공학박사학위논문

공통 피승수 곱셈을 응용한 효율적인 일괄 모듈러 누승 기법

Fast batch modular exponentiation
with common-multiplicand multiplication

2018년 2월

서울대학교 대학원

전기 · 컴퓨터공학부

서 정 주

Abstract

Fast batch modular exponentiation with common-multiplicand multiplication

Jungjoo Seo
Department of Computer Science and Engineering
The Graduate School
Seoul National University

In the field of asymmetric cryptography, the exponentiation operation that raises an element from a group to its power is fundamental for a great part of cryptosystems. The exponents are often large numbers which essentially lead to significant resource consumption for computation. There are two approaches to enhance the performance of exponentiation. One way is to improve the efficiency of multiplication itself. The other way is to reduce the number of multiplications that is required to perform exponentiation.

In this thesis, we present an efficient algorithm for batch modular exponentiation which improves upon the previous generalized intersection method with respect to the cost of multiplications. The improvement is achieved by adopting an extended common-multiplicand multiplication technique that efficiently computes an arbitrary number of multiplications that share a common multiplicand at once. Our algorithm shows a better time-memory tradeoff compared to the previous generalized intersection method.

We analyze the cost of multiplications and storage requirement of the proposed algorithm, and give an optimal parameter choice when the allowed amount of memory is sufficient or limited. The comparison shows that our algorithm

reduces the cost of multiplications by 23% $\sim$ 41% when the number of input exponents is between 10 and 60, and the bit length of exponents is 1024, 2048 and 4096. The practical performance enhancement is also supported by the actual running time comparison. For further improvement, we also present an exponent ordering algorithm that rearranges the input exponents and a precomputation method to reduce the cost of multiplications.

Keywords: Cryptography, Exponentiation, Modular Exponentiation, Public-Key Cryptography, Common-Multiplicand Multiplication, Algorithm

Student Number: 2009-20820

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Batch cryptography is an active research area for generating and verifying multiple exponentiations and signatures, or performing cryptographic primitives such as encryption and decryption simultaneously. Especially in a cryptographic application that requires digital signatures of multiple messages simultaneously, multiple exponentiations with a fixed base and various exponents need to be computed at the same time. We present an efficient algorithm for batch modular exponentiation with a fixed base and various exponents based on the generalized intersection method with exponent grouping. The enhancement is achieved by common-multiplicand multiplication due to the shared repeated squaring process and an expoenent rearragement method. The introduction begins with the background on batch exponentiation followed by the outline of the thesis.

1.1 Background

Exponentiation, that raises an element from a group to its power, is one of the most important arithmetic operations in public-key cryptography. The RSA cryptosystem [43] requires exponentiation in $\mathbb{Z}/n\mathbb{Z}$ for a positive integer $n = pq$

where p and q are large primes. Meanwhile, Diffie-Hellman key agreement [13] and ElGamal scheme [19] require exponentiation in $\mathbb{Z}_p$ for a large prime p.

There are two types of circumstance in which exponentiation algorithms are applied. Firstly, the exponent x is fixed and the base g is chosen arbitrarily as in RSA scheme. Secondly, the base g is fixed and the exponent x is chosen arbitrarily. ElGamal scheme and Diffie-Hellman key agreement protocol benefit from algorithms that computes such exponentiation efficiently. We mainly focus on the latter situation where the base g is fixed.

The efficiency of exponentiation is increasingly important because the required key size has grown to ensure the cryptographic security levels. It is also important because exponentiations are performed in devices with weak computing power as the internet of things becomes ubiquitous. The computation cost for exponentiation is heavily depends on the performance of multiplication, since exponentiation consists of a series of multiplications. There are two approaches to enhance the performance of exponentiation. One way is to improve the efficiency of multiplication itself [2, 11, 26, 37]. The other way is to reduce the number of multiplications that is required to perform exponentiation [24]. If possible, both approaches can be adopted.

One approach to improve the efficiency of the multiplication is to take advantage of a special form of multiple multiplications $\{A_i \times C \mid i = 1, 2, ..., t\}$ that share an identical multiplicand C [25, 46, 47, 49, 50]. Yen and Laih [50] proposed an algorithm for common-multiplicand multiplications (CMM for short) to improve the performance of the square-and-multiply exponentiation algorithm. Rather than handling multiple multiplications independently, the algorithm processes the bitwise common part of the multipliers and decomposes a multiplier by the common part. This method reduces the number of 1's in the decomposed multipliers which results in less number of additions for computing the multiplications. Wu and Chang [47] improved Yen and Laih's algorithm by folding the multipliers k times. Yen [49] proposed another improved algorithm

for the case that the number of multiplications is three.

In an application that requires a modular multiplication such as $x \times y \bmod N$, Montgomery algorithm [37] can be adopted rather than using a straightforward multiplication method [29] due to the efficiency. In Montgomery multiplication, the given integers x and y are transformed into Montgomery form where the modulo operation can be performed by the cheap shift operation rather than the expensive division operation. Ha and Moon [25] introduced an efficient algorithm for Montgomery multiplication where two multiplications share a common multiplicand. Wu [46] extended Ha and Moon's algorithm to handle more than two multiplications.

To compute an exponentiation with a smaller number of multiplications, majority of research focused on finding a short addition chain for a given exponent x. As the problem of finding the shortest addition chain is NP-complete [17], practical algorithms for exponentiation without a division operation have been studied such as the binary method, the m-ary method, window methods [31, 34, 41, 52], exponent-folding methods [33, 49], and precomputation methods [7, 32]. For a group where the inversion operation is efficient like elliptic curve cryptosystems [30, 35], an addition-subtraction chain can yield a good performance [38].

Batch cryptography is an active research area for generating and verifying multiple exponentiations and signatures, or performing cryptographic primitives such as encryption and decryption simultaneously. Batch verification, which is firstly introduced by Naccache et al. [40], is used to verify multiple signatures simultaneously consuming less time than verifying total individual signatures. Successive research results have achieved improvements in terms of efficiency and security [1, 3, 4, 5, 8, 9, 21, 27, 51]. There also have been researches for batch signature schemes [22, 42, 53] and batch generation of exponentiations [10, 39].

In a cryptographic application that requires digital signatures of multi-

ple messages simultaneously, multiple exponentiations with various exponents $\{x_1, x_2, ..., x_n\}$ and a fixed base g need to be computed at the same time [28,44]. As a solution to the batch exponentiation problem, the generalized intersection method was proposed by M'Raïhi and Nacacche [39]. This method takes advantage of the fact that intersecting exponents leads to a less number of multiplications. They also showed that the exponentially increasing number of partitioned exponents, which results in the exponentially increasing number of multiplications, can be reduced by dividing exponents into groups. By grouping exponents, the squaring process is shared amongst exponent groups, and the algorithm achieves various time-memory tradeoffs. Chung et al. [10] proposed a decremental combination strategy that removes the overlapping multiplications in the combination stage of the generalized intersection method.

1.2 Contribution

We propose a k-way batch exponentiation algorithm that improves the evaluation stage of the generalized intersection method with exponent-grouping. Our algorithm divides exponents into several groups and produces many common-multiplicand multiplications in the evaluation stage. This approach significantly enhances the performance of the evaluation stage and the whole procedure achieves a better time-memory tradeoff compared to Chung et al.'s by adopting CMM multiplication. An optimal choice for the exponent group size for both cases when the allowed amount of memory is sufficient or limited is given with performance analysis. The k-way algorithm reduces the cost of multiplications by $23 \sim 41\%$ and running time by $24 \sim 48\%$ when the number of input exponents is $10 \sim 60$. We also present an exponent ordering algorithm for rearranging input exponents and a precomputation method to reduce the cost of multiplications even further.

1.3 Organization

The rest of the thesis is organized as follows. In Chapter 2, we review the background on common-multiplicand multiplication, exponentiation and batch exponentiation algorithms. In Chapter 3, we propose the k-way batch exponentiation algorithm and analyze the time and space performance. In Chapter 4 we show the experimental results and compare the performance with previous algorithms. In Chapter 5, exponent ordering and precomputation are considered for further improvement. Finally, we conclude in Chapter 6

Chapter 2

Related Work

There are two approaches to enhance the performance of exponentiation. One way is to improve the efficiency of multiplication itself. The other way is to reduce the number of multiplications that is required to perform exponentiation. For the former approach, we introduce common-multiplicand multiplication algorithms that enhances the performance of multiplications that share a common-multiplicand. For the latter, a generalized intersection method for batch exponentiation, that handles multiple exponentiations with a fixed base simultaneously, and its improvement by Chung et al. are described.

2.1 Common-Multiplicand Multiplication

Common-multiplicand multiplication (CMM for short) is a widely used method for enhancing the performance of exponentiations [25, 46, 47, 49, 50]. The CMM method reduces the time for performing multiplications, and enhances the performance of exponentiation. In this thesis, we are particullary interested in the methods based on Montgomery multiplication since it efficiently evalutes modular multiplication. Ha and Moon [25] extended Montgomery multiplication to

MontMult(x, y, b, n, m'_0, m)

1 $a \leftarrow 0$

2 **for** $i \leftarrow 0$ **to** $n-1$

3 $\quad a \leftarrow a + x_i y$

4 $\quad u \leftarrow a_0 n'_0 \bmod b$

5 $\quad a \leftarrow (a + um)/b$

6 **if** $a \geq m$

7 $\quad a \leftarrow a - m$

8 **return** a

Figure 2.1: Algorithm for Montgomery multiplication

handle two multiplications with a common-multiplicand efficiently, and Wu [46] generalized Ha and Moon's work for an arbitrary number of multiplications.

2.1.1 Montgomery Multiplication

Montgomery multiplication [37] is a method for efficient modular multiplication and exponentiation. Consider following numbers x, y and m in radix b representation with n digits :

$$x = \sum_{i=0}^{n-1} x_i b^i, y = \sum_{i=0}^{n-1} y_i b^i, m = \sum_{i=0}^{n-1} m_i b^i$$

x_i, y_i and m_i are elements from of $\{0, 1, ..., b-1\}$. Let R be an integer such that $R = b^n > m$ and $\gcd(R, m) = 1$, and R^{-1} be the inverse of R modulo m. The m-residue with respect to R of an integer $T < m$ is defined by $\overline{T} = TR \bmod m$. Montgomery reduction takes an integer a such that $0 \leq a < Rm$ and yields $aR^{-1} \bmod m$. To compute the Montgomery reduction of the product of two integers x and y, multi-precision multiplication and Montgomery reduction method can be combined efficiently [18]. Montgomery multiplication MontMult(x, y) yields $xyR^{-1} \bmod m$ as described in Algorithm MontMult of

MontMultCMM(x, y, z, b, n, m'_0, m)

1 $a \leftarrow 0$
2 $b \leftarrow 0$
3 $t \leftarrow z$
4 **for** $i \leftarrow n-3$ **downto** 0
5 $\quad u \leftarrow t_0 m'_0 \bmod b$
6 $\quad t \leftarrow (t + um)/b$
7 $\quad a \leftarrow (a + tx_i)/b \qquad b \leftarrow (b + ty_i)/b$
8 **for** $i \leftarrow 0$ **to** 1
9 $\quad a \leftarrow a + x_{n-2+i} z \qquad b \leftarrow b + y_{n-2+i} z$
10 $\quad u_a \leftarrow a_0 m'_0 \bmod b \qquad u_b \leftarrow b_0 m'_0 \bmod b$
11 $\quad a \leftarrow (a + u_a m)/b \qquad b \leftarrow (b + u_b m)/b$
12 **if** $a \geq m$
13 $\quad a \leftarrow a - m$
14 **if** $b \geq m$
15 $\quad b \leftarrow b - m$
16 **return** a, b

Figure 2.2: Algorithm for Common-Multiplicand Montgomery multiplication

Figure 2.1, and consumes $2n^2 + n$ single-precision multiplications. Note that $\overline{T} = \text{MontMult}(T, R^2)$ and $T = \text{MontMult}(\overline{T}, 1)$.

