Investigations on Fast Exponentiation Algorithms for RSA Cryptographic Applications

Chia-Long Wu	Der-Chyuan Lou	Te-Jen Chang
Director of Aviation & Communication Electronics Department, Chinese Air Force Institute of Technology	Director of Computer Center, Chung Cheng Institute of Technology, National Defense University	Graduated Department of National Defense Science, Chung Cheng Institute of Technology, National Defense University,
chialongwu@seed.net.tw	dclou@ccit.edu.tw	karl@ccit.edu.tw

Abstract

Exponentiation is to compute X^E for a positive integer E and modular exponentiation is to compute X^E mod M for positive integers E and M. When the lengths of the operators are at least 1024 binary representations or 300 decimal digits, modular exponentiation can be time-consuming and is often the dominant part of the computation in many algebra systems. Since exponentiation is a sequence of multiplications, there are two kinds of methods to accelerate the speed of modular exponentiation. One is to reduce the number of multiplications and the other is to accelerate the multiplication itself.

In this paper, we describe some efficient exponentiation methods, which can effectively reduce the number of multiplications and other methods, which can accelerate multiplication itself respectively. Most importantly, we also detailed analyze the computational complexity for two kinds of these methods respectively.

Keywords:Public-key cryptosystem, cryptography, variable length nonzero window, modular multiplication, addition chain.

1. Introduction

The modular exponentiation is a common operation for most cryptosystems. Most of cryptographic systems based on modular exponentiation. Generally, modular exponentiation is represented using a chain of modular multiplications. The performance of such cryptosystems is primarily determined by the implementation efficiency of the multiplication and the exponentiation. There are two primary ways to reduce the time on the computation of modular exponentiation with large operators. One is to decrease the time to perform basic modular multiplication [1-4] and the other is to reduce the number of modular multiplications used to compute X^E [5-8].

In the rest of this paper, we will present and compare two kinds of methods. Some methods which reduce the number of multiplications are presented in Section 2. In Section 3, we present other methods which accelerate the multiplication itself. In Section 4, we will use tables for computational complexity analyses. Finally, some concise conclusions and future works are given in Section 5.

2. Methods for Reducing the Number of Multiplications

2.1 Right-to-Left Binary Method

The right-to-left binary algorithm starts at the least significant bit and works upward. This algorithm requires an extra data register S to store the middle variable. Note that modular multiplication and square in this right-to-left binary algorithm are independent of one another, and thus two operations at each loop can be parallelized. Provide that one multiplier and one squarer available, the running time of the right-to-left binary algorithm is bounded by the total time required of computing k modular squares. The right-to-left binary method is described in Algorithm 1.

Algorithm 1 (Right–to–Left Binary Method)

```
Input: X, E
Output: X^E contained in C
S = X
C = 1
for i=1 to k
  {
    if (i^th binary bit of E is 1)
      then  C=C*S                /* multiply */
    S=S*S                        /* square */
  }
```

More descriptions of right-to-left binary method are depicted in [9].

2.2 Exponent-Folding Method

Let the exponent E be iteratively folded in half n times i.e. E is divided into 2^n equal sized substrings. Let each substring of E be denoted as E_i for i=1, 2, …,

2^n, i.e. $E= E_{2^n} \| E_{2^n-1} \| ... \| E_1$, where "||" is the concatenation operator and k is the bit length of E. Hence

$$X^E = \prod_{i=1}^{2^n} sq^{((i-1)(\frac{k}{2^n}))}(X^{E_i}) \tag{1}$$

where $sq^{(m)}(Z)$ represents performing m squares on the related value Z. Using Horner's rule [10], Equation (1) can be transformed as shown in Equation (2).

$$X^E = sq^{(\frac{k}{2^n})}(\ldots sq^{(\frac{k}{2^n})}((sq^{(\frac{k}{2^n})}(X^{E_{2^n}})) * X^{E_{2^n-1}})\ldots * X^{E_2}) * X^{E_1} \tag{2}$$

