1 \cdot r^{-1}$

We are currently working on implementing the Montgomery exponentiation algorithm, and extending the Montgomery multiplication and squaring to the normal bases.

References

[1] G. B. Agnew, R. C. Mullin, I. Onyszchuk, and S. A. Vanstone. An implementation for a fast public-key cryptosystem. *Journal of Cryptology*, 3(2):63–79, 1991.

[2] G. B. Agnew, R. C. Mullin, and S. A. Vanstone. An implementation of elliptic curve cryptosystems over $F_{2^{155}}$. *IEEE Journal on Selected Areas in Communications*, 11(5):804–813, June 1993.

[3] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22:644–654, November 1976.

[4] S. R. Dussé and B. S. Kaliski, Jr. A cryptographic library for the Motorola DSP56000. In I. B. Damgård, editor, *Advances in Cryptology — EUROCRYPT 90*, Lecture Notes in Computer Science, No. 473, pages 230–244. New York, NY: Springer-Verlag, 1990.

[5] G. Harper, A. Menezes, and S. Vanstone. Public-key cryptosystems with very small key lengths. In R. A. Rueppel, editor, *Advances in Cryptology — EUROCRYPT 92*, Lecture Notes in Computer Science, No. 658, pages 163–173. New York, NY: Springer-Verlag, 1992.

[6] D. E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Reading, MA: Addison-Wesley, Second edition, 1981.

[7] N. Koblitz. *A Course in Number Theory and Cryptography.* New York, NY: Springer-Verlag, 1987.

[8] R. Lidl and H. Niederreiter. *Introduction to Finite Fields and Their Applications.* New York, NY: Cambridge University Press, 1994.

[9] R. J. McEliece. *Finite Fields for Computer Scientists and Engineers.* Boston, MA: Kluwer Academic Publishers, 1987.

[10] A. J. Menezes, editor. *Applications of Finite Fields.* Boston, MA: Kluwer Academic Publishers, 1993.

[11] A. J. Menezes. *Elliptic Curve Public Key Cryptosystems.* Boston, MA: Kluwer Academic Publishers, 1993.

[12] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.

[13] R. Mullin, I. Onyszchuk, S. Vanstone, and R. Wilson. Optimal normal bases in $GF(p^n)$. *Discrete Applied Mathematics*, 22:149–161, 1988.

[14] J. Omura and J. Massey. Computational method and apparatus for finite field arithmetic. U.S. Patent Number 4,587,627, May 1986.

[15] R. Schroeppel, S. O'Malley H. Orman, and O. Spatscheck. Fast key exchange with elliptic curve systems. In D. Coppersmith, editor, *Advances in Cryptology — CRYPTO 95*, Lecture Notes in Computer Science, No. 973, pages 43–56. New York, NY: Springer-Verlag, 1995.

[16] E. De Win, A. Bosselaers, B. Preneel, and P. De Gersem. A fast software implementation for arithmetic operations in GF(2^n). Manuscript, February 1996.

A Parallel Implementation of RSA

David Pearson
Computer Science Department
Cornell University
Ithaca, NY 14853
pearson@cs.cornell.edu

July 22, 1996

Abstract

Performing RSA public and private key operations fast is increasingly important.. In this paper I describe an efficient implementation of RSA for a highly parallel computer. I present a new algorithm for modular multiplication using a residue number system (RNS) and a variation of Montgomery's method. The heart of the algorithm is a new method for converting from one RNS to another.

1 Introduction

As public-key cryptography has moved from an academic sideline to center-stage in the burgeoning world of electronic commerce, the computational speed of performing public-key operations has become a significant concern. Every major system for public-key cryptography has a very high computational cost, in contrast to the many quite speedy private-key systems available. Despite this drawback, public-key systems like RSA are likely to become increasingly widely used; fast implementations are therefore extremely desirable.

There are several avenues for speeding up RSA:

- Faster sequential algorithms
- Faster clock rates
- Special-purpose hardware
- Parallel computers and algorithms

This article focuses on the last of these options, parallel computing, since it seems to provide the greatest potential for speedup over the long term. I will present a novel algorithm for modular exponentiation which is particularly well-suited to parallel machines, and describe an implementation of it.

The other avenues seem less promising. Sequential algorithms may well continue to improve, but we are unlikely to see an order-of-magnitude speedup. Faster clock rates will certainly help—Moore's law is likely to hold for at least the next 15 years. There are two factors working against this, however: internet traffic, and presumably also secure traffic, is increasing even faster than computer speeds; and Moore's law is also helping the codebreakers and factoring algorithms, perhaps even more than it helps encryption, since they can use parallelism very effectively, as Lenstra and others have so convincingly demonstrated by breaking the original RSA challenge message [5]. Current sequential algorithms require $O(n^3)$ time for private-key operations, so we use up our cycles with more costly operations.

I view Special-purpose hardware as simply another way to get parallelism. The current hardware implementations all use some form of parallelism, even if it's only having very wide ALUs, and the algorithm I describe here could certainly be even faster on specially designed hardware. But I think the appeal of such a solution will never be as wide as a piece of software. I doubt there will be a time when every computer comes equipped with RSA in hardware, but I can imagine a day when every computer comes with several processors.

2 Opportunities for parallelism

Recall the basic operations of RSA [7]: the public key (e, m) consists of an exponent and a modulus, and the encryption operation transforms a message $t < m$ into $t^e \pmod m$. The private key (d, m) uses the same modulus but a different exponent; the operation is the same, modular exponentiation. The modulus m is chosen to be the product of two large prime numbers p and q, and e is usually chosen to be 3 or some other small number, so that public-key operations are reasonably fast (about $O(n^2)$ operations, where n is the size of the modulus). The private exponent, d, is of the same order as m, so private key operations require $O(n^3)$ time. Because of the great speed disparity, I will focus on speeding up private-key operations.

Given a message t, a modulus m, and an exponent $e = \sum_{i=1}^{n} e_i 2^i$, the basic modular exponentiation algorithm used by RSA loops over each bit e_i of the exponent:

1. $x := t$
2. $v := 1$
3. for $i := 1$ to n
4. if $e_i = 1$ then $v := vx \pmod m$
5. $x := x^2 \pmod m$

The final value of v is the result, $t^e \pmod{m}$.

There does not seem to be any way to perform modular exponentiation faster than the method of repeated squaring, so there appears to be an inherent sequentiality to the main loop of RSA. Nevertheless, there are four clear opportunities for parallelism in this basic algorithm:

- If there is a stream of messages to be decrypted (or signed), each of these operations may be performed independently on a different processor (or set of processors). This will not speed up the elapsed time for a single private-key operation, but it can greatly speed up the overall throughput.

- Step 5, squaring, can be performed in parallel with the multiplication in step 4 from the previous iteration, saving some 33%. Note that this is only possible with the loop running from low-order exponent bits to high-order. It is more common to scan the exponent in high-to-low order, but doing so removes some inherent parallelism.

- For private-key operations, we may assume that the factors p and q of m are known. The modular exponentiation can be performed separately and in parallel mod p and mod q, and then the two results can be combined by the Chinese remainder theorem. This is often done in fast software implementations, since two half-size modular exponentiations are still four times as fast as one full-size exponentiation.

- Finally, the high-precision multiplications in steps 4 and 5 are slow ($O(n^2)$) operations. Using standard long-multiplication with 32-bit words and a 512-bit modulus, there are 16 words in each number and $16^2 = 256$ 32×32 bit multiplies to perform. These multiplies can all be performed and the results summed in parallel. Note that although there are asymptotically faster multiplication algorithms, including FFT and Karatsuba [2], they do not appear to be competitive until much larger numbers are used than RSA [3]. Unlike the other methods, parallel multiply requires an architecture that supports fine-grained parallelism—it is not practical if the communication overhead between processors is much larger than the multiplication time.

A good parallel implementation of RSA will certainly incorporate the first three of these ideas. The first, performing several RSA operations independently on different processors, is the most scalable, and arguably the most important. But we would still like to speed up the response time of a single operation, for which the other three techniques are needed. Performing squaring in parallel with multiplication and exponentiating modulo the two prime factors together gain only a factor of 3. The rest of this paper investigates variations on the last idea—using parallelism to speed up multiplication.

3 Modular multiplication

The operation we must perform in RSA is not simple multiple-precision multiplication, but *modular* multiplication: $xy \pmod m$. The sequential implementation of this operation generally requires about twice the time of simple multiplication, since each $1 \times n$ multiplication step is followed by a $1 \times n$ modular reduction step. This interleaved method is inherently sequential, and has no obvious analogue in the parallel multiplication described above. There are parallel algorithms for division and remainder that are approximately as efficient as multiplication, and in fact use multiplication as a subroutine [4]. To divide two numbers one can use Newton's method to form an approximate reciprocal of the denominator and multiply it by the numerator. In the case of RSA, the reciprocal approximation need be computed only once. The algorithm for modular multiplication then becomes:

$$
\begin{aligned}
s &:= xy \\
q &:= m^{-1}s \\
r &:= s - qm
\end{aligned}
$$

The intermediate products s and qm require twice as much space as the modulus. The quotient q is only approximate, so a correction step is normally necessary, but the remainder will still be in the same residue class mod m, so we can accommodate the possible error by allowing an extra one or two bits of precision for the intermediate results, and only doing the full reduction on the last step.