2.1.2 Common-Multiplicand Montgomery Multiplication

Ha and Moon [25] proposed a common-multiplicand Montgomery multiplication algorithm that takes $\{x, y, z\}$ and produces $\{xzR^{-1}, yzR^{-1}\}$ requiring $3n^2 + 3n$ single-precision multiplications. The Montgomery reduction of the product of

two integers u and v can be written as follows :

$$\begin{aligned} uvR^{-1} \bmod m &= u(v_{n-1}b^{n-1} + v_{n-2}b^{n-2} + ... + v_0b^0)b^{s-n}b^{-s} \bmod m \\ &= (v_{n-1}(ub^{s-1} \bmod m) + v_{n-2}(ub^{s-2} \bmod m)) + ... \\ &\quad + v_0(ub^{s-n} \bmod m))b^{-s} \bmod m \end{aligned}$$

where $b^{-n} = R^{-1} \bmod m$. With the choice $s = 2$, the Montgomery reductions of xz and yz are

$$\begin{aligned} xzR^{-1} \bmod m = (x_{n-1}z + (x_{n-2}z + x_{n-3}(zb^{-1} \bmod m) + ... \\ + x_0(zb^{2-n} \bmod m))b^{-1})b^{-1} \bmod m \end{aligned}$$

$$\begin{aligned} yzR^{-1} \bmod m = (y_{n-1}z + (y_{n-2}z + y_{n-3}(zb^{-1} \bmod m) + ... \\ + y_0(zb^{2-n} \bmod m))b^{-1})b^{-1} \bmod m. \end{aligned}$$

In both Montgomery reductions, $zb^{-i} \bmod m$ for $1 \leq i \leq n-2$ can be shared, and $zb^{-i} \bmod m$ can be computed from the previous result $zb^{-i+1} \bmod m$. A common-multiplicand Montgomery multiplication algorithm is described in Algorithm MontMultCMM of Figure 2.2.

The first loop that includes the computation of all $zb^{-i} \bmod m$ requires $(n-2)(3n+1)$ single-precision multiplications. Computing the final result xzR^{-1} and yzR^{-1} in the second loop requires $2(4n+2) = 8n+4$ single-precision multiplications. Thus, the common-multiplicand Montgomery multiplication algorithm requires $3n^2 + 3n + 2$ single-precision multiplications. Since we need $2(2n^2 + n) = 4n^2 + 2n$ single-precision multiplications to perform two instances of Algorithm MontMult, the speedup of common-multiplicand Montgomery multiplication algorithm is about 1.5 considering only the highest term.

2.1.3 Extended Common-Multiplicand Montgomery Multiplication

Ha and Moon's common-multiplicand Montgomery multiplication can be extended to handle more than two multiplications with a common multiplicand

ExtMontMulCMM(X, c, b, n, m'_0, m)

1 $s \leftarrow c$
2 $Y_j \leftarrow 0$ for $1 \le j \le t$
3 **for** $i \leftarrow n-3$ **downto** 0
4 $\quad u \leftarrow s_0 m'_0 \bmod b$
5 $\quad s \leftarrow (s + um)/b$
6 $\quad Y_j \leftarrow Y_j + sX_{j,i}$ for $1 \le j \le t$
7 **for** $i \leftarrow 0$ **to** 1
8 $\quad$ **for** $j \leftarrow 1$ **to** n
9 $\quad\quad Y_j \leftarrow Y_j + cX_{j,n-2-i}$ for $1 \le j \le t$
10 $\quad\quad u \leftarrow Y_{j,0} m'_0 \bmod b$
11 $\quad\quad Y_j \leftarrow (Y_j + um)/b$
12 **for** $j \leftarrow 1$ **to** n
13 $\quad$ **while** $Y_j \ge m$
14 $\quad\quad Y_j \leftarrow Y_j - m$
15 **return** Y

Figure 2.3: Algorithm for extended Common-Multiplicand Montgomery multiplication

of the form $\{\overline{A_1 C}, \overline{A_2 C}, \ \ldots\ , \overline{A_t C}\}$ [46] as described in Algorithm ExtMontMultCMM of Figure 2.3. The required number of single-precision multiplications for the extend algorithm is as follows:

$$(t+1)b^2 + (2t-1)b + 2t - 2. \tag{2.1}$$

size(p) = 1024

#CMM	# Single-Prec. Mult.	Cost($w = 32$)	Cost($w = 64$)
1	$2b^2 + b$	1.000	1.000
2	$3b^2 + 3b$	1.524	1.549
3	$4b^2 + 5b$	2.048	2.098
4	$5b^2 + 7b$	2.572	2.648
5	$6b^2 + 9b$	3.096	3.197

size(p) = 2048

#CMM	# Single-Prec. Mult.	Cost($w = 32$)	Cost($w = 64$)
1	$2b^2 + b$	1.000	1.000
2	$3b^2 + 3b$	1.512	1.524
3	$4b^2 + 5b$	2.024	2.048
4	$5b^2 + 7b$	2.536	2.572
5	$6b^2 + 9b$	3.047	3.096

size(p) = 4096

#CMM	# Single-Prec. Mult.	Cost($w = 32$)	Cost($w = 64$)
1	$2b^2 + b$	1.000	1.000
2	$3b^2 + 3b$	1.506	1.512
3	$4b^2 + 5b$	2.012	2.024
4	$5b^2 + 7b$	2.518	2.536
5	$6b^2 + 9b$	3.024	3.047

Table 2.1: Cost of extended CMM Montgomery multiplication

Table 2.1 shows the performance of the extended CMM Montgomery method with respect to the number of single-precision multiplications when size(m) varies from 1024 to 4096, and w is 32 and 64. The constant terms in the second column are omitted. Let $SP(t)$ be the required number of single-precision multiplications to evaluate t CMMs of the form $\{\overline{A_1C}, \overline{A_2C}, \ ... \ , \overline{A_tC}\}$. The cost of t CMMs is defined as

$$c_t = \frac{SP(t)}{SP(1)} = \frac{b^2 + 2b + 2}{2b^2 + b}(t-1) + 1. \tag{2.2}$$

Note that c_t is approximately $\frac{t+1}{2}$. Henceforth, the cost of multiplications means the number of single-precision multiplications that is divided by $SP(1)$ as in the definition of c_t. We also assume that w is fixed to 64 in the rest of thesis.

2.2 Batch Exponentiation Algorithms

As the internet of things becomes ubiquitous, exponentiations are performed in devices with weak computing power that needs to sign a number of messages simultaneously. The required key size also has grown to ensure the cryptographic security levels. Thus, the efficiency of such exponentiation is increasingly important. In a batch modular exponentiation, given g, p, and n l-bit exponents $X = \{x_1, x_2, ..., x_n\}$, we want to compute $R = \{R_i = g^{x_i} \bmod p \mid 1 \leq i \leq n\}$ simultaneously. As a solution to the batch exponentiation problem, M'Raïhi and Nacacche [39] proposed the generalized intersection method, and Chung et al. [10] improved M'Raïhi and Nacacche's algorithm by introducing a decremental combination strategy that removes the overlapping multiplications.

Throughout the thesis, the i-th element in an array A is denoted by A_i. Likewise, the i-th bit of an integer $a = \sum_{i=0} a[i]2^i$ is denoted by $a[i]$. For binary operations, let $\wedge$, $\vee$ and $\oplus$ denote the bitwise AND, OR and XOR, respectively. Let $d = d[n]d[n-1]...d[1]$ be an n-bit integer and $A = \{a_1, a_2, ..., a_n\}$ be an array of n integers. Given a bit b, $\bigwedge_{i:d[i]=b} a_i$ denotes the result of bitwise AND of all a_i's such that $d[i] = b$. $\bigvee_{i:d[i]=b} a_i$ is defined in the same way. Also the exponents

SquareAndMultiplyLSB(g, x)

1 $y \leftarrow 1$

2 $s \leftarrow g$

3 **for** $i \leftarrow 0$ **to** $l - 1$

4 **if** $x[i] = 1$

5 $y \leftarrow y \times s$

6 $s \leftarrow s \times s$

7 **return** y

Figure 2.4: Algorithm for LSB Square-and-Multiply

are assumed to be randomly chosen non-negative integers which means that each bit in an exponent is 1 with probability $\frac{1}{2}$.

2.2.1 Square and Multiply

The square-and-multiply method, which is also known as binary method, is a simple exponentiation algorithm that uses the binary representation of the exponent. Let l be the length of the exponent x in binary. Then the exponent $x = \sum_{i=0}^{l-1} x_i 2^i$ and $g^x = g^{x_0 2^0} g^{x_1 2^1} ... g^{x_{l-1} 2^{l-1}}$ where $x_i \in \{0, 1\}$. The square-and-multiply method repeatedly raises g to $g^{2^{l-1}}$ by squaring while scanning the bits of x from the least significant bit (LSB), and multiplies all $g^{x_i 2^i}$ with $x_i = 1$ since $g^{x_i 2^i} = 1$ when $x_i = 0$. The LSB square-and-multiply algorithm is described in Algorithm SquareAndMultiplyLSB of Figure 2.4.

Let $w(x)$ be the number of 1's in the binary representation of x. The LSB square-and-multiply algorithm requires l squarings and $w(x)$ multiplications. Assuming that the exponent x is randomly chosen and squaring is treated as multiplication, the expected number of multiplication for the LSB square-and-multiply is $l + \frac{1}{2}l = 1.5l$.

SquareAndMultiplyMSB(g, x)

1 $y \leftarrow 1$

2 **for** $i \leftarrow l-1$ **downto** 0

3 $\quad y \leftarrow y \times y$

4 $\quad$ **if** $x[i] = 1$

5 $\quad\quad y \leftarrow y \times g$

6 **return** y

Figure 2.5: Algorithm for MSB Square-and-Multiply

Alternatively, the bits of the exponent x can be scanned from the most significant bit (MSB) as described in Algorithm SquareAndMultiplyMSB of Figure 2.5. The MSB square-and-multiply method has the same computational complexity with the LSB method.

