通信情報保護論文誌
第9券 第2號 1999. 06

고속 RSA 암호 시스템을 위한 몽고메리 알고리즘의 구현 및 분석

안 준언*, 유 기영**

Implementation and Analysis of the Montgomery Algorithm for the Fast RSA Crypto-System

Joon-En Ahn*, Kee-Young You**

요 약

공개키 암호 시스템에서는 보안을 위해 512 비트 이상의 큰 정수에 대하여 모듈러 지수승 연산을 수행하며, 모듈러 지수승은 모듈러 곱셈의 연속으로 나타내어진다. 본 논문에서는 가장 빠른 모듈러 곱셈 알고리즘인 몽고메리 알고리즘을 기반으로 하여 모듈러 곱셈의 수행 시간을 향상시키는 구현 방식을 제안한다. 제안된 방식을 각각 일반 수 체계에서와 GF(2m)상에서 구현하여 Koc 등이 제시한 방식과 곱셈, 덧셈, 메모리 읽기와 쓰기의 수행 횟수를 비교 분석하였다.

ABSTRACT

Public key cryptosystems carry out the modular exponentiation of large integer for security, and the modular exponentiation is presented by the serial of the modular multiplication. This thesis presents implementation method for improving performance of modular multiplication based on the Montgomery algorithm which is known as the fastest modular multiplication algorithm. These implementation method is implemented in integer system and in Galois Field(2m), and is compared and analyzed with multiplication, addition, reading and writing memory.

I. 서 론

1976년 W. Diffie와 M. E. Hellman이 공개키 암호 시스템의 개념을 제안하였다[1]. 그 후 Rivest, Shamir, Adleman에 의해 구현된 RSA 암호 시스템은 뛰어난 안전성, 키 분배와 관리의 용이, 인증과 디지털 서명(digital signature)이 가능한 장점으로 인해 인정받고 있다[2]. 최근에는 유한체(finite field) GF(2m)상에서 ElGamal형 공개키 암호 시스템이나 Elliptic Curve 암호 시스템의 연구가 많이 이루어지고 있다. RSA 암호 시스템은 실제 데이터 메시지 M에 대하여 암호와 복호시 안전성을 위해 512비트 이상의 큰 수의 모듈러 지수승(modular exponentiation) $M^e \bmod N$으로 나타내어진다. ElGamal형 공개키 암호 시스템에서도 $g^x \bmod p$의 모듈러 지수승이 사용된다[3]. 또한 이 모듈러 지수승 연산은 $A * B \bmod N$형태의 모듈러 곱셈(modular multiplication)의 반복적 수행으로 계산된다. .
그러나 암호와 복호시 512비트 이상의 큰 수에 대하여 계산하기 때문에 처리속도의 지연이 큰 단점이다. 처리 속도를 높이기 위해서 하드웨어로 설계하거나, 소프트웨어적으로는 모듈러 곱셈의 수행 횟수

* 경북대학교 컴퓨터공학과 병렬처리 연구실(rocahn@kyungpook..ac.kr)
** 경북대학교 컴퓨터공학과 병렬처리연구실(yook@bh.kyungpook.ac.kr)
※ 본 연구는 97, 98 년 정보통신부 대학기초연구지원사업에 의한 연구 결과입니다.

를 줄이거나 모듈러 지수승 내의 모듈러 곱셈의 처리 속도를 높이는 두 가지 방법이 있다[4,5,6,7,8]. 모듈러 곱셈으로는 몽고메리 알고리즘이 가장 빠르고 효과적인 것으로 알려져 있다[9]. 몽고메리 알고리즘은 크게 곱셈 단계와 감소 단계로 나눌 수 있다.

본 논문에서는 몽고메리 알고리즘을 분석, 변형하여 보다 빠른 수행을 하는 알고리즘을 제안하였다. Koc 등이 제시한 방식인 SOS (Separated Operand Scanning)방식을 분석하여 성능을 향상시켰으며, 몽고메리 알고리즘에서의 곱셈 단계와 감소 단계에서의 중복되는 계산을 피하고, 중간값 m을 계산할 때 두 단계와의 중복되는 계산을 피함으로써, 덧셈, 메모리 읽기와 쓰기의 횟수를 보다 적게 하여 빠른 수행을 하는 알고리즘 구현방식을 제안한다. 또한 Koc 등이 제시한 방식과 본 논문에서 제안된 방식을 GF(2m)상에서 구현하며 일반 정수 체계에서의 구현과 비교한다. 제안된 방식에 대하여 곱셈, 덧셈, 메모리 읽기와 쓰기의 횟수에 대하여 분석하고, Koc 등이 제시한 5가지 알고리즘들과 비교한다.