This is indeed of the same order of complexity as multiplication, but is slower by a factor of 3, since each step requires a high-precision multiplication (on a sequential machine, this would be a factor of 2, since we need only half the bits of q and qm, but on a parallel machine it is almost as fast to compute the entire product). Furthermore, each multiplication depends on the results of the previous one, so they must be done sequentially. My first implementation used this method, and it can be competitive with hardware implementations, but we can do better. The tools we need to do so are Montgomery's method for modular multiplication (used in many software implementations of RSA) and residue number systems (also called modular arithmetic) where high-precision numbers are represented by their residues modulo a collection of small relatively-prime numbers. Neither of these techniques individually seems to help, but when combined, and with the help of a novel method for converting one residue number system to another, there is a surprising synergy. The final implementation is very close to three times as fast as the first—almost the speed of a single high-precision multiplication.

3.1 Montgomery's method

The purpose of the modular reduction after each multiply is to avoid the exponential growth in the size of the intermediate results, adding on a multiple of the modulus in

order to zero the high-order bits of the product. Montgomery [6] used an idea originally due to Hensel at the turn of the century, which is to zero instead the *low-order* bits of the product, and shift the remaining bits down. The residue classes are still represented uniquely, though it requires a final modular multiplication by a constant to restore the result to its normal representation.

Let $w = 2^n$ be at least $4m$, and choose m' such that $m'm \equiv -1 \pmod{w}$. Montgomery's method for modular multiplication works as follows:

$$\begin{aligned} s &:= xy \\ q &:= m's \pmod{w} \\ r &:= (s + qm)/w \end{aligned}$$

Once again, the intermediate products s and qm have twice as many digits as m, x, y, or r. Since $qm \equiv -s \pmod{w}$, $(s + qm)$ will always be a multiple of w. And it is easy to verify that if both x and y are less than $w/2$ then r will be also.

Reducing modulo w and dividing by w are trivial operations for multiple-precision binary numbers, but it seems we haven't really made any progress, since this still requires three multiplications, just like the first method. But now we translate this method into a new representation.

3.2 Residue number systems

In a residue number system (RNS), a high-precision integer is represented by its residues modulo some set of integers. Multiplication, addition, and subtraction can all be performed element-wise, but comparison and division are in general difficult. Our plan is to implement Montgomery's method in residue arithmetic. Notice that Montgomery's method doesn't depend on w being a power of 2, it simply makes it easy to reduce mod w and divide by w. In residue arithmetic we will choose w to be a product of a subset of the moduli.

Let $v_1, v_2, \ldots, v_n, w_1, w_2, \ldots, w_n$ be relatively prime, and let $v = v_1 v_2 \cdots v_n$ and $w = w_1 w_2 \cdots w_n$. Given a number $x < v$ we can represent it by its residues: $[x_{v_1}, x_{v_2}, \ldots, x_{v_n}]$ where $x_{v_i} = x \pmod{v_i}$, and similarly for w. We will choose enough moduli so that $v > 4m$ and $w > 4m$: then either the v_i or the w_i can represent a reduced product, and we can use both together to represent a full product. Now the three multiplies are much simpler elementwise operations (they might be, say, 16×16 bit operations, reduced mod a 16-bit number). We have two more troublesome operations, though—reducing mod w in the second step ($q = m's \pmod{w}$), and dividing the remainder by w in the final step. But these can be performed using the operation of *converting* a number in one RNS (say the w_j) to another (the v_i). Denote this operation by $[x_{v_1}, x_{v_2}, \ldots, x_{v_n}] \leftarrow [x_{w_1}, x_{w_2}, \ldots, x_{w_n}]$. Assume we have precomputed the residue representation mod the w_j of $m' = [m'_{w_1}, m'_{w_2}, \ldots, m'_{w_n}]$ and similarly the representations mod the v_i of m and w^{-1}

(the inverse of w exists mod v_i because w and v are relatively prime). Here are the steps in doing a modular multiplication of two numbers represented mod the w_j, with the result represented mod the v_i:

Input: $[x_{w_1}, x_{w_2}, \ldots, x_{w_n}], [y_{w_1}, y_{w_2}, \ldots, y_{w_n}]$
Output: $[r_{v_1}, r_{v_2}, \ldots, r_{v_n}]$ where $r = xy \pmod m$

1. $[x_{v_1}, x_{v_2}, \ldots, x_{v_n}] \leftarrow [x_{w_1}, x_{w_2}, \ldots, x_{w_n}]$
2. $[y_{v_1}, y_{v_2}, \ldots, y_{v_n}] \leftarrow [y_{w_1}, y_{w_2}, \ldots, y_{w_n}]$
3. $s_{w_j} := x_{w_j} y_{w_j} \pmod{w_j}$ for $1 \le j \le n$
4. $s_{v_i} := x_{v_i} y_{v_i} \pmod{v_i}$ for $1 \le i \le n$
5. $q_{w_j} := s_{w_j} m'_{w_j} \pmod{w_j}$ for $1 \le j \le n$
6. $[q_{v_1}, q_{v_2}, \ldots, q_{v_n}] \leftarrow [q_{w_1}, q_{w_2}, \ldots, q_{w_n}]$
7. $r_{v_i} := (s_{v_i} + q_{v_i} m_{v_i}) w_{v_i}^{-1} \pmod{v_i}$ for $1 \le i \le n$

Since step 3 computes a representation of $s \pmod w$ and step 4 computes $s \pmod v$, if we combined the two with the Chinese remainder theorem, we would have the correct value of $s \pmod{vw}$ which is just s. It is also easy to see that steps 5 and 6 produce the same value of q as standard Montgomery's method. The number r represented by the r_{v_i} in step 7 has the properties that (1) $r < v$, (2) $rw \equiv s + qm \pmod v$ and (3) trivially $rw \equiv 0 \pmod w$. Since $s + qm \equiv 0 \pmod w$, (2) and (3) together give us that $rw = s + qm$, so r is the correct quotient $(s + qm)/w$.

We are left with the problem that the algorithm started with the inputs mod the w_j and ended with a result represented mod the v_i. But we don't need to convert back again—alternate iterations of the main exponentiation loop can switch back and forth between w and v. Several other ways that the algorithm seems cumbersome also turn out to be easier than they look. For example, the two RNS conversions in steps 1 and 2 are never both necessary—for squaring they are both the same number, and the other modular multiply is by a number that is also being squared, so its RNS conversion is available for free. Although there are still two global operations (the RNS conversions) instead of three for the straightforward modular multiply, these two are independent and can be performed in parallel. And finally, the RNS conversion algorithm allows us to fold the multiplies in steps 5 and 7 into the conversion process without any penalty, as we will see shortly.

3.3 Residue conversion

Converting from one residue number system to another was discussed by Hitz and Kaltofen [1] in the context of division in a RNS. Unfortunately, their algorithm is both fairly complex and computationally costly, essentially forming the full multiple-precision value

(in a mixed-radix system) of the number represented by the residues, then converting that to the output RNS. I describe a new algorithm here that is both faster and simpler.

The starting point for this algorithm is also an application of the Chinese remainder theorem to generate the large integer represented by the residues. But as we go, we reduce that integer mod each of the moduli in the output RNS. It will turn out that almost all the work can be done on small integers. By the CRT, the number represented by $[x_{w_1}, x_{w_2}, \ldots, x_{w_n}]$ is:

$$x = \left(\sum_{j=1}^{n} c_{w_j} x_{w_j} \right) \pmod{w}$$

for easily computed constants c_{w_j}. For now, ignore the final reduction of this sum mod w, and let x' be the unreduced sum. We can compute the value of x' mod v_i as a simple inner product of small integers, modulo a small integer:

$$x' \pmod{v_i} = \left(\sum_{j=1}^{n} (c_{w_j} \pmod{v_i}) x_{w_j} \right) \pmod{v_i}$$

But we cannot ignore the mod w. To account for it, we have to subtract $w \lfloor x'/w \rfloor$ from x', or subtract this amount mod v_i from each x_{v_i}. Notice that $\lfloor x'/w \rfloor$ is not very large—it is bounded by the sum of the w_j since $c_{w_j}/w < 1$, so the necessary adjustment is also a small integer. Although the exact value of x'/w may be difficult to compute, we can get an approximation of its value with a fractional inner product like the one above. With, say, 18-bit moduli and a 32-bit approximation, we would obtain the correct value of the integer part almost always. Since we can distinguish the cases in which our approximation might be wrong, we can handle it as an exceptional case and compute the exact value. This will occur so rarely that the average run-time will be affected insignificantly.