2.2.2 Parallel Square and Multiply

The parallel square-and-multiply algorithm (PSM) is a simple extension of the square-and-multiply algorithm that perform batch exponentiation. Let us consider a set of n l-bit exponents $X = \{x_1, x_2, ..., x_n\}$, and let $x_i[j]$ be the bit at position j in the exponent x_i. To compute the result $R = \{R_i \mid R_i = g^{x_i} \bmod p\}$, the repeated squaring process that raises g to $g^{2^{l-1}}$ is performed only once. The intermediate value g^{2^j} is shared and multiplied when $x_i[j] = 1$ as described in Algorithm PSM of Figure 2.6. The number of squaring is $l-1$ since the squaring at the last iteration can be omitted. Assuming that all input exponents are randomly chosen, the number of multiplications for each exponentiation R_i is $w(x_i) - 1$ where $w(x)$ is the number of 1's in x, since a multiplication with 1 as its multiplicand can be ignored. Thus, the overall computation cost is $l + n\left(\frac{l}{2} - 1\right)$ if squaring is treated as multiplication.

```
PSM(g, p, X)
1  R_i ← 1, 1 ≤ i ≤ n
2  for i ← l − 1 downto 0
3        for j ← 1 to n
4              if x_j = 1
5                    R_j ← R_j × g
6        g ← g × g mod p
7  return R
```

Figure 2.6: Algorithm for Parallel Square-and-Multiply

2.2.3 Generalized Intersection Method

Let us consider two given exponents x_1 and x_2 of length l that are randomly chosen. The expected number of 1's is $\frac{l}{2}$ for both x_1 and x_2, and $\frac{l}{4}$ for the intersection $x_c = x_1 \wedge x_2$ where $\wedge$ is the bitwise AND operation. Then we can observe that following inequality is satisfied where $w(x)$ is the number of 1's in x.

$$w(x_1 - x_c) + w(x_2 - x_c) + w(x_c) \leq w(x_1) + w(x_2).$$

To compute the results of exponentiations g^{x_1} and g^{x_2}, we compute $g^{x_1 - x_c}$, $g^{x_2 - x_c}$ and g^{x_c} first. Then g^{x_1} and g^{x_2} is derived from $g^{x_i} = g^{x_i - x_c} g^{x_c}$ for $1 \leq i \leq 2$. The overall number of multiplication is $2\left(\frac{l}{2} - 1\right) = l - 2$ for computing g^{x_1} and g^{x_2} separately, but it is $3\left(\frac{1}{4}l - 1\right) + 2 = \frac{3}{4}l - 1$ with intersection method where squaring is considered as multiplication.

M'raïhi and Naccache proposed the generalized intersection method for batch exponentiation that applies the exponent intersecting approach to handle more than 2 input exponents. In the generalized intersection method, the n input exponents $X = \{x_1, x_2, ..., x_n\}$ are partitioned to $2^n - 1$ bitwise mutually exclusive cells by intersecting the input exponents. Exponentiations for all disjoint

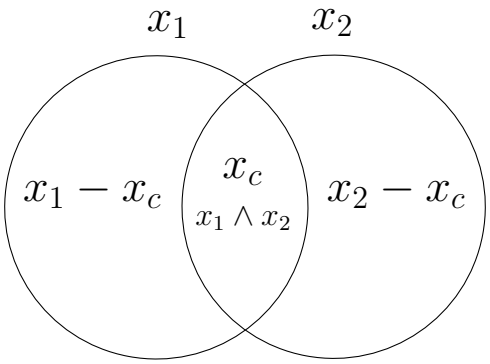

Figure 2.7: Intersection method for two exponents

cells are evaluated by Algorithm PSM and the final result $R = \{g^{x_1}, g^{x_2}, ..., g^{x_n}\}$ is derived by combining the evaluated values.

Partitioning In the generalized intersection method, the set of n input exponents $X = \{x_1, x_2, ..., x_n\}$ is partitioned to $2^n - 1$ disjoint cells by the following formula:

$$C_d = \bigwedge_{i:d[i]=1} x_i \oplus \left(\bigwedge_{i:d[i]=1} x_i \wedge \bigvee_{i:d[i]=0} x_i \right)$$

where $1 \leq d < 2^n$ and $d = d[n]d[n-1]...d[1]$ for $d[i] \in \{0, 1\}$. Note that all C_d's are bitwise mutual exclusive.

Example 1 Let us consider a set X of three exponents $x_1 = 10100111$, $x_2 = 10101111$ and $x_3 = 11100011$ with $n = 3$ and $l = 8$. The seven disjoint cells C_d for $1 \leq d < 8$ are shown in Table 2.2. C_{100} is composed of bits that are set in x_3, but not in x_1 and x_2. Likewise, bits in C_{011} are set in both x_1 and x_2, but not in x_3. As Table 2.2 shows, there is at most one cell for each bit column that is set to 1. For instance, the bit at position 5 is set to 1 only in C_{111}.

Evaluation Each $g^{C_d} \bmod p$ for $1 \leq d < 2^n$ is computed by Algorithm PSM.

Index	7 6 5 4 3 2 1 0
x_1	1 0 1 0 0 1 1 1
x_2	1 0 1 0 1 1 1 1
x_3	1 1 1 0 0 0 1 1
C_{001}	0 0 0 0 0 0 0 0
C_{010}	0 0 0 0 1 0 0 0
C_{011}	0 0 0 0 0 1 0 0
C_{100}	0 1 0 0 0 0 0 0
C_{101}	0 0 0 0 0 0 0 0
C_{110}	0 0 0 0 0 0 0 0
C_{111}	1 0 1 0 0 0 1 1

Table 2.2: Example of exponent partitioning.

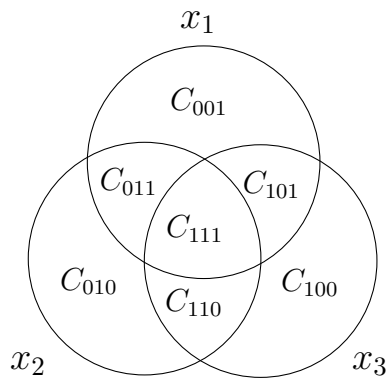

Figure 2.8: Partitioned cells for three exponents

Combination Each result value $R_i = g^{x_i} \bmod p$ for $1 \leq i \leq n$ is derived from the output values g^{C_d}'s from the evaluation stage by following formula :

$$R_i = g^{x_i} = \prod_{d:d[i]=1} g^{C_d} \bmod p.$$

The index $d = d[n-1]d[n-2]...d[1]$ of disjoint cells is represented in binary.

Let us consider the case where the number of input exponents is three for example as depicted in Figure 2.8. The following is the computation in the combination stage.

$$g^{x_1} = g^{C_{00\mathbf{1}}} g^{C_{01\mathbf{1}}} g^{C_{10\mathbf{1}}} g^{C_{11\mathbf{1}}}$$
$$g^{x_2} = g^{C_{01\mathbf{0}}} g^{C_{01\mathbf{1}}} g^{C_{11\mathbf{0}}} g^{C_{11\mathbf{1}}}$$
$$g^{x_3} = g^{C_{\mathbf{1}00}} g^{C_{\mathbf{1}01}} g^{C_{\mathbf{1}10}} g^{C_{\mathbf{1}11}}$$

Performance For each disjoint cell, the probability that the cell has 1 at a bit position is $\frac{1}{2^n}$. Thus, the number of 1's in all cells is expected to be $\frac{l}{2^n}$. Assuming that the number of 1's in each cell is at least one, the evaluation stage requires $(2^n - 1)\left(\frac{l}{2^n} - 1\right)$ multiplications and $l - 1$ squarings. The combination stage requires $n\left(2^{n-1} - 1\right)$ multiplications. Thus, the overall number of multiplications is

$$l\left(2 - \frac{1}{2^n}\right) - 2^n + n\left(2^{n-1} - 1\right), \tag{2.3}$$

assuming that squaring is treated as multiplication. Since the algorithm maintains $2^n - 1$ disjoint cells and $2^n - 1$ output values from the evaluation stage, the required amount of memory is

$$(2^n - 1)\left(l + \text{size}(p)\right) \tag{2.4}$$

in bits where size(p) is the bit length of an evaluated value $g^{C_d} \bmod p$. The space requirement in bits can be normalized to

$$\frac{(2^n - 1)\left(l + \text{size}(p)\right)}{\text{size}(p)} \tag{2.5}$$

by dividing Equation 2.4 by size(p).

PART(x)

1 $P_i \leftarrow 0$ for $0 \leq i < l$

2 **for** $d \leftarrow 1$ **to** $2^n - 1$

3 $\quad c \leftarrow \bigwedge_{i:d[i]=1} x_i \oplus \left(\bigwedge_{i:d[i]=1} x_i \wedge \bigvee_{i:d[i]=0} x_i\right)$

4 $\quad$ **for** $i \leftarrow 0$ **to** $l - 1$

5 $\quad\quad$ **if** $c[i] = 1$

6 $\quad\quad\quad P_i \leftarrow d$

7 **return** P

Figure 2.9: Algorithm for exponent partitioning

2.2.4 Chung et al.'s Algorithm

Chung et al. [10] improved M'raïhi and Naccache's algorithm by reducing the required number of bits to store the partitioned exponents and removing overlapping multiplications in the combination stage.

Position Array In the exponent partitioning stage, the values of disjoint cells can be represented as a position array P of l elements because of the bitwise mutual exclusiveness of the disjoint cells. An i-th element of P is defined to be the index of the partitioned cell that has 1 for its bit at position i. Thus, $P_i = d$ if the i-th bit of C_d is 1, and $P_i = 0$ otherwise. This representation of disjoint cells reduces the memory requirement compared to M'raïhi and Naccache's algorithm. The algorithm for the partitioning stage to compute the position array P is described in Algorithm PART. In the evaluation stage, the index d of g^{C_d} that g^{2^j} is multiplied to at the j-th iteration can be found in $d = P_j$. The algorithm for the evaluation stage with a position array is described in Algorithm EVAL.