We present the variables: E_{com_j} , $E_{com\_(j+1)}$, E_j, $E_{(j+1)}$, E_{excl_i} , E_{com_i} , and E_i in Equation (3), (4), and (5).

$$E_{com\_j} = E_{com\_(j+1)} = E_j \text{ AND } E_{(j+1)} \quad \text{for } j=1, 3, \ldots, 2^n-3, 2^n-1 \tag{3}$$

$$E_{excl\_i} = E_{com\_i} \text{ XOR } E_i \quad \text{for } j=1, 2, \ldots, 2^n \tag{4}$$

Each E_i can be represented as shown in Equation (5).

$$E_i = E_{com\_i} + E_{excl\_i} \tag{5}$$

The exponentiation of the consecutive pairs of $X^{E_{2^n}}$, $X^{E_{2^n-1}}$, X^{E_1} can be computed as shown in Equation (6) and Equation (7).

$$X^{E_j} = X^{E_{com\_j}} * X^{E_{excl\_j}} \tag{6}$$

$$X^{E_{j+1}} = X^{E_{com\_j}} * X^{E_{excl\_(j+1)}} \quad \text{for } j=1, 3, \ldots, 2^n-3, 2^n-1 \tag{7}$$

Let E_y have the binary representation

$e_y^{\frac{k}{2^n}} * e_y^{\frac{k}{2^n}-1} * \ldots * e_y^1$.

Thus, an efficient algorithm for computing X^{E_j} and $X^{E_{j+1}}$ is depicted as Algorithm 2. The result of X^{E_j} and $X^{E_{j+1}}$ are kept in C_1 and C_2 respectively. Based on Equation (2) and Algorithm 2, the average number of multiplications $F(M)$ required in exponent-folding method is shown in Equation (8). Let M denote the required number of multiplications.

$$F(M)= 2^{n-1}(M * \frac{3k}{2^{n+2}} + 1 * \frac{k}{2^{n+2}} + 2) + (k - \frac{k}{2^n}) + (2^n - 1) \tag{8}$$

The exponent-folding method is described in Algorithm 2.

Algorithm 2 (Exponent-Folding Method)

C_1=C_2=C_3=1
S=X
for b=1 to $\frac{k}{2^n}$ do /* scan from LSB to MSB */
{
  if ($e^b_{excl\_j} = 1$) then C_1 =S*C_1 /* multiply */
  if ($e^b_{excl\_(j+1)} = 1$) then C_2 =S*C_2 /* multiply */
  if ($e^b_{com\_j} = 1$) then C_3 =S*C_3 /* multiply */
  S=S*S /* square */
}
C_1= C_1*C_3
C_2= C_2*C_3

More descriptions of exponent-folding method are depicted in [10].

2.3 Exponent-*t*-Folding Exponent Method

When we compute X^E, let the exponent E= $e_k e_{k-1} e_{k-2} \ldots e_1$, where $e_i \in \{0, 1\}$ (i=1, 2, …, k), be divided into t equal-length bit substrings. If k(mod t)≠0, then E is padded with t-k(mod t) zeros to the left. Each bit substring of E is denoted as E_i (1≤i≤t), i.e. E=E_t||E_{t-1}||…||E_1, where "||" is concatenation operation among E_t, E_{t-1}, …, E_1. The corresponding generalization mini-terms E_{com_j} (j=1, 2, …, 2^t) have the binary representations $e_{com\_j}^{\lceil \frac{k}{t} \rceil} * e_{com\_j}^{\lceil \frac{k}{t} \rceil - 1} \ldots e_{com\_j}^1$. The Exponent-*t*-Folding method can be implemented as follows.

Step 1. Derive all the generalization mini-terms except the generalization mini-term $E_{com\_2^t}$ = $\text{AND}_{i=1}^{t}(NOT\ E_i)$ from the bit substrings E_t, E_{t-1}, …, E_1.

Step 2. Employ the extended right-to-left binary algorithm to compute the exponentiation values $X^{E_{com\_1}}$, $X^{E_{com\_2}}$, …, $X^{E_{com\_(2^t-1)}}$. The extended right-to-left binary algorithm is shown in Algorithm 3.