We can now combine the various multiplications by constants into this residue conversion process. If we wish to multiply x by some constant k mod w before converting to v, we can simply use the coefficients $kc_{w_j} \pmod{w}$ instead of c_{w_j} in the above. Note that this affects the computation of $\lfloor x'/w \rfloor$, since the x' uses the new coefficients. To multiply by some other constant l mod v *after* converting to v, we can multiply $l \pmod{v_i}$ into every coefficient of the inner product to compute x_{v_i} (reducing the coefficient modulo v_i) and into the adjustment coefficient $w \pmod{v_i}$ that we multiply by $\lfloor x'/w \rfloor$. The full formula, then, for converting from x represented in the w_j system to $y = l(kx \pmod{w}) \pmod{v}$ represented in the v_i system is this:

$$a = \left\lfloor \sum_{j=1}^{n} x_{w_j} \frac{kc_{w_j} \pmod{w}}{w} \right\rfloor$$

$$y_{v_i} = \left(\sum_{j=1}^{n} x_{w_j} l(kc_{w_j} \pmod{w}) \right) - la \pmod{v_i}$$

One last detail: this allows all the multiplications by constants to be folded into the conversion except one: multiplying s by w^{-1} in the v_i residue system, in step 7 of the algorithm. But if w^{-1} has a square root mod v, we can fold that into the conversions of x and y in steps 1 and 2, and when we take the product (or square) in step 4, the product s will effectively have been multiplied by w^{-1}. An easy way to ensure that w^{-1} has a square root is to make all the moduli $w_1, w_2 \ldots, w_n$ be squares, which is what our implementation does.

4 Implementation

The algorithm described has been implemented, though on a parallel machine which is as yet only simulated. The machine is a 3-dimensional mesh of extremely simple, identical, locally-connected processors, each with 1K bytes of local memory. On this machine, about 56 cycles are required for each modular multiply with a 256-bit modulus (i.e. with one of the two prime factors in a 512-bit key). A full decryption therefore requires about 15000 cycles, or 150 microseconds with a 100MHz clock. This is somewhat better than the fastest reported hardware implementation [8], which can do a 512-bit private-key operation in about 850 μsec (with a 40MHz clock—if the clock rate were 100MHz, the time would be 340 μsec). Since that implementation uses essentially an $O(n^2)$ algorithm, this parallel algorithm (which is approximately $O(n^{1.5})$) will have an increasing advantage as key sizes grow to 1024 bits or more.

We use 16 moduli in each v and w, with all the moduli less than 2^{17}. The target machine has no hardware multiply. For efficiency in the RNS conversion process we multiply with table lookup, reading 3 bits of x_{w_j} at a time and looking up the product of those three bits with the (constant) coefficient in an 8-word table. Since there are only 6 such lookups per multiply, we have a different table for each position, with the product pre-shifted and reduced mod v_i. The final sum for the entire inner product is at most $6 \times 16v_i$, so fully reducing it mod v_i can be done very quickly with a table lookup on the high order bits.

The approximate value of x/w is computed similarly, using tables during the multiplication. Rather than have a complex handler for the exceptional case that the result is too close to an integer to call, we compute the fraction with full precision (down to an error of $1/w$), but propagate the carry without waiting for the lower-order results whenever we can, so we gain in simplicity and speed, but pay a modest penalty in processors whose computation is rarely needed.

5 Conclusions

I have described a new algorithm for modular exponentiation, and shown that it is well suited to parallel implementation. Although using a massively parallel machine to do

RSA private-key operations may seem now to be in the realm of science fiction, I believe it is reasonable to look forward to a day when parallelism is ubiquitous.

Although it was designed for parallel computers, there are some features of this algorithm that may make it suitable for software implementation on a sequential machine. The fact that most of the work is essentially an integer matrix multiplication suggest that it would be suitable for vector machines. On RISC processors, where integer multiplication is expensive, the fact that the coefficients are constant in the matrix allows more extensive use of table lookup than the classic software implementations.

Another fruitful area for further work is implementing this algorithm on other parallel machines. It is suitable for architectures with fine-grained parallelism and low-overhead communication between nodes, like the Connection Machine, the MASPAR, or systolic arrays. It might be possible to achieve a lower degree of parallelism on more standard systems, like the IBM SP/2.

It is possible to implement this algorithm in hardware, too—the peculiar arithmetic tricks used are mostly a way to coax more parallelism from the computation of modular multiplication. But in the long run, I think hardware implementations are not the right approach. The most significant source of parallelism in RSA was mentioned at the beginning—the ability (for a network server, say) to work on different private key operations independently. With special-purpose hardware, this is no longer a very good option. If there are too many requests, the finite number of RSA chips (probably just one) is exceeded, if too few, the hardware sits idle, unable to participate in any other needed tasks. The vision of computers as universal machines is lost. But the main advantage of special-purpose hardware is to take advantage of parallelism, and whenever we can do that in software, we will gain the speed while retaining the flexibility in allocation of resources.

6 Acknowledgements

I would like to thank Susan Landau for helping clarify my thinking, in one case by merely *intending* to meet, and later by actually discussing the algorithm. Dexter Kozen and the anonymous reviewers suggested several improvements to the presentation. This work was supported by a Fannie and John Hertz Foundation fellowship.

References

[1] Markus A. Hitz and Erich Kaltofen. Integer division in residue number systems. *IEEE Transactions on Computers*, 44(8):983–989, August 1995.

[2] Donald E. Knuth. *The Art of Computer Programming*, volume 2: Seminumerical Algorithms. Addison-Wesley, 1991.

[3] Çetin Koç. High-speed RSA implementation. Technical Report TR-201, RSA Laboratories, November 1994.

[4] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann, 1992.

[5] Arjen Lenstra. Factoring. In Gerard Tel and Paul Vitányi, editors, *Distributed Algorithms*, pages 28–38. Springer-Verlag, September 1994.

[6] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521, April 1985.

[7] Ronald L. Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[8] Mark Shand and Jean Vuillemin. Fast implementations of RSA cryptography. In *Proceedings, 11th Symposium on Computer Arithmetic*, pages 252–259. IEEE, June 1993.

Sparse RSA Secret Keys and Their Generation

Chae Hoon Lim*
Center for Advanced Crypto-Technology, Baekdoo InfoCrypt, Inc.
96-14, Chungdam-Dong, Kangnam-Gu, Seoul, 135-100, Korea
E-mail: chlim@baekdoo.co.kr

Pil Joong Lee†
Dept. E. E., Pohang Univ. of Science & Technology (POSTECH)
Pohang, 790-784, Korea, E-mail: pjl@vision.postech.ac.k

Abstract

In this paper we consider the problem of reducing the computational load by use of restricted key parameters in the RSA system. We present various methods for generating RSA key parameters that can produce the secret key with much smaller binary weight than the ordinary case. This will greatly reduce the number of multiplications required for RSA decryption in both software and hardware implementations. Security will be the most critical issue for their practical use. We also present preliminary analysis for every possible attack we could imagine.

1 Introduction

Most public key systems widely used in practice, such as RSA [38], Diffie-Hellman [11] and ElGamal [14], are based on the difficulty of factoring integers and computing discrete logarithms. There are subexponential time algorithms to solve these problems. The most notable algorithm is the general number field sieve, which has the best asymptotic running time estimate (see [21] for a collection of related papers). The progress of technology thus forces us to use bigger and bigger key parameters to attain an adequate level of security even if algorithmic improvements are not taken into account. For example, 512-bit RSA moduli, which are used in a variety of commercial RSA implementations, are already unsafe [31], and it is commonly recommanded to use RSA moduli of at least 768 bits, preferably 1024 bits. The situation is similar to the case of prime moduli for ElGamal-type systems, since computing discrete logarithms modulo a prime is almost as difficult as factoring a hard integer of the same size.

*This work was done while he was a Ph.D student in the Dept. E.E., POSTECH.

†On sabbatical leave during 1996. He is now with NEC Research Institute in Princeton, NJ, USA, E-mail: pjl@research.nj.nec.com.

The use of larger moduli inevitably increases both computation and communication complexity. In particular, computation complexity grows much faster in the RSA system as the size of moduli increases. For example, RSA signature generation with a 1024-bit modulus requires roughly 7 times as much computation time as that with a 512-bit modulus. On the other hand, in the discrete log cryptosystem, we can make use of a relatively small size of exponents (e.g., 160 bits) without loss of security [41], though prime moduli should be almost as large as RSA moduli to achieve the same level of security. This can hardly hold true for RSA, since a fatal security problem arises with secret keys of size less than a certain minimum (see Section 4). However, we may consider the possibility of using secret keys of small weight to reduce the number of multiplications required. The purpose of this paper is to present such a method for generating sparse secret keys.

Our research on this subject was motivated by the recent work due to Vanstone and Zuccherato [43, 44]. They presented several methods for specifying some of the bits in the RSA modulus ahead of time (Later, some of them were shown to be insecure [8, 23]). This provides a way to reduce the storage requirement and transmission bandwidth, since publicly available data, such as user's identity information, can be used as the prespecified bits. This work led us to consider another interesting problem. Rather than storage and transmission efficiency, our interest in this paper is to enhance computation efficiency. In other words, we would like to reduce the number of multiplications required for RSA signature generation. For this, we consider various ways of generating short and/or sparse secret keys without compromising security. Furthermore, to simplify the modular reduction process, a predetermined number of the most significant bits of the modulus will be made all ones.