Example 2 Let us consider the input exponents in Example 1. The disjoint

```
EVAL(P, g, p)
1  G_i ← 1 for 0 < i < 2^n
2  for i ← 0 to l − 1
3        if P_i ≠ 1
4              G_{P_i} ← G_{P_i} × g mod p
5              g ← g × g mod p
6        else
7              g ← g × g mod p
8  return G
```

Figure 2.10: Algorithm for evaluation

Index	7 6 5 4 3 2 1 0
x_1	1 0 1 0 0 1 1 1
x_2	1 0 1 0 1 1 1 1
x_3	1 1 1 0 0 0 1 1
C_{001}	0 0 0 0 0 0 0 0
C_{010}	0 0 0 0 1 0 0 0
C_{011}	0 0 0 0 0 1 0 0
C_{100}	0 1 0 0 0 0 0 0
C_{101}	0 0 0 0 0 0 0 0
C_{110}	0 0 0 0 0 0 0 0
C_{111}	1 0 1 0 0 0 1 1
P	7 4 7 0 2 3 7 7

Table 2.3: Example of position array

COMB(G, p)

1 **for** $i \leftarrow n$ **downto** 1

2 $\quad R_i \leftarrow G_{2^{i-1}}$

3 $\quad$ **for** $j \leftarrow 1$ **to** $2^{i-1} - 1$

4 $\quad\quad R_i \leftarrow R_i \times G_{2^{i-1}+j} \bmod p$

5 $\quad\quad G_j \leftarrow G_j \times G_{2^{i-1}+j} \bmod p$

6 **return** R

Figure 2.11: Algorithm for decremental combination

cells and the position array are shown in Table 2.3. For instance, since the bit at position 5 is set to 1 only in C_{111}, the value of P_5 is 7.

Decremental Combination In the combination stage of M'raïhi and Naccache's algorithm, there are overlapping multiplications amongst the computations of $R_i = g^{x_i} = \prod_{d:d[i]=1} g^{C_d} \bmod p$. To avoid redundant multiplications, Chung et al. devised a decremental combination method where the final results are calculated from R_n down to R_1 (or any arbitrary order) as in Algorithm COMB of Figure 2.11. At the iteration for computing R_i, each cell $C_{2^{i-1}+j}$ is merged with the adjacent cell C_j by multiplying $g^{C_{2^{i-1}+j}}$ to g^{C_j}. After R_i is computed, the set of merged values G_d for $1 \leq d < 2^{i-1}$ is equivalent to the output of the evaluation stage for the exponent set $\{x_1, ..., x_{i-1}\}$. The process of the decremental combination stage for the case that the number of input exponents is three is described in Table 2.4. For example, for computing g^{x_3}, R_3 is initialized to $g^{C_{100}}$. Then, $g^{C_{111}}$ is multiplied to $g^{C_{011}}$ and R_3. The same procedure is applied for $g^{C_{101}}$ and $g^{C_{110}}$. After g^{x_3} is computed, G_i for $1 \leq i \leq 3$ is equivalent to the output of the evaluation stage for the input exponents $\{x_1, x_2\}$.

Index	Exponents of Evaluated Values		
	Before R_3	Before R_2	Before R_1
001	C_{001}	$C_{001} + C_{101}$	$C_{001} + C_{011} + C_{011} + C_{111}$
010	C_{010}	$C_{010} + C_{110}$	$C_{010} + C_{110}$
011	C_{011}	$C_{011} + C_{111}$	$C_{011} + C_{111}$
100	C_{100}	C_{100}	C_{100}
101	C_{101}	C_{101}	C_{101}
110	C_{110}	C_{110}	C_{110}
111	C_{111}	C_{111}	C_{111}

Table 2.4: Example of decremental combination when $n = 3$

Performance The expected number of multiplications of the evaluation stage is the same with M'raïhi and Naccache's. The required number of multiplications for computing R_i in the combination stage is $2^{i-1} - 1$. Thus, the combination stage requires $2\sum_{i=1}^{n}\left(2^{i-1} - 1\right)$ multiplications, and the total cost of Chung et al.'s algorithm is given by

$$\left(2 - \frac{1}{2^n}\right) l + 2^n - 2n - 2. \tag{2.6}$$

The amount of required memory in bits to store P and the evaluated values are nl and $(2^n - 1)\text{size}(p)$, respectively, where size(p) is the bit length of an evaluated value G_d. Thus, the space requirement in bits is

$$nl + (2^n - 1)\text{size}(p). \tag{2.7}$$

The space requirement in bits can be normalized to

$$\frac{nl + (2^n - 1)\text{size(p)}}{\text{size}(p)} \tag{2.8}$$

by dividing the equation (2.7) by size(p).

Chapter 3

k-way Batch Exponentiation

We propose an enhanced batch exponentiation algorithm that improves efficiency of the evaluation stage of the generalized intersection method. As described in [39] and [10], the set of n input exponents can be decomposed into several groups to achieve various time-memory tradeoffs. However, we point out that there is a room for improvement in the evaluation stage by observing that all exponent groups share the same multiplicand at each iteration. To handle more than two multiplications that share the same multiplicand at once, we utilize an extended common-multiplicand multiplication method.

3.1 Exponent Grouping and Partitioning

In our batch modular exponentiation algorithm, the set of input exponents $X = \{x_1, x_2, \ ... \ , x_n\}$ is divided into $k = \lceil \frac{n}{m} \rceil$ groups $X_1, X_2, ..., X_k$, given the exponent group size m. Then, each exponent group X_i contains $m_i = \lceil \frac{n}{k} \rceil$ exponents if $1 \leq i \leq n \bmod k$, and $m_i = \lfloor \frac{n}{k} \rfloor$ exponents, otherwise (i.e., m_i is the number of exponents in X_i). (The grouping above is slightly better than the grouping where $m_i = m$ for $1 \leq i < k$ and $m_i = n \bmod m$ for $i = k$.) After

the exponents are divided into k groups, the exponents in the exponent group X_i are partitioned to $C_{i,d}$ for $1 \leq d < 2^{m_i}$ and represented as a position array P_i as in Section 2.2.3.

Example 3 *Let us consider three more exponents* $x_4 = 10010010$, $x_5 = 10110101$ *and* $x_6 = 10110000$ *with those in Example 1. By setting the number of groups* k *to 2, we have two exponent groups* $X_1 = \{x_1, x_2, x_3\}$ *and* $X_2 = \{x_4, x_5, x_6\}$. *Then the position arrays of* X_1 *and* X_2 *are* $P_1 = \{7, 7, 3, 2, 0, 7, 4, 7\}$ *and* $P_2 = \{2, 1, 2, 0, 7, 6, 0, 7\}$, *respectively.*

3.2 k-way Evaluation

For each position array P_i from the exponent grouping and partitioning stage, intermediate values $g^{C_{i,d}} \bmod p$ are calculated in this stage where $0 < d < 2^{m_i}$. These values will be combined in the combination stage. The intermediate values can be calculated by simply invoking Algorithm EVAL k times. However, the same repeated squaring process is performed in the k instances of Algorithm EVAL as described in [39] and [10]. Furthermore, we observe the following:

1. At each iteration of repeated squaring, each exponent group requires at most 1 multiplication due to the mutual exclusiveness of the disjoint cells of partitioned exponents.

2. All multiplications for exponent groups at each iteration share a common multiplicand, and the squaring for the next iteration also has the same common multiplicand.

The cost of multiplications in the evaluation stage can be reduced by exploiting these properties.

First, we initialize all $G_{i,d}$ to 1 for $1 \leq i \leq k$ and $1 \leq d < 2^{m_i}$. The value of $G_{i,d}$ will be $g^{C_{i,d}} \bmod p$ at the end of the evaluation stage. After the initialization, a repeated squaring process is started with the initial value g

j	P_1	P_2	g^{2^j}	g^{2^j} multiplied to
0	7	2	g^{2^0}	$G_{1,7}$, $G_{2,2}$, g^{2^0}
1	7	1	g^{2^1}	$G_{1,7}$, $G_{2,1}$, g^{2^1}
2	3	2	g^{2^2}	$G_{1,3}$, $G_{2,2}$, g^{2^2}
3	2	0	g^{2^3}	$G_{1,2}$, g^{2^3}
4	0	7	g^{2^4}	$G_{2,7}$, g^{2^4}
5	7	6	g^{2^5}	$G_{1,7}$, $G_{2,6}$, g^{2^5}
6	4	0	g^{2^6}	$G_{1,4}$, g^{2^6}
7	7	7	g^{2^7}	$G_{1,7}$, $G_{2,7}$

Table 3.1: Example of k-way evaluation.

which is squared at each iteration. At the j-th iteration, $g^{2^j} \bmod p$ has to be multiplied to $G_{i,P_{i,j}}$ for each exponent group X_i where $P_{i,j} \neq 0$, because $P_{i,j}$ is the index to the disjoint cell of partitioned exponents of X_i that has 1 for its bit at position j. The number of such cells for each X_i is at most 1 due to the mutual exclusiveness of disjoint cells. Also, $g^{2^j} \bmod p$ needs to be multiplied by itself for the next iteration. All required multiplications are

$$
\begin{aligned}
G_{i_1,P_{i_1,j}} &= G_{i_1,P_{i_1,j}} \times g^{2^j} \bmod p \\
G_{i_2,P_{i_2,j}} &= G_{i_2,P_{i_2,j}} \times g^{2^j} \bmod p \\
&\cdots \\
G_{i_t,P_{i_t,j}} &= G_{i_t,P_{i_t,j}} \times g^{2^j} \bmod p \\
g^{2^{j+1}} &= g^{2^j} \times g^{2^j} \bmod p
\end{aligned}
$$

where i_s for $1 \leq s \leq t$ are the indexes of exponent groups that have non-zero $P_{i_s,j}$ values. We can observe that all multiplications at each iteration can be efficiently evaluated by extended CMM Montgomery multiplication due to the common multiplicand $g^{2^j} \bmod p$. Therefore, the cost of the multiplications is c_{t+1} rather than $t+1$ which significantly enhances the performance of the evaluation stage. Note that t is at most k at each iteration.

At the iteration of $j = 5$ in Example 3, both $P_{1,5} = 7$ and $P_{2,5} = 6$ are non-zero and thus $G_{1,P_{1,5}}$ and $G_{2,P_{2,5}}$ need to be multiplied by $g^{2^5} \mod p$. This is because $C_{1,7}$ and $C_{2,6}$ have 1 for their bits at position 5. The number of multiplications for all iterations is 20. By adopting the extended common-multiplicand Montgomery multiplication method, the cost of multiplications is reduced to $4c_3 + 3c_2 = 12.764$, where $c_2 = 1.524$ and $c_3 = 2.048$. All multiplications that are computed during the whole evaluation stage are presented in Table 3.1.

3.3 Combination

The evaluated values are combined by decremental combination strategy to produce the final results $R_i = g^{x_i} \mod p$. Since the two multiplications in lines 4-5 of Algorithm COMB also share the same multiplicand, we can use CMM to reduce the multiplication cost. This modified algorithm is applied to the evaluated values of each exponent group. The whole procedure of k-way batch exponentiation algorithm is described in Figure 3.1.

3.4 Performance Analysis

3.4.1 Cost of Multiplications

The most part of the running time of the batch exponentiation algorithm is spent for computing multiplications including squares. We analyze the running time performance by counting the costs of multiplications in the evaluation and combination stages. In this section, we assume that n is divisible by m for simplicity. (When n is not divisible by m, the analysis is more complicated, but the result is almost identical.)