Step 3. X^{E_1} , X^{E_2} , …, X^{E_t} can be constructed in Equation (9).

Step 4. X^E can be evaluated in Equation (10).

$$X^{E_i} = X^{\sum_{j=1,\, E_{com\_j} AND\ E_i \neq 0_{bs}}^{2^t} E_{com\_j}} \quad \text{for } i=1, 2, \ldots, t. \tag{9}$$

$$X^E = \prod_{i=1}^{t} sq^{(i-1)*\lceil \frac{k}{t} \rceil}(X^{E_i}) = sq^{\lceil \frac{k}{t} \rceil}(\ldots sq^{\lceil \frac{k}{t} \rceil}((sq^{\lceil \frac{k}{t} \rceil}(X^{E_t})) * X^{E_{t-1}})\ldots X^{E_2}) * X^{E_1} \tag{10}$$

Algorithm 3 (Extended Right-to-Left Binary Method)

Input: X, E_{com_1}, E_{com_2}, …, $E_{com\_(2^t-1)}$

Output: $X^{E_{com\_1}}$, $X^{E_{com\_2}}$, …, $X^{E_{com\_(2^t-1)}}$ contained in C_1, C_2, …, C_{2^t-1}

S=X;
C_1=1, C_2=1, …, C_{2^t-1} =1
for m=1 to $\lceil \frac{k}{t} \rceil$ do /* scan from LSB to MSB */
{
 if ($e^m_{com\_1} = 1$) then C_1=S*C_1
 if ($e^m_{com\_2} = 1$) then C_2=S*C_2
  M /* multiply */

if ($e^{m}_{com\_(2^t-1)} = 1$) then $C_{2^t-1} = S * C_{2^t-1}$

S=S*S /* square */

}

More descriptions of Exponent-t-Folding method are depicted in [11].

2.4 Variable Length Nonzero Window Method

The variable length nonzero window (VLNW) partitioning strategy requires that during the formation of a nonzero window (NW), we switch to ZW when the remaining bits are all zero. The VLNW partitioning strategy has two integer parameters:

d: maximum nonzero window length,

q: minimum number of zeros required to switch to ZW.

This VLNW method proceeds as follows.

ZW: Check the incoming single bit: if it is zero then stay in ZW; else stay in NW.

NW: Checking the incoming q bits: if they are all zero then go to ZW; else stay in NW. Let d=1+kq+r where 1<r≤q. Stay in NW until 1+kq bits are received. At the last step, the number of incoming bits will be equal to r. If there r bits are all zero, then go to ZW; else stay in NW. After all d bits are collected, check the incoming single bit: if it is zero, then go to ZW; else go to NW.

The VLNW partitioning produces nonzero windows which start with a 1 and end with a 1.

Two nonzero windows may be adjacent. However, the one in the least significant position will necessarily have d bits. Two zero windows will not be adjacent since they will be concatenated. For example, let d=5 and q=2, then 5=1+1*2+2, thus k=1 and r=2. The following example in binary representation illustrates the partitioning of a long exponent according to the above parameters d, q, k, r:

$(\underline{101}\ 0\ \underline{11101}\ 00\ \underline{101}\ \underline{10111}\ 000000\ \underline{1}\ 00\ \underline{111}\ 000\ \underline{1011})_2$.

Also, let d=10 and q=4, which implies k=2 and r=1. Another partitioning example is illustrated below:

$(\underline{1011011}\ 0000\ \underline{11}\ 0000\ \underline{1111110101}\ 00\ \underline{11110111}\ 0000\ \underline{11011})_2$.

More descriptions of variable length nonzero window are depicted in [3].