II. 모듈러 곱셈 알고리즘

2.1 몽고메리(Montgomery) 알고리즘

P.L Montgomery는 $C = A * B \bmod N$을 계산하기 위한 효율적인 알고리즘을 제안하였다[10]. 몽고메리 알고리즘은 다음의 식과 같은 결과를 계산한다.

$$MMM(A, B, N, R) = A * B * R^{-1} \bmod N$$

여기에서 $A, B < N$ 이며, R은 기저(base)이며, N과 R은 서로소(relatively prime)이다.

2.1.1 일반 정수체계에서의 몽고메리 알고리즘

몽고메리 알고리즘은 임의의 값을 가지는 모듈러스(modulus) N에 의한 나눗셈을 수행하여 모듈러 감소를 수행하는 대신에 R에 의한 나눗셈과 모듈러 감소를 통해 수행한다. 여기서 R을 일반적으로 2의 멱수(power)로 선택함으로써 범용 컴퓨터의 빠른 연산 - shift 연산- 을 통해 다른 일반 모듈러 곱셈보다 간단하게 구현할 수 있을뿐 아니라 빠른 수행을 할 수 있다. R이 2의 멱승으로 표현되기 때문에 N을 홀수로 취함으로써 $gcd(R, N) = 1$을 만족한다.

P.L. Montgomery가 제안한 곱셈 알고리즘은 다음과 같다.[10]

알고리즘 1. 몽고메리 알고리즘

MMM(A, B, N, R)

1 $N' = -N^{-1} \bmod R$
2 $T = A * B$
3 $M = T * N' \bmod R$
4 $T = (T + M * N) \text{ div } R$
5 *if* $T \geq N$ *then return* $T - N$
6 *else return* T

여기서 $A = \sum_{i=0}^{s-1} A^i * r^i$, $B = \sum_{i=0}^{s-1} B^i * r^i$, $r^{s-1} \leq N < r^s$ 이고, $R = r^s$ 이다. 이때 N'는 다음의 성질을 만족하는 정수이다.

$$R * R^{-1} - N * N' = 1 \qquad (1)$$

R^{-1}과 N'값은 확장 유클리드 알고리즘을 통해 구할 수 있다.

실제 구현에 있어서, 큰 정수에 대한 연산을 수행하므로 각 수를 워드 단위로 나누어 계산한다. 한 워드의 크기를 w라고 하면 정수는 기수(radix) $W = 2^w$로 표현되는 값의 연속으로 본다. A를 k 비트의 정수라고 하면 한 워드의 크기를 w 비트, $s = k / w$ 라고 하면 A는 기수가 2^w인 s개의 워드로 구성된다. 예를 들어, A가 512 비트의 정수라고 하고, 한 워드의 크기를 8 비트로 하면, A는 각 기수가 28 인 64개이 워드로 구성된다.

2.1.2 GF(2^m)상에서의 몽고메리 알고리즘

GF(2^m)상의 원소(elements)를 표현하는 방식은 여러 가지가 있으며[11,12] 본 논문에서는 소프트웨어 구현에 적합한 다항식 표기 방식을 사용한다. GF(2^m)상의 한 원소 a는 차수(degree)가 $m - 1$ 보다 작거나 같은 다항식이며, 다음과 같이 표기한

다.