Lemma 3.4.1 *Given k exponent groups X_i for $1 \leq i \leq k$ with m exponents each, let p_t be the probability that there are t non-zero $P_{i,j}$ values at each j-th iteration in the evaluation stage. Then $p_t = \binom{k}{t} \left(\frac{1}{2^m}\right)^{k-t} \left(1 - \frac{1}{2^m}\right)^t$.*

BatchExponentiation(X, n, g, p, m)

1 $k \leftarrow \lceil \frac{n}{m} \rceil$, $j \leftarrow 1$
2 **for** $i \leftarrow 1$ to k
3 **if** $i \leq n \bmod k$
4 $m_i \leftarrow \lceil \frac{n}{k} \rceil$
5 **else**
6 $m_i \leftarrow \lfloor \frac{n}{k} \rfloor$
7 $P_i \leftarrow \text{Part}(\{x_j, \ \ldots \ , x_{j+m_i-1}\})$
8 $j \leftarrow j + m_i$
9 $G_{i,j} \leftarrow 1$ for $1 \leq i \leq k$ and $1 \leq j < 2^{m_i}$
10 **for** $j \leftarrow 1$ to $l - 1$
11 $M \leftarrow \{g\}$
12 **for** $i \leftarrow 1$ to k
13 **if** $P_{i,j} \neq 0$
14 M.append$(G_{i,P_{i,j}})$
15 Cmm(M, g)
16 $u \leftarrow 0$
17 **for** $i \leftarrow 1$ to k
18 **for** $j \leftarrow m_i$ downto 1
19 $R_{u+j} \leftarrow G_{i,2^{j-1}}$
20 **for** $s \leftarrow 1$ to $2^{j-1} - 1$
21 Cmm$(\{R_{u+j}, G_{i,s}\}, G_{i,s+2^{j-1}})$
22 $u \leftarrow u + m_i$
23 **return** R

Figure 3.1: Algorithm for batch exponentiation

Proof: Let X_i for $1 \leq i \leq k$ be a random variable such that $X_i = 0$ if $P_{i,j} = 0$, and $X_i = 1$ otherwise at the j-th iteration. Let $X = X_1 + X_2 + ... + X_k$ be the random variable that represents the number of non-zero $P_{i,j}$. Because $P_{i,j} = 0$ happens when bit values at bit position j for all exponents in X_i are zero, the probability that $P_{i,j} = 0$ is $\frac{1}{2^m}$. Thus, X_i is a Bernoulli random variable with $p = 1 - \frac{1}{m}$. Since X_i's are independent and identically distributed random variables, X is a binomial random variable with parameters k and $p = 1 - \frac{1}{2^m}$. Therefore, the probability that there are t non-zero $P_{i,j}$ values at the j-th iteration is $\binom{k}{t} p^t (1-p)^{k-t}$. □

Theorem 1 *With extended common-multiplicand Montgomery multiplication, the expected cost of multiplications of the k-way batch exponentiation is*

$$l\left(k\frac{b^2+2b+2}{2b^2+b}\left(1-\frac{1}{2^m}\right)+1\right)+k\left(\frac{b^2+2b+2}{2b^2+b}+1\right)(2^m-m-1) \quad (3.1)$$

where $b = \frac{size(p)}{w}$.

Proof: Let us consider the random variable X in Lemma 3.4.1 which represents the number of non-zero $P_{i,j}$ at the j-th iteration in the evaluation stage. By the definition of c_t in Section 2.1.3, the expected cost of multiplications at each iteration is $E[c_{X+1}] = E\left[\frac{b^2+2b+2}{2b^2+b}X + 1\right]$. The first term is derived from $lE\left[\frac{b^2+2b+2}{2b^2+b}X + 1\right] = l\left(\frac{b^2+2b+2}{2b^2+b}E[X] + 1\right) = l\left(\frac{b^2+2b+2}{2b^2+b}kp + 1\right)$ because there are l iterations and X follows the binomial distribution $B(k,p)$ where $p = 1 - \frac{1}{2^m}$. The overall cost of multiplications in the combination stage is $c_2 k \sum_{i=1}^{m} \left(2^{i-1} - 1\right)$ because Algorithm Comb is applied to each exponent group and the cost for two CMMs is reduced to c_2. Note that the c_{t+1} and c_2 become $t+1$ and 2, respectively, for Chung et al.'s algorithm where t is the value of the random variable X. □

3.4.2 Storage Requirement

The required amount of memory for k position arrays is nl bits. Since there are $k(2^m - 1)$ intermediate values from the evaluation stage, the total amount of

memory that is required for the whole procedure is $nl + (k(2^m - 1))\text{size}(p)$ bits.

3.5 Optimal Group Size

The expected cost of multiplications per exponentiation with extended common-multiplicand Montgomery multiplication is obtained by dividing the result of Theorem 1 by n:

$$l\left(\frac{b'}{m}\left(1 - \frac{1}{2^m}\right)\right) + \frac{l}{n} + \frac{1}{m}\left(b' + 1\right)\left(2^m - m - 1\right).$$

where $b' = \frac{b^2+2b+2}{2b^2+b}$. Since the second term $\frac{l}{n}$ does not change when n, l and b are given, the optimal integer value of m is obtained by finding the integer solution that minimizes the following:

$$F(l, b, m) = l\left(\frac{b'}{m}\left(1 - \frac{1}{2^m}\right)\right) + \frac{1}{m}\left(b' + 1\right)\left(2^m - m - 1\right). \tag{3.2}$$

The optimal values of m for various l and size(p) are shown in Table 3.2. The optimal group size of the k-way algorithm is smaller than that of Chung et al.'s. For example, given size(p) = 1024 and $l = 160$, the optimal value of m is 4 for the k-way algorithm but it is 5 for Chung et al.'s algorithm.

3.6 Parameters

We show how to choose optimal values for the size of an exponent group m and the number of exponent groups k when the number of exponents n, the bit-length of exponent l and size(p) are given. The computing environment can be restricted in terms of memory, or a sufficient amount of memory can be allowed. We consider both cases for the choice of parameters.

Given input parameters l and size(p), we first choose the optimal value of m from Table 3.2. With a sufficient amount of memory, the k-way algorithm is applied to n input exponents with the chosen m. If the algorithm runs on a computing device where the allowed amount of memory is restricted by M, the

m	Length of exponent l				
	Ours			Chung et al.	M'Raïhi Naccache
	$size(p) = 1024$	$size(p) = 2048$	$size(p) = 4096$		
4	$160 \sim 175$	$160 \sim 178$	$160 \sim 179$		$160 \sim 196$
5	$176 \sim 421$	$179 \sim 427$	$180 \sim 431$	$160 \sim 289$	$197 \sim 538$
6	$422 \sim 995$	$428 \sim 1011$	$432 \sim 1019$	$290 \sim 684$	$539 \sim 1433$
7	$996 \sim 1024$	$1012 \sim 2048$	$1020 \sim 2372$	$685 \sim 1594$	$1434 \sim 3714$
8			$2373 \sim 4096$	$1595 \sim 3657$	$3715 \sim 4096$
9				$3658 \sim 4096$	

Table 3.2: Optimal exponent group size m when $w = 32$.

m	Length of exponent l				
	Ours			Chung et al.	M'Raïhi Naccache
	$size(p) = 1024$	$size(p) = 2048$	$size(p) = 4096$		
4	$160 \sim 170$	$160 \sim 175$	$160 \sim 178$		$160 \sim 196$
5	$171 \sim 408$	$176 \sim 421$	$179 \sim 427$	$160 \sim 289$	$197 \sim 538$
6	$409 \sim 965$	$422 \sim 995$	$428 \sim 1011$	$290 \sim 684$	$539 \sim 1433$
7	$966 \sim 1024$	$996 \sim 2048$	$1012 \sim 2354$	$685 \sim 1594$	$1434 \sim 3714$
8			$2355 \sim 4096$	$1595 \sim 3657$	$3715 \sim 4096$
9				$3658 \sim 4096$	

Table 3.3: Optimal exponent group size m when $w = 64$.

optimal batch size $n_b = k_b m$ is calculated by picking the largest integer value k_b that satisfies the inequality $M \geq k_b(ml + (2^m - 1)\text{size}(p))$ which is obtained by storage analysis in Section 3.4. Then the k-way algorithm with the chosen m is applied for every n_b exponents of the given n input exponents.

Chapter 4

Experimental Results and Comparison

In this chapter, we compare the time-memory tradeoff of the k-way algorithm with two previous batch exponentiation algorithms, i.e. M'Raïhi-Naccache and Chung et al.'s algorithm, which decompose exponents into groups but do not apply a CMM method in the evaluation and combination stages. We also compare the performance in terms of cost of multiplications with Chung et al.'s.

For comparisons, the cost of multiplications is counted in the evaluation and combination stages for randomly generated exponents. At each iteration of the evaluation stage, the number of non-zero $P_{i,j}$ values t is counted, which results in the cost of multiplications c_{t+1}. In the combination stage, the cost of multiplications of each exponent group X_i is counted as $c_2(2^{m_i} - m_i - 1)$, where m_i is the number of exponents in X_i. Each experiment is carried out 50 times, given particular values for parameters n, l and size(p). The costs of multiplications obtained from Theorem 1 and experiments are almost identical.

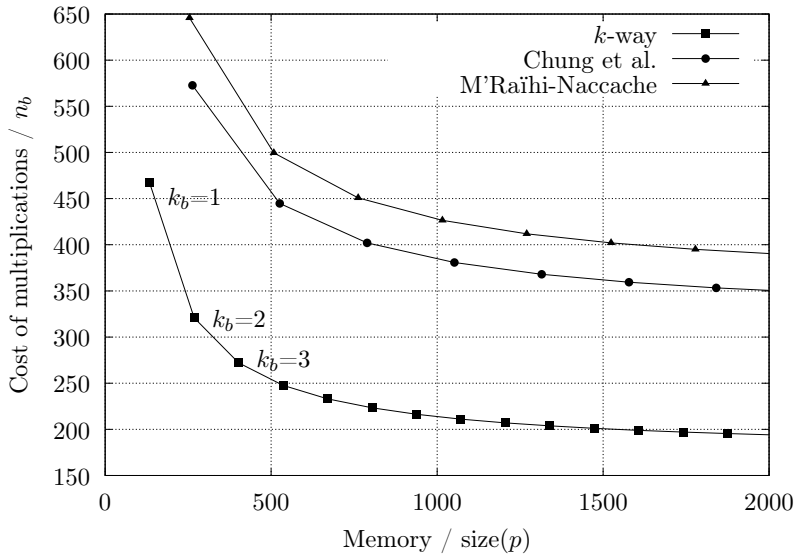

Figure 4.1: Tradeoff comparison when $l = 2048$ and size$(p) = 2048$

4.1 Time-Memory Tradeoff

The time-memory tradeoffs of the k-way algorithm, Chung et al.'s and M'Raïhi-Nacacche's are compared in Figure 4.1 for size$(p) = 2048$ and $l = 2048$. The optimal values of m for the k-way, Chung et al.'s and M'Raïhi-Nacacche's are 7, 8 and 7, respectively, from Table 3.2. The x-axis represents the required amount of memory divided by size(p), and the y-axis represents the cost of multiplications for n_b exponents divided by n_b. A point on each line corresponds to a particular value for k_b. The figure shows that the k-way algorithm achieves a better tradeoff compared to other algorithms.