3. Methods for Accelerating the Multiplication Itself

3.1 *M*-ary Method

The computation of X^E for a positive integer E is required in many important applications in computer science and engineering. Let E=($E_kE_{k-1}E_{k-2}\ldots E_2E_1$) be the binary expansion of the exponent E, where n is the number of bits in the binary expansion of E. This representation of E is partitioned into n words of length d, such that nd=k. The exponent is padded with at most d-1 zeros, if d does not divide k. We define

$$F_i = (E_{id+d-1}E_{id+d-2}\ldots E_{id}) = \sum_{j=0}^{d-1} E_{id+j}2^j \qquad (11)$$

such that 0≤F_i≤2^d-1 and $E = \sum_{i=1}^{n} F_i 2^{id}$. The m-ary method first computes the values of X^W for W=2, 3, …, 2^d-1. The exponent E is then scanned d bits at a time from the most significant to the least significant. At each step, the partial result is raised to the 2^d power and multiplied with X^{F_i} where F_i is the current nonzero word. The m-ary method is described in Algorithm 4.

Algorithm 4 (The *m*-ary Method)

Input: X, E

Output: y=X^E

Compute and store X^w for all w=2, 3, 4, …2^d-1.

Decompose E into d-bit words F_i for i=1, 2, …, n.

$y = X^{F_{k-1}}$

for i=n-1 downto 1

{

$y = y^{2^d}$

if F_i≠0 then y=y* X^{F_i}

}

return y

It requires 2^d-2 preprocessing multiplications and the number of multiplication operations is equal to (n-1)d=k-d in Algorithm 4. We perform a multiplication if F_i≠0. Since 2^d-1 out of 2^d values of F_i are nonzero, the average number of multiplications required is (n-1)(1-2^{-d}) in Algorithm 4. Thus, we find the average number of multiplications as Equation (12).

$$T(k,d) = 2^d - 2 + k - d + (\frac{k}{d} - 1)(1 - 2^{-d}) \qquad (12)$$

The average number of multiplications for the binary method can be found simply by substituting d=1 in Equation (12), which gives T=1.5(k-1). Also note that there exists an optimal value for each n such that T(k, d) is minimized. The optimal values of d can be found by enumeration [12, 13].

More descriptions of m-ary method are depicted in [3].

3.2 Addition Chain Method

Computing the shortest addition is an NP-complete problem [14], but we see Knuth's method [12] for an excellent introduction to addition chains. Therefore we can find near optimal ones.

An addition chain for the binary representation of positive integer r is a list of positive integers

a_1=1, a_2, …, a_l=r,

such that, for each i>1, there is some j and k with 1≤j≤k<i and a_i= a_j+ a_k. A short addition chain for r gives a fast algorithm for computing g^r: compute $g^{a_2}, g^{a_3}, \ldots, g^{a_{l-1}}, g^r$.

Let l(r) be the length of the shortest addition chain for r. The exact value of l(r) is known only for relatively small values of r. When r is large, l(r) is shown in Equation (13).

$$l(r) = \log r + \frac{\log r}{\log\log r} + O(\frac{\log r}{\log\log r}) \qquad (13)$$

The lower bound was shown by Erdos' method [15] using a counting argument. The upper bound is just the binary algorithm [12].

For example, the standard (binary) addition chain [12] for the number 15 has length 6:

1 2 3 6 7 14 15.

There is, however, a chain of length 5 that produces 15:

1 2 3 6 12 15.

This means that one can compute X^{15} from x in 5 multiplications.

Naturally we are interested in addition chain with as small a length as possible. Knuth's method [12] is capable of producing an addition chain for a 512-bit number of length 605 on average. This is an improvement of 21% over the binary algorithm (which has length 768 on average) and an improvement of 5% over Knuth's 5-window algorithm.

More descriptions of addition chain method are depicted in [16-18].

4. Computational Complexity

In this section, we will present the computational complexity performance comparisons of many methods described as above and some other related methods shown in recent researches [19-26]. We distinguish two kinds of methods to compute the computational complexity. One is to reduce the number of multiplications and the other is to accelerate the multiplication itself.

For the first situation, as we know, the squaring operations can be regarded as a case of multiplication operations. For clarity, the modular reductions and the processes of using lookup tables can be omitted. So we make a table for comparisons of different methods as shown in Table 1, where k is the bit length of the exponent and r is the radix. In order to measure the speed of the modular multiplication, modular exponentiation, etc., we use the numbers of modular multiplications to express the speed-up efficiency [12, 27, 30-34].