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

위 식에서 계수들은 $a^i \in GF(2)$이고, 한 원소 a는 $a = (a^{m-1}\ a^{m-2}\ \cdots\ a^1\ a^0)$로 나타낸다. $GF(2^m)$상의 두 원소 a와 b의 덧셈은 다항식 $a(x)$와 $b(x)$의 차수가 같은 계수들에 대해 GF(2)상에서 더하기를 함으로써 수행되어 진다. 이것은 XOR(Exclusive OR)연산을 하는 것과 동일하다. $GF(2^m)$상의 두 원소 a와 b의 곱셈은 GF(2)상에서 차수가 m인 기약 다항식 $n(x)$(irreducible polynomial)이 필요하다. $GF(2^m)$상의 곱 $c = a * b$는 $c(x) = a(x)b(x)\ mod\ n(x)$를 계산함으로써 얻을 수 있다. $c(x)$는 차수가 $m-1$이하인 다항식이고, $c \in GF(2^m)$이다. 따라서, $GF(2^m)$상의 곱셈연산은 대응하는 다항식의 곱을 기약 다항식 $n(x)$으로 모듈러 연산을 함으로서 수행된다.
$GF(2^m)$상에서는 곱셈연산은 $a(x) * b(x)\ mod\ n(x)$를 계산한다. 몽고메리 알고리즘은 $GF(2^m)$상에서 아주 잘 적용되고, 특히 $r(x) = x^m$으로 두는 것은 아주 유용한 것으로 판명되었다[8]. 필드상의 한 원소인 r은 다항식 $r(x)\ mod\ n(x)$으로 표기되고, 만약 $n = (n^m n^{m-1}\ \cdots\ n^1 n^0)$이면 $r = (n^{m-1} n^{m-2}\ \cdots\ n^1 n^0)$이다. 몽고메리 곱셈 알고리즘에서 $r(x)$와 $n(x)$는 서로 소(relatively prime) 즉, $gcd(r(x), n(x)) = 1$이다. $r(x)$와 $n(x)$는 일반정수체계에서와 마찬가지로 식 (1)과 같이 $r(x) * r^{-1}(x) + n(x) * n'(x) = 1$의 성질을 갖는 $r^{-1}(x)$와 $n'(x)$가 존재한다. 다항식 $r^{-1}(x)$와 $n(x)$는 확장 유클리드 알고리즘(extended Enclidean algorithm)을 사용하여 구한다. C. K. Koc과 T. Acar가 제안한 $GF(2^m)$상의 a와 b의 몽고메리 곱셈은 $c(x) = a(x) * b(x) * rr^{-1}(x)\ mod\ n(x)$로 정의되고, 알고리즘은 다음과 같다[13].

MMM_GF(a(x), b(x), n(x), r(x))
1. n'(x) = n(x)^-1 mod r(x)
*2. t(x) := a(x) * b(x)*
*3. m(x) := t(x) * n'(x) mod r(x)*
*4. t(x) := (t(x) + m(x) * n(x)) div r(x)*

위의 알고리즘은 일반 정수체계에서의 알고리즘과 비교하여, 결과 $t(x)$의 차수가 $m - 1$보다 작기 때문에 일반 정수체계에서 $T \geq N$ 일 경우의 빼기 부분이 없다는 점이 다르다. 4번 줄에서 계산되는 $t(x)$의 차수는 다음과 같다.

$$\begin{aligned} deg(t(x)) &\leq max(deg(t(x)) + deg(m(x)) + deg(n(x))) - deg(r(x)) \\ &\leq max(2m-2\ ,\ m-1+m) - m \\ &\leq m - 1 \end{aligned}$$

2. 2 몽고메리 알고리즘 구현 방식

Koc 등은 몽고메리 알고리즘을 두 가지 요소를 통해 구별하였다[14]. 첫 번째는 곱셈 단계와 감소 단계의 위치에 따라 분리 접근 방식과 통합 접근 방식으로 나누었다. 분리 접근에서는 먼저 A, B 의 곱셈 후에 모듈러 감소를 수행한다. 통합 접근에서는 곱셈과 감소가 번갈아 수행된다. 두 번째 요소는 곱셈과 감소 단계의 일반적인 형태에 따라 피연산자(operand) 검색과 곱셈합(product) 검색으로 나누었다. 피연산자 검색은 루프의 인덱스의 증감이 피연산자의 인덱스에 따르는 것이다. 곱셈합 검색은 루프의 인덱스의 증감이 곱셈합의 인덱스에 따른다. 두 가지 요소를 통해 Koc 등은 일반 정수 체계에서 다음과 같은 다섯 가지 구현 형태로 나누었다.

- SOS(Separated Operand Scanning)
- CIOS(Coarsely Integrated Operand Scanning)
- FIOS(Finely Integrated Operand Scanning)
- FIPS(Finely Integrated Product Scanning)
- CIHS(Coarsely Integrated Hybrid Scanning)

2.3 사용되는 연산

$T \geq N$ 일 경우 한번의 뺄셈 연산이 필요하다. 다음의 모든 방식에서 같은 뺄셈 루틴을 가진다.

```
B = 0
for i = 0 to s-1
        (B, D) = T[i] - N[i] - B
        U[i] = D
```

$(B, D) = T[s] - B$
$U[s] = D$
if $B = 0$ *then return* U
else return T

위의 루틴에서 다음의 연산

$$(B, D) = T[i] - N[i] - B$$

은 B는 내림수(borrow)이고 D는 결과값이다. 즉 $T[i]$값이 $N[i]$값보다 적을 경우 $B = 1$이 되며, $D = T[i] + 2 * w - N[i]$가 된다. 이때 $2*w$는 내림수이다. 다음의 모든 알고리즘 구현 방식에서는 이 뺄셈 단계가 공통으로 사용되며, 각각의 성능 비교 시 $2s + 2$ 번의 덧셈, $2s + 2$ 메모리 읽기, 그리고 $s + 1$ 번의 메모리 쓰기의 횟수로 계산된다.