4.2 Cost of Multiplications

Figure 4.2 shows the cost of multiplications of the k-way algorithm and Chung et al.'s as n grows when the bit length of exponents l and the size of modulus size(p) vary from 1024 to 4096, and a sufficient amount of memory is allowed,

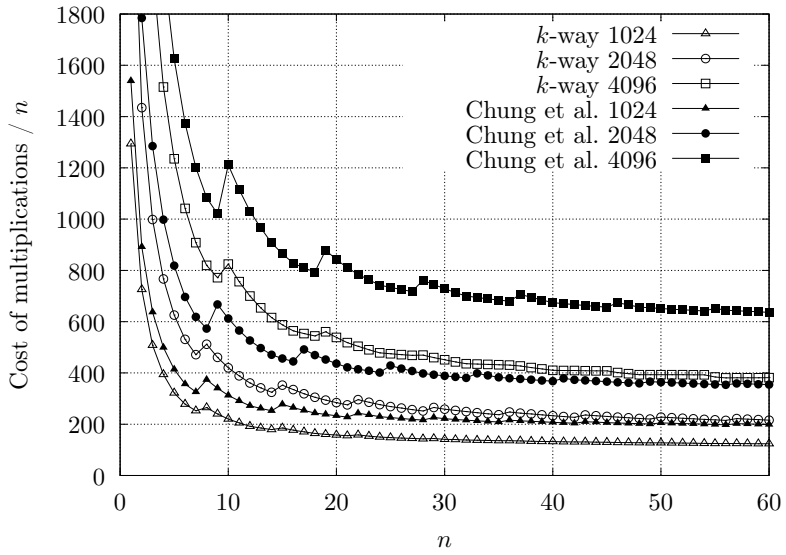

Figure 4.2: Performance comparison

i.e. all exponentiations are computed at once. As in Section 4.1, l and size(p) are set to be identical for each line, and the optimal value of the exponent group size m for each line is chosen from Table 3.2. Increasing steps in lines occur when n is increased by 1 from a multiple of m. The k-way algorithm outperforms Chung et al.'s since the extended CMM Montgomery multiplication reduces the cost of multiplications in the evaluation stage from $t+1$ to c_{t+1} as analyzed in Theorem 1.

Table 4.1 shows costs of multiplications of the k-way algorithm and Chung et al.'s for various values for (size(p), l) including parameters (1024, 160), (2048, 224), (2048, 256) and (3072, 256) from Digital Signature Standard (DSS) [28]. The number of input exponents n is assumed to be a multiple of the exponent group size m and the word size w is set to 64. The costs of multiplications in the fifth and seventh columns are calculated from the result of Theorem 1. The table shows that we obtained the experimental result for randomly generated exponents as expected in the analysis in Section 3.4.

size(p)	b'	l	Chung et al.		k-way	
			m	Cost of Mults	m	Cost of Mults
1024	0.549	160	5	$41.4 + \frac{160}{n}$	4	$24.9 + \frac{160}{n}$
		1024	7	$179.4 + \frac{1024}{n}$	7	$106.3 + \frac{1024}{n}$
2048	0.524	224	5	$53.8 + \frac{224}{n}$	4	$30.7 + \frac{224}{n}$
		256	5	$60.0 + \frac{256}{n}$	4	$33.9 + \frac{256}{n}$
		2048	8	$316.8 + \frac{2048}{n}$	7	$178.2 + \frac{2048}{n}$
3072	0.516	256	5	$60.0 + \frac{256}{n}$	4	$33.5 + \frac{256}{n}$
		3072	8	$444.3 + \frac{3072}{n}$	8	$244.1 + \frac{3072}{n}$
4096	0.512	4096	9	$565.8 + \frac{4096}{n}$	8	$307.7 + \frac{4096}{n}$

Table 4.1: Costs of multiplications for various parameters

4.3 Running Time

In order to demonstrate that the k-way algorithm is practically improved upon Chung et al.'s, the actual running time is also measured by implementing the whole procedure of both algorithms. The actual running time is measured on a machine with an Intel i5 2.7GHz CPU and 8GB memory. All algorithms including Montgomery multiplication are implemented in C++ using g++ and GNU MP library with -O2 option. Since the experiment is conducted on a 64-bit computing environment, parameters are chosen by setting $w = 64$.

Figure 4.3 shows the measured running time of the k-way algorithm and Chung et al.'s in seconds when $l = \text{size}(p)$ vary from 1024 to 4096. We assumed that a sufficient amount of memory is allowed, and the optimal values for the exponent group size m is chosen from Table 3.2 as in Section 4.2. The k-way algorithm also performs better than Chung et al.'s with respect to the actual running time in the real world.

Table 4.2 shows performance improvement that is achieved by the k-way algorithm. The values in the third column indicates how much cost of mul-

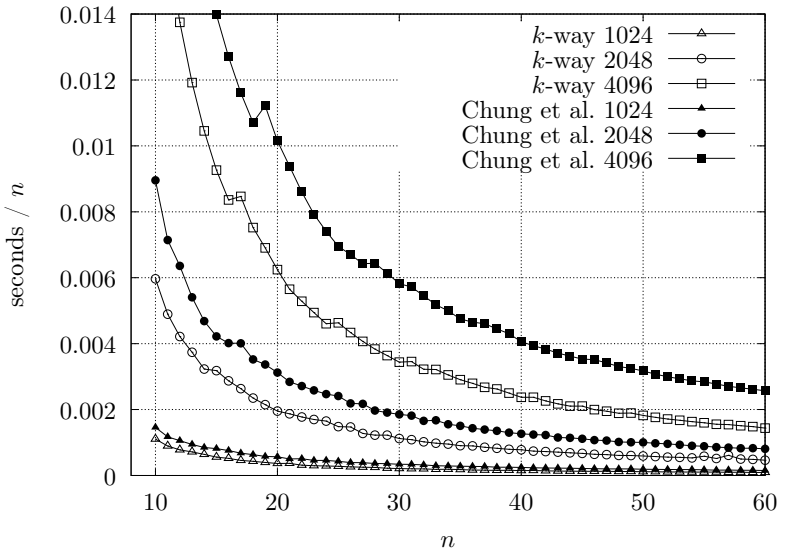

Figure 4.3: Running time comparison

$l, \text{size}(p)$	n	Cost of Mults	Running Time
1024	$10 \sim 60$	$31.1 \sim 41.5\%$	$23.9 \sim 42.4\%$
2048	$10 \sim 60$	$22.8 \sim 39.8\%$	$24.6 \sim 42.7\%$
4096	$10 \sim 60$	$23.9 \sim 41.1\%$	$27.1 \sim 44.1\%$

Table 4.2: Improvement upon Chung et al.'s

tiplications is reduced by k-way algorithm compared to Chung et al.'s. For example, the k-way algorithm reduces the cost of multiplications of Chung et al.'s by $22.8\% \sim 39.8\%$ when n is $10 \sim 60$, and $l = \text{size}(p)$ is 2048. Likewise, the fourth column shows the percentage of the reduced running time by the k-way algorithm. For instance, the k-way algorithm reduces the running time of Chung et al.'s by $24.6\% \sim 42.7\%$ when n is $10 \sim 60$, and $l = \text{size}(p)$ is 2048. The result of the experiment shows that we obtained the improvement ratio with respect to running time as expected from the result in terms of the cost of multiplications.

Chapter 5

Further Improvement

In this chapter, we introduce methods for further improvement: exponent ordering and precomputation. While the exponents are grouped in a straightforward way in Section 3.1, the exponent ordering algorithm takes a heuristic approach to rearrange the input exponents so that the cost of multiplications in the evaluation stage can be reduced. On top of that, we show how to utilize the fixed base property to speed up the exponentiation in the computing environment with a sufficient amount of offline storage.

5.1 Exponent Ordering

The exponent grouping and partitioning stage of the k-way algorithm groups the input exponents in a straightforward way : X_i contains $x_s, x_s+1, ..., x_{s+m_i-1}$ where $s = \sum_{j=1}^{i-1} m_j$ and m_j is the number of input exponents of group X_j as defined in Section 3.1. However, there can be an optimal order of the input exponents that results in the minimum cost of multiplications in the evaluation stage.

There are bit patterns of a set of exponents that benefits from the exponent

Input		Not Rearranged	Rearranged
		Group 1	Group 1
		00101110	10111000
00101110		10111000	11101000
10111000		00111010	10101000
00111010		11101000	11011000
11101000	→		
00110110		Group 2	Group 2
10101000		00110110	00101110
00101010		10101000	00111010
11011000		00101010	00110110
		11011000	00101010
		$o_1 = 11111110$	$o_1 = 11111000$
		$o_2 = 11111110$	$o_2 = 00111110$

Table 5.1: An example of exponent ordering

intersection method with this observation. To produce exponents of such a bit pattern, we can choose a set of s indexes of non-zero bit columns for each exponent group. Then for each group, we assign random bits to the non-zero bit columns for exponents ensuring that a non-zero bit column has at least one exponent that has 1 for that bit column. Although such a biased bit pattern is not usual for a set of given input exponents, the k-way algorithms can benefit from rearranging for the random exponents.

Let $o_i = \bigvee_{x \in X_i} x$. The optimal order of input exponents also minimizes $\sum_{i=1}^{k} w(o_i)$ where $w(o_i)$ is the number of 1's in o_i, since the cost of multiplica-

tions in the evaluation stage can be stated as follows:

$$\begin{aligned} cost_{eval} &= \sum_{j=0}^{l-1} c_{\sum_{i=1}^{k} o_i[j]+1} \\ &= \sum_{j=0}^{l-1} \left(b' \left(\sum_{i=1}^{k} o_i[j] \right) + 1 \right) \\ &= l + b' \sum_{j=0}^{l-1} \sum_{i=1}^{k} o_i[j] \\ &= l + b' \sum_{i=1}^{k} w(o_i). \end{aligned}$$

Table 5.1 shows an example of exponent ordering. If the input exponents are not rearranged, the cost of multiplications in the evaluation stage is 10.5 assuming that $c_t = \frac{t+1}{2}$. However, the cost is reduced to 8.5 when the input exponents are rearranged and grouped as in the table.