For the second situation, we use the area and the time to show the differences between methods [19-20, 23-26, 35-39] as shown in Table 2 and Table 3. Sometimes we use interpolation and extrapolation methods to estimate the result for 1024-bit size as shown in Table 4. In Table 2, 3, and 4, the unit of area is the numbers of 2-input NAND. The area for one Logic gate of 2-input NAND is 2.73×10^{-6} mm^2.

Table 1. Comparisons for computational complexities of modular multiplications (*k*:exponent, *r*: radix).

Methods	The number of Multiplications
Lou-Wu [33]	$\frac{r^3+3r^2-2r+1}{r^2(r+1)} * k$
Lou-Wu [27]	0.689*k*+11
Lou-Wu-Chen [31]	(29*k*/36)+3
Avizienis [34]	1.292*k*
Yen [32]	1.292*k*
Yen and Laih [30]	1.375*k*+3
D. E. Knuth [12]	1.5*k*

Table 2. Comparisons of the area and the time for 16, 32, 64, and 128 bits.

Author	Area	Time	Size
Wang-Lin [35]	172	0.00142ns	16bits
Lee-Yoo [25]	367	0.00145ns	16bits
Srikanthan-Lam-Suman [20]	107.67	0.99ns	16bits
Wang-Lin [35]	240	0.00286ns	32bits
Lee-Yoo [25]	586	0.00289ns	32bits
Srikanthan-Lam-Suman [20]	244.16	1.1ns	32bits
Wang-Lin [35]	220	0.00574ns	64bits
Lee-Yoo[25]	735	0.00577ns	64bits
Srikanthan-Lam-Suman [20]	516.62	1.25ns	64bits
Yeh-Reed-Truong[36]	1260	1.8ns	64bits
Srikanthan-Lam-Suman [20]	1061.61	1.4ns	128bits
Nedjah-Mourelle [24]	3179	3.3ns	128bits
Yeh-Reed-Truong[36]	2991	2.5ns	128bits
Nedjah-Macedo [23]	259	23ns	128bits

Table 3. Comparisons of the area and the time for 256, 512, and 768 bits.

Author	Area	Time	Size
Srikanthan-Lam-Suman [20]	2011.13	1.59ns	256bits
Nedjah-Mourelle [24]	4004	6.6ns	256bits
Yeh-Reed-Truong [36]	4074	8.9ns	256bits
Blum-Paar[37]	1180	19.7ns	256bits
Nedjah-Macedo [23]	304	42ns	256bits
Nedjah-Mourelle [24]	5122	7.1ns	512bits
Blum-Paar [37]	2217	19.5ns	512bits
Nedjah-Macedo [23]	492	76ns	512bits
Nedjah-Mourelle [24]	6278	8.3ns	768bits
Blum-Paar [37]	3275	20ns	768bits
Nedjah-Macedo [23]	578	82ns	768bits

Table 4. Comparisons of the area and the time for 1024 bits.

Author	Area	Time
Wang-Lin [35]	3520	0.0922ns
Srikanthan-Lam-Suman [20]	7708.25	2.73ns
Nedjah-Mourelle [24]	7739	8.9ns
Blum-Paar [37]	4292	18ns
Yeh-Reed-Truong [36]	10572	47.3ns
Yile-Xingjun [19]	8050	114ns
Nedjah-Macedo [23]	639	134ns
Yang-Wu-Zhou [38]	9100	160ns
Kwon-You-Heo [39]	46000	325ns

5. Conclusions and Future Works

An efficient computation of the modular exponentiation is very important and useful public-key cryptosystems. We know many researchers are devoted to reducing the number of multiplications and improving the hardware design in computer algorithms for information management and network security usages.

Now there are still many novel methods issued in many computer security journals [11, 26-28] and reports for computer arithmetic operations and theoretical analyses. In the future, we will incorporate modular arithmetic and some novel techniques (including hardware and software design) to effectively perform overall RSA evaluation (the number of multiplications or accelerate the multiplication itself respectively) for modern cryptographic applications.