The rest of this paper is organized as follows. We first describe in Section 2 some preliminaries and motivations on this work. In particular, we describe a simple modular reduction algorithm using a modulus of diminished radix form. In Section 3 we present three methods for generating sparse secret keys. Each method has the same number of bits that can be predetermined, but places these bits in different parts of the secret key. Section 4 applies the methods of Section 3 to the case of generating smaller secret keys. In Section 5 we describe a method for generating sparse secret keys for use with the Chinese remainder theorem. The security of the resulting secret keys are further discussed in Section 6, and Section 7 contains some implementation results. We finally conclude in Section 8.

2 Preliminaries and Motivations

2.1 Reducing Number of Multiplications

Let $n = pq$ be an RSA public modulus, where p and q are large primes. Given a public exponent e relatively prime to both $p-1$ and $q-1$, the secret exponent d can be computed by $d = e^{-1} \bmod \lambda(n)$, where $\lambda(n)$ is given by the least common multiple of $p-1$ and $q-1$, i.e., $\lambda(n) = \mathrm{lcm}(p-1, q-1)$. The public exponent e can be taken as small as 3 in most applications[1] but the secret exponent d has almost the same size as $\lambda(n)$. Thus, the computation of $y = x^d \bmod n$ requires a lot of modular multiplications.

[1] When the RSA system is used in encryption mode, for example to transmit a session key for a symmetric algorithm, the use of small public exponents may be dangerous in some applications if suitable precautions are not taken. See [9, 10] for details.

Let l_d be the bit-length of d and w_d be the Hamming weight of d. The standard binary (square-and-multiply) algorithm can perform the exponentiation $y = x^d \bmod n$ in $l_d + w_d - 2$ modular multiplications. For random values of d, w_d is expected to be a half of l_d. One can further reduce w_d to approximately $\frac{l_d}{3}$ using a minimal weight signed binary encoding (e.g., see [18, 47, 2]). This signed binary algorithm, of course, requires the computation of $x^{-1} \bmod n$.

With some precomputations based on x, we may scan a number of bits, instead of one bit, at a time, resulting in the sliding window algorithm. It can be shown that the number of multiplications required by the window method is $\frac{w+2}{w+1} l_d + 2^{w-1} - w - \frac{1}{2^{w-1}}$ on average and $\frac{w+1}{w} l_d + 2^{w-1} - 2$ in the worst case, where w denotes the window size. The optimal value of w can be determined from a given l_d. For example, the choice of $w = 6$ for $l_d = 768$ minimizes the above two functions, giving 903.7 and 926.0, respectively.

The most economical way to evaluate powers is to use addition chains. This approach is particularly advantageous when the exponent is fixed as in the RSA system. However, finding the shortest addition chain is known to be an NP-complete problem [12]. It is reported in [5] that heuristic algorithms can compute addition chains of length around 605 for 512-bit moduli. (Recently, Koç has shown that the sliding window technique with a more sophiscated partitioning strategy can reduce the expected number of multiplications to 595 for 512-bit moduli [19].)

Despite of extensive research on this topic, no algorithm is developed that can substantially reduce the required number of multiplications.[2] The window algorithm is most widely used, in particular for software implementations, due to its simplicity and high efficiency. Obviously, a better performance could be obtained in any algorithm by reducing the binary weight of the exponent. This paper aims at investigating such possibilities to generate sparse RSA secret keys. We present several methods in which almost a half of the bits in d can be specified during key generation. This allows us to substantially reduce the binary weight of d by taking the prespecified bits as a very sparse random number. For further improvements we also consider a method for producing short and sparse secret keys and for producing sparse secret keys for the Chinese remainder technique.

2.2 Speeding up Modular Reduction

The computation time for modular exponentiation can be further reduced by speeding up multiplication and modular reduction processes. Multiplication can be accelerated by the divide-and-conquer approach based on Karatsuba's idea (see [25, pages 278–279]). More complicated is the modular reduction process. There are many algorithms for this operation. Examples are the classical algorithm [25, section 4.3.1], Barret's algorithm [3] and Montgomery's algorithm [30, 13] (see [6] for comparison of these three algorithms). The fastest known algorithm, if only the reduction operation is considered, is the Montgomery reduction. Though requiring pre- and post-transformations of the target number, this algorithm will be efficient enough to compensate for such overhead if a number of modular multiplications should be done for the same base as in modular exponentiation.

[2] We note that when the base is fixed (as in the discrete log cryptosystem), it is possible to substantially reduce the computation time at the cost of storage for a precomputation table (see [7, 39, 24] for such methods).

On the other hand, a special form of modulus can greatly simplify the modular reduction process. Suppose that the RSA modulus n is represented in base b notation as

$$n = n_{k-1}b^{k-1} + \cdots + n_1 b + n_0, \quad 0 \le n_i \le b-1.$$

The most obvious choice for b will be a power of 2 determined by the programming language to be used (e.g., $b = 2^{16}$ or 2^{32} for high-level language implementations in most computers). A modulus n is said to be of *diminished radix form* (a *DR modulus*, for short) if it satisfies

$$n = b^k - n' \ (n' < b^{k'}, \ k' < k).$$

This form of moduli makes the modular reduction process extremely simple. Suppose that we want to reduce a $2k$-digit integer x modulo n. From the congruence $b^k = n' \bmod n$, we can easily see that the following simple algorithm does perform the required reduction. For notational convenience, let $x[i:j] = x_i b^{i-j} + \cdots + x_j$ $(i > j,\ x[i:i] = x_i)$.

```
for(i = 2k − 1; i >= k; i = i − 1)
    x[i − 1 : i − k] = x[i − 1 : i − k] + x_i n';
    if(final carry)
        x[i − 1 : i − k] = x[i − 1 : i − k] + n';
```

If $x[k-1:0]$ is larger than n at the end of the algorithm, we then need to subtract n from this value to obtain the final result. Note, however, that in the case of modular exponentiation, this correction is unnecessary for intermediate reduction steps. The performance of this algorithm heavily depends upon the probability of the 'if' condition being true. If n' is δ digits less than n, i.e., $k' = k - \delta$, then this probability is expected to be less than $b^{-(\delta-1)}$ on average, since the final carry will occur only if $x[i-1:i-\delta+1]$ consists of all 1's and there is an incoming carry to $x_{i-\delta+1}$. Thus, with an appropriate choice of δ (e.g., $\delta \ge 2$), we can avoid the addition in most cases. Consequently, the above algorithm only requires $k(k-\delta)$ multiplications. By comparison, Mongomery's reduction requires $k(k+1)$ multiplications (this number can be reduced to k^2 if $n_0 = b-1$ so that $n_0^{-1} = -1 \bmod b$). We note that there exists a modular multiplier design based on a similar idea [33].

The possibility of using a DR modulus for fast reduction was first suggested by Mohan and Adiga [29]. They proposed to use an RSA modulus n of the form $n = b^k - n'$ with $n' < b^{k/2}$. Modular reduction would then require just two multiplications of $\frac{k}{2}$-digit numbers. However, Meister [27] showed that this choice of modulus may be insecure due to insufficient choices for prime factors of n. In the process of key generation, we will force the first h bits of n to have all 1's, so that the above simple reduction method can be used. The size of h may be determined from a particular base used. For example, it would be reasonable to choose $h = m\lceil \log_2 b \rceil$ with $m = 2 \sim 4$.

3 Generating Sparse Secret Keys

This section describes three methods for generating RSA key parameters. The first method produces a secret key d having small binary weight in the low order bits of d, and the second method produces d having small weight in the high order bits. The third method is a hybrid

scheme using the techniques of the above two methods and produces a secret key that is sparse in some of the high order and low order bits. All three methods have the same number of bits that can be predetermined, but have different requirements on their binary weight for security.

In the subsequent descriptions, we will assume that the prime factors of n, p and q, are chosen so that $\gcd(p-1, q-1) = 2$ and thus $\lambda(n) = \frac{1}{2}(p-1)(q-1)$ (the case of having a larger gcd will be considered in Section 4). Furthermore, given a public exponent e such that $\gcd(e, p-1) = \gcd(e, q-1) = 1$, the secret exponent d is assumed to be computed by $d = \frac{1+\rho\lambda(n)}{e}$, where ρ is an odd integer less than e (hereafter, all divisions will denote integer divisions). For this, we will choose the modulus n such that $\rho\lambda(n) = -1 \bmod e$. When counting the number of bits, we assume for simplicity that the product of a-bit and b-bit numbers yields a number $a+b$ bits long. Recall that we denote by l_x the bit-length of x (i.e., $l_x = |x|$). We also assume that $l_p = l_q = \frac{l_n}{2}$.