본 논문에서는 알고리즘의 비교 분석 시 루프 생성 시에 생기는 오버헤드와 인덱스 계산 시간은 고려하지 않았다. 그리고 N'값은 미리 계산하여 메모리에 저장하여 사용한다.

III. 몽고메리 알고리즘 구현

3.1 일반정수체계에서의 구현 방식 제안

3.1.1 SOS 방식의 향상

Koc 등이 제안한 SOS 방식은 곱셈 단계와 감소 단계가 완전히 분리된 두개의 루프를 통해 차례로 수행이 된다. 향상된 SOS 방식은 각각의 단계를 분석하여 덧셈의 횟수를 줄이고 CIOS방식에서의 인덱스 조정 방식을 이용하여 메모리의 읽기와 쓰기의 횟수를 줄였다. 먼저 곱셈 단계에서 루프의 시작 시 ($j = 0$) 올림수 C는 항상 0이므로 C값을 더할 필요가 없다. 따라서 다음과 같이 이 부분을 루프 밖으로 빼으로써 덧셈의 횟수를 줄일 수 있다(코드 2). 곱셈 단계의 구현은 다음과 같다.

```
1  for i = 0 to s-1
2    (C, S) = T[i] + B[0] * A[i]
3    T[i] = S
4    for j = 1 to s-1
5      (C, S) = T[i+j] + B[j] * A[i] + C
6      T[i+j] = S
7  T[i+s] = C
```

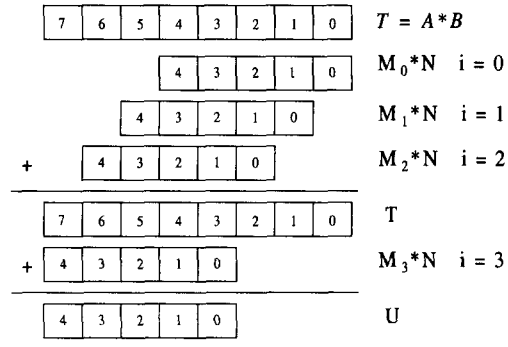

그림 1. 향상된 SOS 방식 (s = 4).
Fig 1 Improved SOS method (s = 4).

감소 단계에서도 $j = 0$ 일 경우 C값은 항상 0임으로 식을 루프 밖에서 계산함으로 덧셈의 횟수를 감소시킨다.(코드 10) 또한 $j = 0$일 때 계산되어 지는 S와 C의 값 중에서 C의 값은 다음 식에서 사용되어 지는 값이지만, S는 다음 루프에서 시프트(shift)되어 다시 사용되어지지 않으므로 저장할 필요가 없다.(코드 12) 시프트 연산 시 CIOS 방식과 마찬가지로 인덱스를 조정하여 수행할 수 있다. 그림 1과 같이 $i = s - 1$ 일 경우 저장되는 값을 $T[i + j]$에 저장하지 않고 $T[j - 1]$에 저장함으로써 SOS방식에서 곱셈과 감소단계후의 시프트 연산을 생략할 수 있다. (코드 15-22) 시프트 연산을 위해 따로 루프를 두지 않고 수행할 수 있게 되므로 메모리 읽기와 쓰기의 횟수를 줄일 수 있다. 감소 단계의 구현은 다음과 같다.

```
8   for i = 0 to s-2
9     m := T[i] * N'[0] mod W
10    (C, S) = T[i] + m * N[0]
11    for j = 1 to s - 1
12      (C, S) = T[i+j] + N[j] * m + C
13      T[i+j] = S
14      ADD(T[i + s], C)
15    m := T[s-1] * n'[0] mod W
16    (C, S) = T[s-1] + m * N[0]
17    for j = 1 to s - 1
18      (C, S) = T[s+j-1] + N[j] * m + C
19      U[j-1] = S
20    (C, S) = T[2*s-1] + C
21    U[s-1] = S
```