References

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

[2] P. Downey, B. Leong, and R. Sethi, "Computing sequences with addition chains," *SIAM Journal of Computing*, Vol. 10, No. 3, pp. 638-646, 1981.

[3] C. K. Koç, "Analysis of sliding window techniques for exponentiation," *Computers & Mathematics with Applications*, Vol. 30, No. 10, pp. 17-24, Nov. 1995.

[4] Y. Yacobi, "Exponentiating faster with addition chains," *Eurocrypt'90, Lecture Notes in Computer Science 473*, Springer Verlag, 1991, pp. 222-229.

[5] S. M. Hong, S. Y. Oh, and H. S. Yoon, "New modular multiplication algorithms for fast modular exponentiation," *In Advances in Cryptology-EUROCRYPT'96, LNCS 1070*, Springer-Verlag, 1996, pp. 166-177.

[6] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, Vol. 44, No. 170, pp. 519-521, April 1985.

[7] H. Morita, "A fast modular-multiplication algorithm based on a higher radix," *Proceeding of CRYPTO in Advances in Cryptology*, 1990, pp. 387-399.

[8] C. D. Walter, "Faster modular multiplication by operand scaling," *Advances in Cryptology-CRYPTO '91*, *Lecture Notes in Computer Science*, Springer-Verlag, Vol. 576, pp. 313-323, 1992.

[9] M. Joye and S.-M. Yen, "Optimal left-to-right binary signed-digit recoding," *IEEE Transactions on Computers*, Vol. 49, No. 7, pp. 740-748, July 2000.

[10] D.-C. Lou and C.-C. Chang, "Fast exponentiation method obtained by folding the exponent in half," *IEE Electronics Letters*, Vol. 32, No. 11, pp. 984-985, May 1996.

[11] D.-Z. Sun, Z.-F. Cao, and Y. Sun, "How to compute modular exponentiation with large operators based on the right-to-left binary algorithm," *Applied Mathematics and Computation*, Vol. 176, No. 1, pp. 280-292, May 2006.

[12] D. E. Knuth, *Seminumerical Algorithms, 2nd Edition*, *The Art of Computer Programming*, Vol. 2, Addison-Wesley, Reading MA, 1981.

[13] C. K. Koc, "High-radix and bit recoding techniques for modular exponentiation," *International Journal of Computer Mathematics*, Vol. 40, No. 7, pp. 139-156, 1991.

[14] P. Downey, B. Leong, and R. Sethi, "Computing sequences with addition chains," *SIAM Journal of Computing*, Vol. 10, No. 3, pp. 638-646, 1981.

[15] P. Erdos, "On addition chains," *Acta Arithmetic*, Remarks on Number Theory, Vol. III, pp. 77-81, 1960.

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

[17] D. M. Gordon, "A survey of fast exponentiation methods," *Journal of Algorithms*, Vol. 27, No. 1, pp. 129-146, April 1998.

[18] N. Nedjah and L. M. Mourelle, "Efficient and secure cryptographic systems based on addition chains: Hardware design vs. software/hardware co-design," to be appeared in Integration the VLSI Journal.

[19] S. Yile and W. Xingjun, "An area efficient modular arithmetic processor," *Proceedings of 5th International Conference on ASIC*, Vol. 2, Oct. 2003, pp. 1273-1276.

[20] T. Srikanthan, S. K. Lam, and M. Suman, "Area-time efficient sign detection technique for binary signed-digit number system," *IEEE Transactions on Computers*, Vol. 53, No. 1, pp. 69-72, Jan. 2004.

[21] M. E. Kaihara and N. Takagi, "A hardware

algorithm for modular multiplication/division," *IEEE Transactions on Computers*, Vol. 54, No. 1, pp. 12-21, Jan. 2005.

[22] N. Nedjah and L. M. Mourelle, "A review of modular multiplication methods and respective hardware implementations," *Informatica*, Vol. 30, No. 1, pp. 111-129, 2006.