3.1 Making the Last t Bits Sparse

Suppose that we would like the low order t bits of d to be a sparse random number f (which must be odd). Let $m = e2^t$. Then we have to find primes p and q such that $\frac{1}{2}(p-1)(q-1)\rho = ef - 1 \bmod m$. For this, we first generate a random prime p of length $l_p = \frac{l_n}{2}$ such that $\frac{p-1}{2}$ is odd and relatively prime to e, and an odd integer $\rho < e$. Then we compute $g = (\frac{p-1}{2}\rho)^{-1} \cdot (ef-1) \bmod m$ so that the congruence $\frac{p-1}{2}\rho \cdot g = ef - 1 \bmod m$ holds. Note that g is even since $\frac{p-1}{2}\rho$ is odd and $ef-1$ is even. The desired prime q can then be found by testing $q = km + g + 1$ for primality, for example, using the popular Miller-Rabin primality test (e.g., see [25, page 379]). Here the range of k should be chosen so that there are enough possibilities for the desired prime q to be found in the interval $2^{l_q-1} < km + g + 1 < 2^{l_q}$. For primes of interest to us (256 to 512 bits in size), a reasonable choice is $l_k = 16$. If no prime is found with k in this range, we can start with a different f. We can easily see that $t = \frac{l_n}{2} - l_e - l_k$.

For security we have to make sure that the sparseness of f should not give any advantage in finding the secret d over other known methods. We first note that $\frac{n+1}{2} = \lambda(n) + \frac{p+q}{2}$. Let $b = \frac{n+1}{2} \bmod m$ and $a = \frac{p+q}{2m}$. Then $\frac{p+q}{2}$ can be expressed as $\frac{p+q}{2} = am + b - ef + 1$, since $\frac{n+1}{2} \bmod m = (ef - 1 + (\frac{p+q}{2} \bmod m)) \bmod m$. Thus, to factor n, we only need to find a t-bit number f of weight w_f and an l_k-bit number a. These can be found by solving the congruence equation, with arbitrary base x,

$$x^{\frac{n+1}{2}-b-1} = (x^m)^a \cdot (x^{-e})^f \bmod m.$$

Using the birthday paradox, we can find the unknowns, a and f, in about $N \log_2 N$ steps, where $N = 2^{l_k/2}\binom{t-1}{w_f/2}$ (for details on this type of attack, see [17, 34]). Therefore, the binary weight of f, w_f, should be large enough for this attack to require more workload than other general-purpose factoring methods (e.g., taking $w_f = 30$ is sufficient even for a 512-bit modulus).

For a large e, it is important for security to choose a large ρ. Since $ed = 1 + \rho\lambda(n) = 1 + \frac{n-(p+q)+1}{2}\rho$, we can approximate the secret d by $d \sim \frac{\rho}{2e}(n-(p+q))$ if e is large (e.g., on the order of p) and ρ is chosen very small. Therefore, we have to choose ρ such that

$|\rho| > |e| - \frac{|n|}{2} + c$ for complexity of 2^c steps in this type of attack. This restriction should be true for all other schemes described in this section.

We cannot see whether there is other more efficient attack on this key generating scheme. Since p is a random prime and q is determined by this p and another sparse random number f, the resulting modulus n would appear as difficult to factor with general-purpose factoring algorithms as a general product of two random primes.

As discussed in Section 2, it will be preferable to use a DR modulus to simplify the modular reduction process. Let h be the number of the leading bits of n that we want to make all ones. The required form of modulus may be obtained by choosing both p and q in this form. However, a better approach would be to take the first h bits of p and q as α and β, respectively, such that $\alpha\beta = 2^{2h} - r$ where $r < 2^h$.[3] For this, an h-bit number β is first chosen as $\beta = e\gamma$ with random γ and then α is computed by $\alpha = \frac{2^{2h}}{\beta}$. We then find a prime p whose first h bits have a value α. The other prime q can be found by testing $q = \beta 2^{l_q - h} + km + g + 1$ for primality as before, where the parameter t should now be taken as $t = l_q - h - l_e - l_k$ during the computation of m, f and g. It is easy to see that f appears in the last t bits of d as before, since $\beta 2^{l_q - h} = 0 \bmod m$.

3.2 Making the First t Bits Sparse

With a different technique, we may generate a secret key d whose high order bits are sparse in binary weight. To satisfy the equation $ed = 1 + \frac{1}{2}(p-1)(q-1)\rho$, we will choose p and q such that $\frac{p-1}{2}\rho = -1 \bmod e$ and $q = 2 \bmod e$. We first generate a prime p of length $\frac{l_n}{2}$ as before, with additional constraint $\frac{p-1}{2}\rho = -1 \bmod e$ for a random $\rho < e$. We next find a prime q such that $q = 2 \bmod e$ and that a sparse t-bit number f appears in the first t bits of d. Let l_k be the bit-length of a margin for primality tests (e.g., $l_k = 16$ in most cases). we first compute g by $g = g' - (g' \bmod e) + 2$, where $g' = \frac{ef2^{l_p + l_k + l_\rho}}{\rho p}$, and add e to make g odd if g is even. We then test $q = g + 2ke$, for $0 \le k < 2^{l_k}$, until a prime q is found. It is easy to see that t is given by $t = \frac{l_n}{2} - l_e - l_k$ as before and that f appears in the first t bits of the secret d.

Unlike the previous method, there seems no restriction on the choice of f in this method, since the higher half of $\lambda(n)$ can always be assumed to be known (note that $n = 2\lambda(n) + p + q - 1$). We may choose f as $f = 2^t$, which then reduces the storage for the secret key in addition. In this case, however, the modulus n is completely determined by the first chosen prime p. This seems to raise no security problem as long as p is chosen at random. But it would be desirable to involve some randomness also in the generation of q.

It is also simple to construct a DR modulus with the method. All that is required is to make the first h bits of ef all ones. Let $\alpha = \frac{2^{h+l_e}}{e}$. This α is placed in the first h bits of f. Then ef will have the desired form, since $e\alpha = 2^{h+l_e} - r$ for some $r < e$. In this case, of course, the number of bits that can be predetermined is reduced to $\frac{l_n}{2} - l_e - l_k - h$, as in the previous method.

[3] To guarantee that the first h bits of n are actually all ones rather than all zeros, we need to have a few zeros follow α and β in p and q, respectively, so that the carry digit to $\alpha\beta$ from the lower computation is less than r. Of course, we may instead use the positive form, $n = 2^{l_n} + n'$ with $n' < 2^{l_n - h}$, just by replacing addition with subtraction in the DR reduction algorithm.

3.3 A Hybrid Scheme

The two methods described before can be combined to produce a secret key with some high order and low order bits sparse in binary weight. Let f be an s-bit sparse number and g be a t-bit sparse number. We would like f and g to appear in the first s bits and the last t bits of d, respectively.

Let $m = e2^t$ as in Section 3.1. We will generate two primes p and q such that $p = f_1 2^t + g_1$ and $q = f_2 2^t + g_2 + km$ for some small k, where $f_1, f_2 < 2^{l_n/2-t}$, $g_1 < 2^t$ and $g_2 < m$. The first prime p is found with random f_1 and g_1, where $\frac{g_1-1}{2}$ must be relatively prime to m (see below). To simplify the computation for finding q, we had better choose p such that $p = 2 \bmod e$.

Considering the desired form of d, we next generate the other prime q as follows. To make g appear in the last t bits of d, we have to choose a q such that $\frac{1}{2}(p-1)(q-1)\rho = eg - 1 \bmod m$. This can be satisfied by choosing f_2, g_2 such that $f_1(g_2 - 1) + f_2(p-1) = 0 \bmod e$ and $\frac{1}{2}(g_1 - 1)(g_2 - 1)\rho = eg - 1 \bmod m$. From the latter congruence, we can determine g_2 as $g_2 = (\frac{g_1-1}{2}\rho)^{-1}(eg - 1) + 1 \bmod m$. To make f appear in the first s bits of d, we first compute f_2' as $f_2' = \frac{ef2^{l_n/2-t+l_k+l_\rho}}{\rho f_1}$. From this we can see that $s = \frac{l_n}{2} - t - l_e - l_k$, since $|f_1| = |f_2| = \frac{l_n}{2} - t$. This f_2' is then modified to f_2 so that $(p-1)f_2 + f_1(g_2 - 1) = 0 \bmod e$ holds. We thus obtain the final value of f_2 as $f_2 = f_2' - (f_2' + f_1(g_2 - 1) \bmod e)$, where we assume that p was chosen such that $p = 2 \bmod e$. With these f_2 and g_2, we can find the prime q by testing $q = f_2 2^t + g_2 + km$, for $0 \le k < 2^{l_k}$, for primality.