22 $U[s] = T[2*s] + C$

향상된 SOS방식에서는 $2s^2 + s$번의 곱셈, $4s^2 + 2s + 2$번의 덧셈, $6s^2 + 6s + 2$번의 읽기와 $2s^2 + 4s + 1$번의 쓰기 횟수를 가진다. SOS방식과 비교하여 보면 덧셈, 메모리 읽기와 메모리 쓰기에서 향상을 가져왔다. 다른 다섯 가지 방식들과 비교하여 보면 곱셈, 덧셈, 메모리 읽기와 쓰기에서 모두 가장 적은 횟수를 가진다. 그러나 시프트연산의 생략하기 위해 인덱스 조정방식을 사용함으로써 감소단계시 $i = s - 1$ 일 때 - 마지막 루프실행 시- 모듈라 연산($mod\ W$)을 수행하기 위해 루프를 한 번더 실행함으로써 실제 C 언어로 구현했을 경우 소스 코드의 길이가 크며, 덧셈, 메모리 읽기와 쓰기의 횟수를 줄이기 위해 중복되는 코드의 반복으로 컴파일한 후의 실행 파일의 크기도 크게 되어 실제적으로는 SOS 방식보다는 조금 빠른 수행을 가져왔으나, CIOS 보다는 조금 늦은 수행 시간이 걸렸다.

3.1.2 제안 방식 I

Koc 등이 제안한 FIOS방식과 CIOS방식은 유사하다. 차이점은 CIOS방식은 곱셈단계와 감소단계 사이에서 중간값 m을 계산하며, FIOS방식은 곱셈단계와 감소단계가 하나의 루프로 합쳐지면서 중간값 m 계산을 제일 먼저 수행한다. 그러나, FIOS방식의 알고리즘에서 곱셈 단계와 감소 단계가 하나의 루프에서 수행되어, 하나의 루프로 합쳐지면서 올림수의 발생이 빈번해지고 전달이 길어짐에 따라 실제 CIOS보다 덧셈과 읽기, 쓰기의 횟수가 늘어나게 되었다. 따라서, 중간값을 먼저 계산하며 하나의 루프로 합치지 않고, 곱셈단계와 감소단계를 각각의 루프에서 수행함으로써 덧셈과 메모리 읽기 횟수를 줄였으며, 그 형태는 CIOS와 FIOS의 중간 형태를 취한다. (그림 2)

몽고메리 알고리즘에서 중간값 m은 $T[0]$와 $n'[0]$값에 의해 결정이 된다. 이때 $n'[0]$값은 미리 계산되어 있는 값이며, $T[0] = T[0] + A[i] * B[0]$이다. 여기에서 $T[0]$을 계산할 때 사용되는 식은 곱셈단계에서의 식 $(C, S) = T[0] + A[i] * B[0] + C$에서 $j = 0$ 일 때 올림수 C가 항상 0임으로 $(C, S) = T[0] + A[i] * B[0]$로 같다. 따라서 이 부분을 한번만 계산하여 다음 곱셈단계에서 사용하여 중복을 피함으로써 덧셈과 읽기의 횟수를 줄였다.(표 1 코드 2) 또한 감소단계에서도 4.1의 SOS방식의 향상에서와 같이 인덱스 $j = 0$ 때 올림수가 0임을 이용하여 루프 밖에서 수행함으로 올림수를 더하는 것을 피함으로써 덧셈의 횟수를 줄였다. 또한 곱셈 단계에서 올림수를 전달할 때(표 1의 코드 10), 실제 $T[s+1]$에 있는 값은 전체 루프의 전 단계의 감소단계에서 시프트되어진 값으로(표 1의 코드 17) 가비지(garbage)값이 들어 있으므로 $T[s+1]$에 올림수를 더하지 않고 대입함으로써 덧셈의 횟수가 s번만큼 줄어든다.