[23] N. Nedjah and L. M. Macedo, "Reconfigurable hardware implementation of Montgomery modular multiplication and parallel binary exponentiation," *Proceedings of the Euromicro Symposium on Digital System Design*, pp. 226-233, Sept. 2002.

[24] N. Nedjah and L. M. Mourelle, "Fast reconfigurable systolic hardware for modular multiplication and exponentiation," *Journal of Systems Architecture*, Vol. 49, No. 7-9, pp. 387-396, Oct. 2003.

[25] W.-H. Lee, K.-J. Lee, and K.-Y. Yoo, "Design of a linear systolic array for computing modular multiplication and squaring in GF(2^m)," *Computers and Mathematics with Applications*, Vol. 42, No. 1, pp. 231-240, July 2001.

[26] N. Nedjah and L. M. Mourelle, "Reconfigurable hardware for addition chains based modular exponentiation," *International Conference on Information Technology: Coding and Computing*, pp. 603-607, Vol. 1, Apr. 2005.

[27] D.-C. Lou and C.-L. Wu, "Parallel exponentiation using common-multiplicand-multiplication and signed-digit-folding techniques," *International Journal of Computer Mathematics*, Vol. 81, No. 10, pp. 1187-1202, Oct. 2004.

[28] C.-L. Wu, D.-C. Lou, and T.-J. Chang, "Computational complexity analyses of modular arithmetic for RSA cryptosystem," *Proceedings of the 23rd Workshop on Combinatorial Mathematics and Computation Theory*, Section C2: Security and Applications, Apr. 2006, pp. 215-224.

[29] Y. Yacobi, "Exponentiating faster with addition chains," *Eurocrypt'90, Lecture Notes in Computer Science 473*, Springer Verlag, 1991, pp. 222-229.

[30] S.-M. Yen and C.-S. Laih, "Common-multiplicand multiplication and its applications to public key cryptography," *IEE Electronics Letters*, Vol. 29, No. 17, pp. 1583-1584, Aug. 1993.

[31] D.-C. Lou, C.-L. Wu, C.-Y. Chen, "Fast exponentiation by folding the signed-digit exponent in half," *Journal Title: International Journal of Computer Mathematics*, Vol. 80, No. 10, pp. 1251-1259, Oct. 2003.

[32] S.-M. Yen, "Improved common-multiplicand multiplication and fast exponentiation by exponent decomposition," *IEICE Transaction Fundamentals*, Vol. E80-A, No. 6, pp. 1160-1163, June 1997.

[33] D.-C. Lou and C.-L. Wu, "Parallel modular exponentiation using signed-digit-folding technique," *Informatica* Vol. 28, No. 2, pp. 197-205, July 2004.

[34] A. Avizienis, "Signed digit number representation for fast parallel arithmetic," *IRE Transaction on Electronic Computers*, Vol. EC-10, No. 3, pp. 389-400, Sep. 1961.

[35] C. L. Wang and J. L. Lin, "Systolic array implementation of multipliers for finite fields GF(2^m), *IEEE Transaction Circuits Systems*, Vol. 38, pp. 796-800, July 1991.

[36] C. S. Yeh, I. S. Reed, and T. K. Truong, "Systolic multipliers for finite fields GF(2^m), *IEEE Transaction Computer*, Vol. C-33, pp. 357-360, Dec. 1984.

[37] T. Blum and C. Paar, "Montgomery modular exponentiation on reconfigurable hardware," *Proceedings. 14th IEEE Symposium on Computer Arithmetic*, Vol. 14 No. 16, April 1999, pp. 70-77.

[38] Q. Yang, X. Wu, R. Zhou, and et al, "An embedded RSA processor for encryption and decryption," *Proceedings of 4th International Conference On ASIC*, 2001, pp. 356-359.

[39] T.-W. Kwon, C.-S. You, W.-S. Heo, and et al, "Two implementation methods of a 1024-bit RSA cryptoprocessor based on modified Montgomery algorithm", *IEEE International Symposium on Circuit and Systems*, pp. 650, 2001.