Though the last $l_e + l_k$ bits of f_2 are changed by the addition of km, this does not affect the first s bits of $d = \frac{1+\rho\lambda(n)}{e}$, since we have allowed an l_k-bit margin when determining f_2. Thus the total number of bits that can be predetermined, the sum $s + t$, is given by $\frac{l_n}{2} - l_e - l_k$, as in the previous two methods. The modulus n can be made to be of diminished radix form much the same way as before. Its security aspects can also be deduced from the previous methods. For example, it would not be desirable to take t large. The choice of $s \simeq t$ seems reasonable. Then we can choose f and g as arbitrary sparse numbers, including $f = 2^s$ and $g = 1$.

4 Generating Short and Sparse Secret Keys

The number of multiplications required for decryption could be reduced more dramatically if we could make d much smaller than n. To this end, we may generate two primes p and q such that $p = 2rp_1 + 1$ and $q = 2rq_1 + 1$, where r is prime and p_1 and q_1 are relatively prime integers. Then $\lambda(n)$ will be given by $\lambda(n) = 2rp_1q_1$, which in turn will reduce the size of the secret key by l_r bits. Girault [15] has proposed an identity-based variant of Schnorr's identification scheme using this type of modulus, where r is used as an auxiliary prime modulus for arithmetics on exponents. In this section we examine how large this prime r can be chosen without compromising security. We then consider the problem of making the resulting secret key sparse.

We first note that $\frac{n-1}{2r^2} = 2p_1q_1 + \frac{p_1+q_1}{r}$. This equation gives us a 'minimal' requirement on the size of r. The two terms in the right-hand side must be overlapped at least in u bits for 2^u steps of complexity against the exhaustive search attack. This shows that the size of r should be at least larger than $\frac{1}{2}(\frac{l_n}{2} - u)$, where $u = l_{p_1} - l_r$ (e.g., for $u = 80$ we

get $l_r \leq 88, 152, 216$ when $l_n = 512, 768, 1024$, respectively). This seems to be the basis on the choice of security parameters in Girault's scheme. However, we describe a more efficient attack which can factor this form of modulus in about $2^{u/2}$ steps. For this attack, We will assume that r is known. This assumption is obviously ture for a 512-bit modulus. But it is also reasonable for larger moduli, since finding a relatively small r by factoring $n-1$ (e.g., using the elliptic curve method [20]) is much easier than factoring n itself.

From the identity $\frac{n-1}{2r} = 2rp_1q_1 + p_1 + q_1$, we know the residue $b = p_1 + q_1 \bmod r$. Thus we can write the sum $p_1 + q_1$ as $p_1 + q_1 = ar + b$, where a is u bits long and is the only unknown that we need to find. We now have $\frac{n-1}{2r} = ar + b \bmod \lambda(n)$. This gives the final equation to be solved for a,

$$x^{\frac{n-1}{2r}-b} = (x^r)^a \bmod n.$$

Therefore, we can find the unknown a and thus can factor n in about $N \log_2 N$ operations (by sorting) or N operations (by hashing), where $N = 2^{u/2}$, using Shanks' baby-step giant-step algorithm (see also [32, 17]).[4] This analysis shows that the difference of the sizes in p_1 (or q_1) and r, $u = l_{p_1} - l_r$, has the same security consequence as the size of exponents in discrete log cryptosystems. For example, by taking $u = 160$ to achieve $N = 2^{80}$, we obtain, as maximal allowable sizes of r, 48, 112 and 176 bits for moduli of 512, 768 and 1024 bits, respectively. We could not find yet other precautions that should be taken for the security of this key generating method.

Now, let us consider a method for producing sparse secret keys in this case. We only describe a method for specifying some of the leading bits of d. The other methods can be easily derived from the previous sections. We start with the equation $ed = \rho\lambda(n) + 1 = 2rp_1q_1\rho + 1$. To satisfy this equation, we first generate a random prime r of given size and then choose p_1, q_1 and $\rho < e$ such that $2rp_1\rho = 1 \bmod e$ and $q_1 = -1 \bmod e$. Then p is found by testing $2rp_1 + 1$ with p_1 increasing by $2e$ at a time, where p_1 is initially taken odd.

Let f be a sparse number which we want to appear in the first t bits of d. We first compute $g = \frac{ef2^{l_n/2+l_k+l_\rho}}{\rho p}$ and set $q_1 = g - (g \bmod e) - 1$. We then test $q = 2r(q_1 + ek) + 1$, for $0 \leq k < 2^{l_k}$, until q is found to be prime. Since $q_1 + ek$ must be relatively prime to p_1, it would be better to choose p_1, when generating p, so that it does not have small prime factors. We can easily see that $t = |q_1| - l_e - l_k$ and that this size of f can appear without modification in the high order bits of d.

To generate a DR modulus with this method, we have to ensure that the first h bits of the product of r and ef should be all ones. This can be done by taking the first h bits of f, f_h, as $f_h = \frac{2^{2h+l_e}}{er_h}$, where r_h denotes the first h bits of r.

5 Sparse Secret Keys for CRT Application

The Chinese remainder theorem allows us to compute $y = x^d \bmod n$ by evaluating two half-size exponentiations, $y_p = x^{d_p} \bmod p$ and $y_q = x^{d_q} \bmod q$, where $d_p = d \bmod p - 1$ and $d_q = d \bmod q - 1$. This increases the performance of the RSA system roughly by a factor of up to 4. In this section we consider a method for obtaining sparse values of d_p and d_q.

[4] Pollard's rho method (or lambda method for this case) [36] requires negligible memory, compared to Shanks' method requiring space of about N elements. However, Pollard's methods require knowledge of the order of the underlying group and thus cannot be used for the present case.

Since each of d_p and d_q can be generated independently, we only consider the case of d_p. The key parameters, e, d_p and p, must satisfy the equation $ed_p = 1 \bmod p-1$. Thus we can generate a prime p such that $\rho(p-1) = -1 \bmod e$ with a random $\rho < e$, from which d_p can be obtained by $d_p = \frac{1+\rho(p-1)}{e}$. Now, to generate a sparse key d_p, we first choose a random integer f of length $\frac{l_n}{2} - l_e + l_\rho$ and of desired weight w_f. Let $g = \frac{f}{\rho}e + 1 - (\rho^{-1} \bmod e)$. Add e to g if g is even. The desired prime p can then be found by testing $p = g + 2ke$, for $k = 0, 1, \cdots$, for primality. The same procedure can be used for generating d_q and q.

In this method, we would like all three moduli, p, q and n, to be of diminished radix form, so that the DR reduction algorithm can be used for both encryption and decryption. For this it suffices for both p and q to have all ones in the leading h bits. Let $\alpha = \frac{2^{h+l_e+l_\rho}}{e\rho}$. To generate a DR modulus p, we perform the Miller-Rabin test for the number $p = e\alpha 2^{l_p-h-l_e} + g + 2ke$, for $k = 0, 1, \cdots$, where g is computed as above with f of length $l_p - h - l_e + l_\rho$. Then it can be easily seen that the leading h bits of p are all ones. We can generate q in the same way.

An attractive feature of this method from the previous ones is that most bits of the secret key can be specified during the key generation process. To guarantee security, we have to allow large enough possibilities for primes p and q, so that the exhaustive search attack is infeasible. We can find the prime p by computing $\gcd(x^{p'-1} - 1 \bmod n, n)$ for each guess p' at p. This search attack requires $\binom{l_p-h-l_e-1}{w_f-1}$ steps. Using the fast Fourier transform [1, Chapter 8], we can reduce this complexity to $N(\log_2 N)^2$ steps with $N = \binom{l_p-h-l_e-1}{w_f/2}$ (see also [4, Section 5.2]). In either case, we can easily achieve a required complexity with a relatively small value of w_f (e.g., $N > 2^{80}$ with $w_f = 30$ and $l_p = 384$).

On the other hand, the sparseness of d_p implies that the prime p is also sparse. A product of two sparse primes may reveal some knowledge on the bits of the primes. For example, the low order and high order bits of n may give some advantage in guessing the corresponding bits of p and q. This is because these bits result from the sum of a few shifted versions of a sparse p (or q). In this respect, it would be better to choose some of the low order and high order bits of f at random. This not only makes it difficult to guess the corresponding bits of p and q, but also makes the resulting modulus appear more random (not sparse). We do not see how the sparseness of prime factors can be exploited in general to factor the modulus.

Finally, we note that this method of generating RSA secret keys is independent of the number of prime factors of n. For larger RSA moduli, we may use more than two primes. For example, we may construct a 1024-bit modulus using three primes of around 341 bits rather than two 512-bit primes to further speed up the decryption process. With the current state of knowledge on factoring methods, it seems safe to use this size of integers with three prime factors as RSA moduli. In this case, all three secret keys can be made sparse as above. This will substantially increase the performance compared to the two prime case. However, it is easy to see that with the methods of Section 3 we can control only $\frac{l_n}{c} - l_e - l_k$ bits of d for moduli having c equal prime factors (see also [44]).

6 Further Considerations on Security

In this section we consider the security of our key generating methods from the viewpoint of other specialized attacks on the RSA system.