Finding an optimal order of exponents is not straightforward. There is a similar problem that partitions a given set of integers into k groups in a way that the sum of the integers in each partition are as near as possible. This multi-way partitioning problem is NP-complete. In the k-way algorithm, the size for each partition is given, and the objective of the optimization is to minimize $\sum w(o_i)$. Although the optimal order might not be found in a polynomial time, we can take a greedy approach for a heuristic algorithm. For group i, we pick an integer x amongst the set of input exponents X that satisfies $\min_{x \in X} w(o \vee x)$ where o has the initial value 0 for each group i. Then we remove the selected exponent x from X, and set o to $o \vee x$. This process is repeated until group i has m_i exponents. The whole procedure is described in Algorithm REARRANGEEXP of Figure 5.1.

Table 5.2 shows the improvement achieved by exponent ordering when size$(p) = 4096$ and random exponents are given. The costs of multiplication are measured for randomly generated exponents and rearranged exponents. Improvement ratios are obtained by measuring the reduced costs of multipli-

```
REARRANGEEXP(X, n, m)
 1  k ← ⌈n/m⌉
 2  X' ← {}
 3  for i ← 1 to k
 4      if i ≤ n mod k
 5          m_i ← ⌈n/k⌉
 6      else
 7          m_i ← ⌊n/k⌋
 8      o ← 0
 9      for j ← 1 to m_i
10          e ← min_{x∈X} w(o ∨ x)
11          X'.append(e)
12          X.remove(e)
13          o ← o ∨ e
14  return X'
```

Figure 5.1: Algorithm for Ordering Exponents

cations by the exponent ordering. The proposed exponent ordering algorithm improves the performance of the k-way algorithm as shown in the table. On the other hand, the improvement ratio decreases as the exponent group size grows, because larger exponent group size leads to less number of bit columns that have 0 for all group exponents.

5.2 Precomputation

In applications where the common base g and the modulus p are fixed and known in advance, we can consider an offline precomputation technique to speed up exponentiation when a sufficient amount of offline storage is allowed. One

l	m	n	Improvement Ratio
160	4	$8 \sim 100$	$0.73 \sim 4.45$ %
256	5	$10 \sim 100$	$0.39 \sim 2.82$ %
512	6	$12 \sim 102$	$0.28 \sim 1.53$ %
1024	7	$12 \sim 105$	$0.13 \sim 0.85$ %
2048	7	$14 \sim 105$	$0.12 \sim 0.64$ %
4096	8	$16 \sim 104$	$0.08 \sim 0.47$ %

Table 5.2: Improvement by exponent ordering when size(p) = 4096 for random exponents

simple method is to precompute the intermediate values g^{2^i} in the repeated squaring process [23]. Brickell et al. (BGMW) proposed a fixed-base windowing technique where some powers g^{x_i} are precomputed and exponentiation is performed for the decomposed input exponent $x = \sum a_i x_i$. Algorithms by Lim-Lee [32] and De Rooji [12] take advantage of vector addition chains to compute exponentiation in the form of a product of multiple powers $g^x = g_0^{x_0} g_1^{x_1} ... g_{h-1}^{x_{h-1}}$ where $x = \sum x_i 2^i$ and $g_i = g^{2^{\frac{il}{h}}}$. Lim-Lee proposed a fixed-base comb method that provides a simple and efficient vector addition chain by decomposing exponents into $h \times v$ subexponents of bit length $\frac{l}{hv}$. Then exponentiation is efficiently preformed with precomputed values $\prod_{t=0}^{h-1} g^{2^{at+bj} i[t]}$ for $0 \leq i < 2^h$ and $0 \leq j < v$ where $a = \frac{l}{h}$ and $b = \frac{a}{v}$. The efficiency of precomputation method also can be improved by adopting various number systems for representing exponents such as non-adjacned form [36, 45, 48], m-ary [20] and a double-base number system [14,15,16]. In case a power g^x is generated when an exponent is not given, g^x can be calculated from a subset of the precomputed pairs $(a_1, g^{a_1}), (a_2, g^{a_2}), ..., (a_n, g^{a_n})$ by adding all exponents and multiplying all powers in the chosen subset [6].

There are some obstacles to adopt precomputation techniques above for

the generalized intersection method. Number systems that have negative digits such as signed-digit representation and non-adjacent form reduce the number of non-zero digits. However, they are not suitable for modular exponentiation because of the expensive inversion operation. For other number systems, we need to find how to apply the exponent intersection method to produce mutual exclusive partitioned exponents. Methods for vector addition chains cannot be utilized directly because the inputs g and $\{C_0, C_1, ..., C_{2^m-1}\}$ and outputs $\{g^{C_1}, g^{C_2}, ..., g^{C_{2^m-1}}\}$ of the evaluation stage have different forms compared to those of exponentiation methods for vector addition chains. An online power generating method such as [6] cannot be applied when input exponents are given. Devising a sophisticated offline precomputation method for the generalized intersection method utilizing some of these methods is an interesting open research topic.

In this thesis, we apply a simple precomputation method that precomputes the intermediate values g^{2^j} of the repeated squaring process. At iteration j in the evaluation stage, g^{2^j} is obtained from the precomputed values and multiplied to corresponding cells for each exponent group that has a non-zero element at position j in its position array. Since squaring is not required, the number of common-multiplicand multiplications is reduced by 1 at each iteration. Thus, the total cost of multiplications with precomputation is

$$lb'k\left(1-\frac{1}{2^m}\right)+l(1-b')+k(b'+1)\left(2^m-m-1\right) \tag{5.1}$$

where $b'=\frac{b^2+2b+2}{2b^2+b}$ and $b=\frac{\text{size}(p)}{w}$, and the cost of multiplications per exponentiation can be obtained by dividing Equation 5.1 by n as follows:

$$\frac{lb'}{m}\left(1-\frac{1}{2^m}\right)+\frac{l}{n}(1-b')+k(b'+1)\left(2^m-m-1\right). \tag{5.2}$$

The optimal group size for given l, size(p) and w is same as in Section 3.5. The additional required amount storage for the offline precomputation values is $(l-1)$size(p) bits.

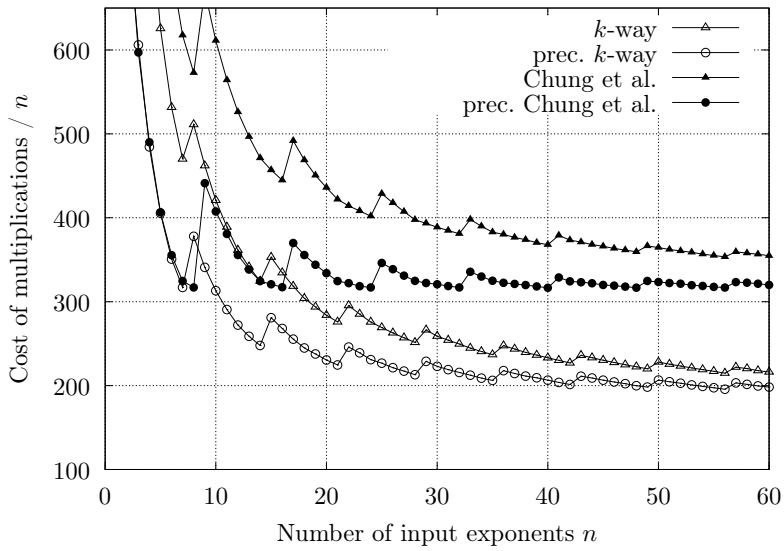

Figure 5.2: Cost of multiplications by precomputation when $l = \text{size}(p) = 2048$

Figure 5.2 shows the cost of multiplications of the k-way algorithms and Chung et al.'s with the precomputed intermediate values of the repeated squaring process when $l = \text{size}(p) = 2048$. The k-way algorithm reduces the cost of multiplications by about $12\% \sim 40\%$ compared to Chung et al.'s when both algorithms use the precomputation technique and $n = 10 \sim 60$. Compared to the k-way algorithm without precomputation, the cost of multiplications is reduced by about $8\% \sim 25\%$ for the same range of n, and the improvement ratio decreases as n grows because the squaring cost is amortized over the number of input exponents.

Chapter 6

Conclusion

In this thesis, we have proposed the k-way algorithm that efficiently computes multiple modular exponentiations simultaneously. First, we have introduced an improved batch exponentiation algorithm by utilizing the extended common-multiplicand multiplication method which evaluates more than two multiplications simultaneously. When the input exponents are divided into k groups, the multiplications at each iteration in the evaluation stage share the same multiplicand, and the number of multiplications is bounded by $k + 1$. Multiplications at each iteration in the combination stage also share the same multiplicand. These multiplications are handled by the extended common-multiplicand Montgomery multiplication algorithm that computes arbitrary number of multiplications that share a same multiplicand.

Second, we have analyzed the performance of the k-way algorithm in terms of the cost of multiplications which is defined with respect to the number of single precision multiplications. We also analyzed the optimal algorithm parameters for both computing environments: the amount of memory is restricted, or a sufficient amount of memory is allowed. The comparison of time-memory tradeoffs shows that the k-way algorithm performs better than previous algo-

rithms given the same amount of memory. When a sufficient amount of memory is allowed, the k-way algorithm reduces the cost of multiplications of Chung et al.'s by $23\% \sim 41\%$ when n is $10 \sim 60$, and $l = \text{size}(p)$ is 1024, 2048 and 4096.

Third, we have implemented the whole procedure of the k-way algorithm to show that our algorithms performs better than Chung et al.'s in the real world. The algorithm including the extended common-multiplicand Montgomery multiplication has been implemented in C++ using GNU MP library on a 64-bit machine. The optimal algorithm parameters for the word size 64 are chosen as analyzed in the thesis, and the experimental results have shown that the k-way algorithm reduces the running time of Chung et al.'s by $24\% \sim 44\%$ when n is $10 \sim 60$, and $l = \text{size}(p)$ is 1024, 2048 and 4096.

Finally, we have presented an exponent ordering algorithm and a precomputation method for further improvement. With the exponent ordering algorithm, exponents are grouped in a way that the number of 1's in the partitioned exponents is reduced for each group. We also have presented a simple offline precomputation method for the evaluation stage where the intermediate values of the repeated squaring process are computed in advance and stored in the offline storage.

Bibliography

[1] J. A. Akinyele, M. Green, S. Hohenberger, and M. Pagano. Machine-generated algorithms, proofs and software for the batch verification of digital signature schemes. *Journal of Computer Security*, **22**(6):867–912, 2014.

[2] P. Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In *Advances in Cryptology — CRYPTO' 86*, pages 311–323. Springer, 1987.

[3] M. Bellare, J. A. Garay, and T. Rabin. Fast batch verification for modular exponentiation and digital signatures. In *Advances in Cryptology — EUROCRYPT '98*, pages 236–250. Springer, 1998.

[4] D. J. Bernstein, J. Doumen, T. Lange, and J.-J. Oosterwijk. Faster batch forgery identification. In *Progress in Cryptology - INDOCRYPT 2012*, pages 454–473. Springer, 2012.