There are many special-purpose factoring algorithms. Pollard's $p-1$ method [35] and

Williams' $p+1$ method [46] are particularly efficient for finding prime factors p for which $p-1$ and $p+1$ respectively have only relatively small prime factors. Lenstra's elliptic curve method (ecm) [20], which can be viewed as a generalization of the $p-1$ method, is successful in finding small prime factors, say up to 40 decimal digits. It is therefore often recommended that RSA moduli should be constructed with strong primes of almost equal size [16]. However, we note that the ecm succeeds with probability almost equal to the probability that a random integer close to p (not exactly $p-1$ or $p+1$) is completely factored into only small primes. This observation shows that it seems to make little sense to require that $p-1$ and $p+1$ should each have at least one large prime factor (see also [26, 28]).

Nevertheless, if desired, it is possible to make both $p-1$ and $q-1$ to have a prime factor of arbitrary size, at the cost of decrease in the number of bits that can be predetermined. Suppose that we want $q-1$ to have a prime factor r. In the method of Section 3.1, we can find a desired prime q by computing g as $g = (\frac{(p-1)r\rho}{2})^{-1}(ef-1) \bmod m$ and then performing the Miller-Rabin primality test for the number $q = r(g+km)+1$. In the method of Section 3.2, we choose r such that $r = 2 \bmod e$, compute g as $g = g' - (g' \bmod e) + 1$, where $g' = \frac{ef2^{l_p+l_r+l_k+l_\rho}}{rp\rho}$, and then test $q = r(g+ke)+1$ for primality. These two techniques can be applied to other methods as well. In these cases, of course, the parameter t should be chosen as $t = \frac{l_n}{2} - l_r - l_e - l_k$.

If we are patient with longer computation time, a better approach would be to find a prime q such that $\frac{q-1}{k}$ is also prime, where k is a product of small prime factors of $q-1$ found by trial division. This method does not decrease the number of bits that we can control. For this we find a prime q with any of the methods described before and test $\frac{q-1}{k}$ for primality. These two steps must be repeated until the desired prime is found. This will lead to a substantial increase in the key generation time compared to the original method. However, implementation results (for 768 bit moduli) show that this method of finding a more desirable prime is quite within the bound of practicality (see the next section).

There is another reason why we may prefer the above form of prime for RSA moduli. One way of deciphering a ciphertext in the RSA system is to repeatedly encrypt the ciphertext until the original ciphertext comes out [42] (see also [37]). This can be seen from the observation that there always exists an integer $k < \lambda(n)$ for some b such that $b^{e^k} = b \bmod n$.[5] We can easily show that the number of b's satisfying this recurrence relation is equal to $(1+\gcd(e^k-1, p-1))(1+\gcd(e^k-1, q-1))$. Thus, if $p-1$ and $q-1$ each have a very large prime factor, the fraction of these k-th order fixed points will be negligible for any k (see also [26] for further discussions on this topic).

The special number field sieve [22] applies to numbers of the special form $n = r^c \pm s$ for small integers r and s. In any modulus generated by our methods, however, the size of s is unlikely to be smaller than $\frac{l_n}{2}$ (e.g., even if we take f as $f = 2^t$ in the method of Section 3.2). Thus the possibility that this factoring algorithm can factor our moduli seems not much better than that for ordinary moduli.

[5] If it holds true for any b, this then leads to factorization of n. In this case, however, there is no need to use the exponent e in the recurrence relation. It suffices to find any exponent which has small order modulo $\lambda(n)$ (in fact, either modulo $p-1$ or modulo $q-1$). We do not pursue this issue further, since the possibility that a particular choice of e has small order (say, less than 2^{60}) modulo $p-1$ is negligible even for a random prime p (see [37]).

There exists a method for finding the prime factors p, q of the modulus n when a certain number of bits of p and/or q is known. In this case, the problem is reduced to that of finding bounded integer solutions x and y, when given $c_i (0 \leq i \leq 3)$, such that $c_0 = c_1 x + c_2 y + c_3 xy$. Rivest and Shamir [40] first considered this problem and suggested a way of factoring n when the high (or low) order $\frac{l_n}{3}$ bits of q is given. Recently, Coppersmith [8] extended their work and developed a method for finding p, q when the the high order bits of q known is less than or equal to $\frac{1}{4} l_n$. This method can factor a modulus generated by some methods in [43]. However, our key generation methods never reveal such a large number of bits.

A final remark is concerning the use of small secret exponents. We used the relation $ed = 1 + \rho\lambda(n)$ to compute the secret key d, where $\lambda(n) = \frac{1}{2}(p-1)(q-1)$ for the methods of Section 3 and $\lambda(n) = 2rp_1q_1$ for the method of Section 4. Thus the bit-length of d is always given by $l_{\lambda(n)} - l_e + l_\rho$. There may be a situation where it is desirable to reduce the time for signature generation at the cost of longer verification time (e.g., signature generation by a weak power smart card and verification by a powerful computer). For this we may generate a small d by taking e large and ρ small. In this case, we have to note Wiener's attack [45] on short RSA secret keys. By Wiener's attack we can find a secret exponent d in polynomial time as far as the inequality $\rho dg < \frac{2pq}{3(p+q)}$ holds, where $\rho = \frac{ed-1}{\lambda(n)}$ and $g = \gcd(p-1, q-1)$. Therefore, to prevent this attack, we have to choose the bit-length of d at least larger than $\frac{l_n}{2}$ in the methods of Section 3 and larger than $\frac{l_n}{2} - l_r \simeq l_{p_1}$ in the method of Section 4. It is not known whether there is a variant of this attack when d_p and d_q are short.

7 Implementation Results

We have implemented our key generating methods in Sun Sparc 20 workstation using our experimental crypto library. For primality test, we first applied trial division by the first 1000 primes and then iterated the Miller-Rabin primality test ten times (5 iterations with small bases and 5 iterations with random bases). Table 1 shows the key generation times of various methods for 768-bit moduli, which were obtained by averaging over 100 moduli. Two cases were considered. In the first case (Case 1), we generated RSA key parameters using the basic schemes described in Sections 3 to 5. In the second case (Case 2), two primes p and q were produced with further requirements such that $p-1$ and $q-1$ each should have a large prime factor and that the modulus n should be of diminished radix form with 32 leading ones. For comparison, we included the time for generating a product of two random primes (denoted 'random' in Table 1).

In the second case, p and q were generated as follows. When p can be chosen at random as in the methods of Sections 3 and 4, we first generated a prime p_1 of length $l_p - 16$ and then tested $p = kp_1 + 1$ $(2^{15} < k < 2^{16})$ until a p is found to be prime. To guarantee a large prime factor of $q-1$, we generated q as in the first case and eliminated small prime factors of $q-1$ by trial division with the first 1000 primes. We then tested the remaining number by iterating the Miller-Rabin primality test ten times as before. These steps were repeated until a desired prime q were found. In the method of Section 4, the same steps were done for the number $\frac{q-1}{r}$, where r was taken as a 112-bit prime.

The table shows that in Case 1 our key generation methods consume almost the same CPU time as that for generating random RSA moduli. However, the key generation times in Case 2 show that finding the second prime q (both primes in the method of Sec. 5) requires

Methods	random	Sec. 3.1	Sec. 3.2	Sec. 3.3	Sec. 4	Sec. 5
Case 1	34.1	44.0	39.8	41.1	47.8	39.9
Case 2	64.3	401.7	368.4	386.9	215.9	746.0

Table 1: Average key generation times for 768-bit moduli (in seconds)

about six times as much time as generating random primes of the same type. This seems better than expected, considering that in Case 2 we have to generate Case 1 primes until the desired form of prime is found. This will be mainly due to the elimination of small prime factors of $q-1$ before applying the Miller-Rabin primality test. We simply chosen the first 1000 primes for trial division, without trying to decide their optimal number.

During the above experiments, we took the public key as $e = 257$ and also counted the average weight of the corresponding secret keys (only for Case 1). The results are summarized in Table 2 below. The last row shows the number of multiplications required for RSA decryption using the binary method (which equals $l_d + w_d - 2$). These data were obtained as follows. The binary weight of the predetermined bits was set to 30 in all our methods, though it can be chosen arbitrarily small in the methods of Sections 3.2, 3.3 and 4. In the method of generating secret keys for CRT, the first and last 16 bits of f were randomly chosen and thus 30 non-zero bits were randomly distributed in the other bits. In this case, the weight of d denotes the sum of weights in d_p and d_q. The same is true for the length of d. We counted one multiplication mod p as $\frac{1}{4}$ multiplication mod n. The last row of the table well explains the level of improvements that can be obtained with the secret keys by our key generation methods.

Methods	random	Sec. 3.1	Sec. 3.2	Sec. 3.3	Sec. 4	Sec. 5
Weight of d	380.9	230.2	227.6	227.4	228.1	89.9
Length of d	764.8	759	759	759	648	754
No of Mul.	1143.7	987.2	984.6	984.4	874.1	210.0

Table 2: Performance comparison for RSA decryption ($e = 257, \rho = 1$)

8 Conclusion

We have presented various methods for generating RSA key parameters, $n = pq, e$ and d (or d_p and d_q), in which some of the high order and/or low order bits of the secret key d can be predetermined. By taking these predetermined bits as a very sparse random number, we can considerably reduce the number of multiplications required for RSA decryption. The most attractive case is to use the Chinese remainder theorem with sparse secret keys d_p and d_q, in which case most of the bits in d_p and d_q (except for a few tens of bits) can be

predetermined. In the case that prime factors of n are unknown, the best alternative is to use a short and sparse secret key. The use of such secret keys will result in a considerable speed-up in both hardware and software implementations of RSA. We have considered, to the best of our knowledge, various security issues on these restricted secret keys. We would like to encourage the reader to further investigate their security.

References

[1] A.V.Aho, J.E.Hopcroft and J.D.Ullman, *The design and analysis of computer algorithms*, Addison-Wesley, Reading, Mass., 1974.

[2] S.Arno and F.S.Wheeler, Signed digit representation of minimal Hamming weight, *IEEE Trans. Computers*, 42(8), 1993, pp.1007-1010.

[3] P.D.Barret, Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor, In *Advances in Cryptology-Crypto'86*, LNCS 263, Springer-Verlag, 1987, pp.311-323.

[4] P.Béguin and J.J.Quisquater, Fast server-aided RSA signatures secure against active attacks, In *Advances in Cryptology-Crypto'95*, LNCS 963, Springer-Verlag, 1995, pp.57-69.

[5] J.Bos and M.Coster, Addition chain heuristics, In *Advances in Cryptology-Crypto'89*, LNCS 435, Springer-Verlag, 1990, pp.400-407.

[6] A.Bosselaers, R.Govaerts and J.Vandewalle, Comparison of three modular reduction functions, In *Advances in Cryptology-Crypto'93*, LNCS 773, Springer-Verlag, 1994, pp.175-186.

[7] E.F.Brickell, D.M.Gordon, K.S.McCurley and D.Wilson, Fast exponentiation with precomputation, In *Advances in Cryptology-Eurocrypt'92*, LNCS 658, Springer-Verlag, 1993, pp.200-207.

[8] D.Coppersmith, Finding a small root of a bivariate integer equation: factoring with high bits known, In *Advances in Cryptology-Eurocrypt'96*, LNCS 1070, Springer-Verlag, 1996, pp.178-189.

[9] D.Coppersmith, Low-exponent RSA with related messages, In *Advances in Cryptology-Eurocrypt'96*, LNCS 1070, Springer-Verlag, 1996, pp.1-9.

[10] D.Coppersmith, Finding a small root of a univariate modular equation, In *Advances in Cryptology-Eurocrypt'96*, LNCS 1070, Springer-Verlag, 1996, pp.155-165.

[11] W.Diffie and M.E.Hellman, New directions in cryptography, *IEEE Trans. Info. Theory*, 22(6), 1976, pp.644-654.

[12] P.Downey, B.Leony and R.Sethi, Computing sequences with addition chains, *Siam J. Computing*, 3, 1981, pp.638-696.

[13] S.R.Dussé and B.S.Kaliski, A cryptographic library for the Motorola DSP56000, In *Advances in Cryptology-Eurocrypt'90*, LNCS 473, Springer-Verlag, 1991, pp.230-244.

[14] T.ElGamal, A public key cryptosystem and a signature scheme based on discrete logarithms, *IEEE Trans. Info. Theory*, IT-31, 1985, pp.469-472.

[15] M.Girault, An identity-based identification scheme based on discrete logarithms modulo a composite number, In *Advances in Cryptology-Eurocrypt'90*, LNCS 473, Springer-Verlag, 1991, pp.481-486.

[16] J.Gordon, Strong RSA keys, *Electronics Letters*, 20(12), 1984, pp.514-516.

[17] R.Heiman, A note on discrete logarithms with special structure, In *Advances in Cryptology-Eurocrypt'92*, LNCS 658, Springer-Verlag, 1993, pp.454-457.

[18] J.Jedwab and C.J.Mitchell, Minimum weight modified signed-digit representations and fast exponentiation, *Electronics Letters*, 25(17), 1989, pp.1171-1172.

[19] C.K.Koç, Analysis of sliding window techniques for exponentiation, *Computers Math. Applic.*, 30(10), 1995, pp.17-24.

[20] H.W.Lenstra, JR, Factoring integers with elliptic curves, *Ann. Math.*, 126, 1987, pp.649-673.

[21] A.K.Lenstra and H.W.Lenstra,Jr., *The development of the Number Field Sieve*, Lecture Notes in Math. 1554, Springer, 1993.

[22] A.K.Lenstra, H.W.Lenstra, Jr., M.S.Manasse and J.M.Pollard, The number field sieve, In *Proc. 22nd ACM Symp. on Theory of Computing*, 1990, pp.564-572.

[23] C.H.Lim and P.J.Lee, Use of RSA moduli with prespecified bits, *Electronic Letters*, 31(10), 1995, pp.785-786.

[24] C.H.Lim and P.J.Lee, More flexible exponentiation with precomputation, In *Advances in Cryptology-Crypto'94*, LNCS 839, Springer Verlag, 1994, pp.95-107.

[25] D.E.Knuth, *The art of Computer Programming, Vol.2, Seminumerical Algorithms*, 2nd Edition, Addison-Wesley, Reading, Mass., 1981.

[26] U.M.Maurer, Fast generation of prime numbers and secure public-key cryptographic parameters, *J. Cryptology*, 8(3), 1995, pp.123-156.

[27] G.Meister, On an implementation of the Mohan-Adiga algorithm, In *Advances in Cryptology-Eurocrypt'90*, LNCS 473, Springer Verlag, 1991, pp.496-500.

[28] P.Mihailescu, Fast generation of provable primes using search in arithmetic progressions, In *Advances in Cryptology-Crypto'94*, LNCS 839, Springer Verlag, 1994, pp.282-293.

[29] S.B.Mohan and B.S.Adiga, Fast algorithms for implementing RSA public key cryptosystem, *Electronics Letters*, 21(7), 1985, p.761.

[30] P.L.Montgomery, Modular multiplication without trial division, *Math. Comp.*, 44, 1985, pp.519-521.

[31] A.M.Odlyzko, The future of integer factorization, *CryptoBytes*, 1(2), 1995, pp.5-12.

[32] P.C.van Oorschot and M.J.Wiener, Parallel collision search with application to hash functions and discrete logarithm, *Proc. of 2nd ACM Conference on Computer and Communications Security*, Fairfax, Virginia, Nov. 1994, pp.210-218.

[33] G.Orton, L.Peppard and S.Tavares, A design of a fast pipelined modular multiplier based on a diminished-radix algorithm, *J. Cryptology*, 6(4), 1993, pp.183-208.

[34] B.Pfitzmann and M.Waidner, Attacks on protocols for server-aided RSA computation, In *Advances in Cryptology-Eurocrypt'92*, LNCS 658, Springer Verlag, 1993, pp.153-162.

[35] J.M.Pollard, Theorems on factorization and primality testing, In *Proc. of the Cambridge Philosophical Society*, 76, 1974, pp.521-528.

[36] J.M.Pollard, Monte Carlo methods for index computation (mod p), *Math. Comp.*, 32(143), 1978, pp.918-924.

[37] R.Rivest, Remarks on a proposed cryptanalytic attack on the M.I.T. public key cryptosystem, *Cryptologia*, 2(1), 1978, pp.62-65.

[38] R.Rivest, A.Shamir and I.Adleman, A method for obtaining digital signatures and public key cryptosystems, *Comm. ACM*, 21(2), 1978, pp.120-126.

[39] P.de Rooij, Efficient exponentiation using precomputation and vector addition chains, In *Advances in Cryptology-Eurocrypt'94*, LNCS 950, Springer Verlag, 1995.

[40] R.Rivest and A.Shamir, Efficient factoring based on partial information, In *Advances in Cryptology-Eurocrypt'85*, LNCS 219, Springer Verlag, 1986, pp.31-34.

[41] C.P.Schnorr, Efficient signature generation by smart cards, *Journal of Cryptology*, 4(3), 1991, pp.161-174.

[42] G.Simmons and M.Norris, Preliminary comments on the M.I.T. public key cryptosystem, *Cryptologia*, 1(4), 1977, pp.406-414.

[43] S.A.Vanstone and R.J.Zuccherato, Short RSA keys and their generation, *J. Cryptology*, 8(2), 1995, pp.101-114.

[44] S.A.Vanstone and R.J.Zuccherato, Using four-prime RSA in which some of the bits are specified, *Electronics Letters*, 30(25), 1995, pp.2118-2119.

[45] M.J.Wiener, Cryptanalysis of short RSA secret exponents, *IEEE Trans. Inform. Theory*, IT-36, 1990, pp.553-558.

[46] H.C.Williams, A $p+1$ method of factoring, *Math. Comp.*, 39(159), 1982, pp.225-234.

[47] C.N.Zhang, An improved binary algorithm for RSA, *Computers and Math. Applic.*, 25(6), 1993, pp.15-24.