[5] C. Boyd and C. Pavlovski. Attacking and repairing batch verification schemes. In *Advances in Cryptology — ASIACRYPT '00*, pages 58–71. Springer, 2000.

[6] V. Boyko, M. Peinado, and R. Venkatesan. Speeding up discrete log and factoring based schemes via precomputations. In K. Nyberg, editor, *Advances in Cryptology — EUROCRYPT '98*, pages 221–235, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[7] E. F. Brickell, D. M. Gordon, K. S. McCurley, and D. B. Wilson. Fast exponentiation with precomputation. In *Advances in Cryptology — EUROCRYPT' 92*, pages 200–207. Springer.

[8] J. Camenisch, S. Hohenberger, and M. O. Pedersen. Batch verification of short signatures. In *Advances in Cryptology — EUROCRYPT '07*, volume 4515, pages 246–263. Springer, 2007.

[9] J. H. Cheon and D. H. Lee. Use of sparse and/or complex exponents in batch verification of exponentiations. *IEEE Transactions on Computers*, **55**(12):1536–1542, 2006.

[10] B. Chung, J. Hur, H. Kim, S.-M. Hong, and H. Yoon. Improved batch exponentiation. *Information Processing Letters*, **109**(15):832 – 837, 2009.

[11] B. Chung, S. Marcello, A.-P. Mirbaha, D. Naccache, and K. Sabeg. Operand folding hardware multipliers. In *Cryptography and Security: From Theory to Applications*, pages 319–328. Springer, 2012.

[12] P. de Rooij. Efficient exponentiation using precomputation and vector addition chains. In *Advances in Cryptology — EUROCRYPT '94*, pages 389–399. Springer.

[13] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, **22**(6):644–654, 1976.

[14] V. Dimitrov, L. Imbert, and P. Mishra. The double-base number system and its application to elliptic curve cryptography. *Mathematics of Computation*, **77**(262):1075–1104, 2008.

[15] V. S. Dimitrov, G. A. Jullien, and W. C. Miller. An algorithm for modular exponentiation. *Information Processing Letters*, **66**(3):155–159, 1998.

[16] C. Doche and L. Imbert. Extended double-base number system with applications to elliptic curve cryptography. In *Progress in Cryptology - INDOCRYPT '06*, pages 335–348. Springer, 2006.

[17] P. Downey, B. Leong, and R. Sethi. Computing sequences with addition chains. *SIAM Journal on Computing*, **10**(3):638–646, 1981.

[18] S. R. Dussé and B. S. Kaliski. A cryptographic library for the Motorola DSP56000. In *Advances in Cryptology — EUROCRYPT '90*, pages 230–244. Springer, 1991.

[19] T. Elgamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, **31**(4):469–472, Jul 1985.

[20] B. Feix and V. Verneuil. There's something about m-ary. In *Progress in Cryptology – INDOCRYPT '13*, pages 197–214. Springer, 2013.

[21] A. L. Ferrara, M. Green, S. Hohenberger, and M. Ø. Pedersen. Practical short signature batch verification. In *Topics in Cryptology – CT-RSA '09*, pages 309–324. Springer, 2009.

[22] A. Fiat. Batch RSA. *Journal of Cryptology*, **10**(2):75–88, Mar 1997.

[23] R. Fuji-Hara. Cipher algorithms and computational complexity. *Bit*, 17:954–959, 1985.

[24] D. M. Gordon. A survey of fast exponentiation methods. *Journal of Algorithms*, **27**(1):129 – 146, 1998.

[25] J.-C. Ha and S.-J. Moon. A common-multiplicand method to the Montgomery algorithm for speeding up exponentiation. *Information Processing Letters*, **66**(2):105 – 107, 1998.

[26] S.-M. Hong, S.-Y. Oh, and H. Yoon. New modular multiplication algorithms for fast modular exponentiation. In *Advances in Cryptology — EUROCRYPT '96*, pages 166–177. Springer, 1996.

[27] S. Karati, A. Das, D. Roychowdhury, B. Bellur, D. Bhattacharya, and A. Iyer. New algorithms for batch verification of standard ECDSA signatures. *Journal of Cryptographic Engineering*, **4**(4):237–258, 2014.

[28] C. F. Kerry and P. D. Gallagher. Digital signature standard (DSS). *FIPS PUB*, pages 186–4, 2013.

[29] D. E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[30] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, **48**(177):203–209, 1987.

[31] C. K. Koç. Analysis of sliding window techniques for exponentiation. *Computers & Mathematics with Applications*, **30**(10):17–24, 1995.

[32] C. H. Lim and P. J. Lee. More flexible exponentiation with precomputation. In *Advances in Cryptology — CRYPTO'94*, pages 95–107. Springer, 1994.

[33] D. C. Lou and C. C. Chang. Fast exponentiation method obtained by folding the exponent in half. *Electronics Letters*, **32**(11):984–985, May 1996.

[34] A. J. Menezes, S. A. Vanstone, and P. C. V. Oorschot. *Handbook of Applied Cryptography*. CRC Press, 1996.

[35] V. S. Miller. Use of elliptic curves in cryptography. In *Advances in Cryptology — CRYPTO '85*, pages 417–426. Springer, 1986.

[36] N. A. F. Mohamed, M. H. A. Hashim, and M. Hutter. Improved fixed-base comb method for fast scalar multiplication. In *Progress in Cryptology - AFRICACRYPT '12*, pages 342–359. Springer, 2012.

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

[38] F. Morain and J. Olivos. Speeding up the computations on an elliptic curve using addition-subtraction chains. *Theoretical Informatics and Applications*, 24:531–543, 1990.

[39] D. M'Raïhi and D. Naccache. Batch exponentiation: A fast DLP-based signature generation strategy. In *Proceedings of the 3rd ACM Conference on Computer and Communications Security*, pages 58–61. ACM, 1996.

[40] D. Naccache, D. M'RaÏhi, S. Vaudenay, and D. Raphaeli. Can D.S.A. be improved? — complexity trade-offs with the digital signature standard —. In *Advances in Cryptology — EUROCRYPT '94*, pages 77–85. Springer, 1995.

[41] H. Park, K. Park, and Y. Cho. Analysis of the variable length nonzero window method for exponentiation. *Computers & Mathematics with Applications*, **37**(7):21–29, 1999.

[42] C. Pavlovski and C. Boyd. Efficient batch signature generation using tree structures. In *International Workshop on Cryptographic Techniques and E-Commerce, CrypTEC*, volume 99, pages 70–77, 1999.

[43] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, **26**(1):96–99, Jan. 1983.

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

[45] W.-J. Tsaur and C.-H. Chou. Efficient algorithms for speeding up the computations of elliptic curve cryptosystems. *Applied Mathematics and Computation*, **168**(2):1045–1064, 2005.

[46] C.-L. Wu. An efficient common-multiplicand-multiplication method to the Montgomery algorithm for speeding up exponentiation. *Information Sciences*, **179**(4):410 – 421, 2009.

[47] T.-C. Wu and Y.-S. Chang. Improved generalisation common-multiplicand multiplications algorithm of Yen and Laih. *Electronics Letters*, **31**(20):1738–1739, Sep 1995.

[48] W. Yang, K. Lin, and C. Laih. A precomputation method for elliptic curve point multiplication. *Journal - Chinese Institute of Electrical Engineering*, **9**(4):339–344, 2002.

[49] S.-M. Yen. Improved common-multiplicand multiplication and fast exponentiation by exponent decomposition. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, **80**(6):1160–1163, 1997.

[50] S. M. Yen and C. S. Laih. Common-multiplicand multiplication and its applications to public key cryptography. *Electronics Letters*, **29**(17):1583–1584, Aug 1993.

[51] S.-M. Yen and C.-S. Laih. Improved digital signature suitable for batch verification. *IEEE Transactions on Computers*, **44**(7):957–959, 1995.

[52] S. M. Yen, C. S. Laih, and A. K. Lenstra. Multi-exponentiation (cryptographic protocols). *IEE Proceedings - Computers and Digital Techniques*, **141**(6):325–326, Nov 1994.

[53] T.-Y. Youn, Y.-H. Park, T. Kwon, S. Kwon, and J. Lim. Efficient flexible batch signing techniques for imbalanced communication applications. *IEICE Transactions on Information and Systems*, **91**(5):1481–1484, 2008.

초 록

그룹의 원소를 거듭제곱으로 증가시키는 누승 연산은 많은 공개키 암호 시스템에서 핵심적으로 사용되는 연산이다. 누승 연산에 주어지는 지수는 큰 수인 경우가 많아 연산에 많은 자원을 필요로 한다. 누승 연산의 성능을 향상시키는 두 가지 접근법이 있다. 첫째는 곱셈 연산의 효율을 향상시키는 것이다. 다른 방법은 누승 연산에 필요한 곱셈의 수를 줄이는 것이다.

본 논문에서는 곱셈 연산 비용 측면에서 다수의 누승 연산을 일괄 계산하는 지수 교차 기법을 두 가지 접근법을 모두 고려하여 개선한 알고리즘을 제안한다. 지수들을 여러 그룹으로 나누어 지수 교차 기법을 적용할 경우 공통피승수를 가지는 여러 개의 곱셈이 반복적으로 수행됨에 주목한다. 제안한 알고리즘의 성능 향상은 공통피승수를 가지는 임의의 수의 곱셈 연산들을 한 번에 처리하는 확장된 공통피승수 곱셈 기법의 적용을 통해 얻을 수 있다. 이렇게 개선된 알고리즘은 이전 연구 결과보다 더 적은 메모리를 사용하여 빠른 시간에 일괄 누승 연산을 수행할 수 있다.

제안된 알고리즘의 곱셈 연산 비용 측면에서의 수행시간과 소요 메모리를 분석하고, 메모리의 양이 제한되거나 충분한 환경에서 지수 그룹의 크기의 최적값을 선택하는 방법을 제시한다. 실험결과를 통해 지수의 수가 $10 \sim 60$이고 그 비트 길이가 1024, 2048, 4096일 때 곱셈 연산 비용이 이전 연구 대비 $23\% \sim 41\%$ 가량 감소함을 보인다. 또한 실제로 구현하여 얻은 수행시간 역시 이전 연구 대비 감소하여 현실 세계에서도 제안한 알고리즘이 더 효율적임을 보인다.

끝으로 지수들을 지수 교차 기법에서 효율이 좋은 그룹으로 나누는 알고리즘을 제시하고 이를 통해 곱셈 연산 비용을 줄일 수 있음을 보인다. 그리고 저장공간이 충분히 주어지는 환경에서 밑수의 거듭제곱 값들을 선계산을 통해 저장하여 이를 지수 교차 기법에서 활용하는 방법을 설명하고, 이 때 감소하는 곱셈 연산 비용에 대해 분석한다.

주요어: 암호학, 누승 연산, 공개키 암호, 공통피승수, 알고리즘
학번: 2009-20820