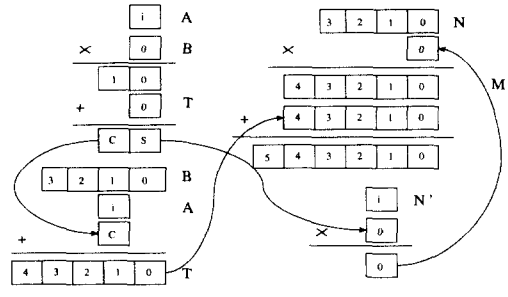

그림 2. 제안된 방식 I.
Fig 2 Proposed Method I

제안된 방식의 알고리즘과 분석은 표 1 과 같다. 제안된 알고리즘은 $2s^2 + s$번의 곱셈, $4s^2 + 2s + 2$번의 덧셈, $6s^2 + 5s + 2$번의 읽기, $2s^2 + 5s + 1$번의 쓰기 횟수를 가진다. Koc 등이 제안된 5가지 방식들과 비교하여 보면 곱셈, 덧셈, 읽기와 쓰기에서 가장 적은 횟수를 가진다. 특히, 덧셈과 읽기에서는 CIOS방식보다 더 적은 횟수를 가진다.

표 2. 제안 방식 I의 분석
Table 1. Analysis of Proposed Method I

코 드	muli	add	read	write
1. for i = 0 to s-1				
2. (C, S) = T[0] + B[0]*A[i]	s	s	3s	
3. m = S*n'[0] mod W	s		s	s
4. T[0] = S				s
5. for i = 1 to s-1				
6. (C, S) = T[j] + B[j]* A[i] + C	s(s-1)	2s(s-1)	3s(s-1)	
7. T[i] = S				s(s-1)
8. (C,S) = T[s] + C		s	s	
9. T[s] = S				s
10. T[s+1] = C				s
11. (C, S) = T[0] + m * N[0]	s	s	3s	
12. for i = 1 to s-1				
13. (C, S) = T[j] + N[j]*m + C	s(s-1)	2s(s-1)	3s(s-1)	
14. T[i-1] = S				s(s-1)
15. (C,S) = T[s] + C		s	s	
16. T[s-1] = S				s
17. T[s] = T[s+1] + C				s
18. 마지막 계산(T - N)		2(s+1)	2(s+1)	s+1
합 계	$2s^2+s$	$4s^2+2s+2$	$6s^2+5s+2$	$2s^2+5s+1$

3.1.3 제안 방식 II

위의 제안된 구현 방식에서 C 언어와 같은 고급 프로그래밍 언어에서 구현 할 경우 효과적으로 구현할 수 있다. 연산 $T[i] + A[i] * B[j] + C$ 에서 결과값의 범위는 2개의 워드 크기를 넘지 않으므로 실제 C언어로 구현 시 C, S는 변수로 선언하여 다음과 같이 구현된다. (한 워드의 크기가 4비트인 경우)

$$S = (T[i] + A[i]*B[j] + C) \ \&\ 0Fh$$
$$C = ((T[i] + A[i]*B[j] + C) \gg 4) \ \&\ 0Fh$$

이때 C의 선언을 하나의 워드 크기보다 큰 범위를 갖은 변수로 선언하여 사용하여 연산 $T[i] + A[i] * B[j] + C$ 에 값을 하나 더 더할 수 있게 한다. 다시 말해 연산 $T[i] + A[i] * B[j] + C + M[i] * N[j]$ 와 같이 결과값이 최대 3개의 워드를 가질 수 있다. 따라서 C의 범위를 2워드 크기로 하여 다음과 같이 구현하면 올림수의 전달이 2번 이상이 일어날 경우 한번의 덧셈으로 수행할 수 있다.

$$S = (T[i] + A[i]*B[j] + M[i]*N[j] + C) \ \&\ 0Fh$$
$$C = ((T[i] + A[i]*B[j] + M[i]*N[j] + C) \gg 4) \ \&\ FFh$$

예를 들어 $T[i] + A[i]*B[j] + C$의 값이 $1FF$ 일 경우 $S = F$가 되고 C는 한 워드 값을 가지므로 $1F$의 10가 실제적으로 올림수가 두 워드를 거쳐 전달되어야 하지만 C를 두 워드 크기로 선언하게 되면 올림수의 전달이 한 워드만을 거쳐 전달하게 된다. 이러한 고급 언어의 이점을 이용하여 제안 방식 I에서 곱셈 단계와 감소 단계를 하나의 루프로 합쳤다. (그림 3)

또한 마지막에 전달된 올림수가 2워드일 경우 올림수의 전달이 한번 더 필요함으로 이를 피하기 위해 피연산자(B), 모듈러스(N)의 최상위워드에 0을 하나 추가하여 루프를 한번 더 수행하게 되면 $j = s$ 일 때 $B[j]$, $N[j]$ 값은 0이 되어 표 2의 6번 줄에서 단순히 올림수의 전달만이 있을 뿐이다. 따라서 표 2의 8 - 10번 줄에서 올림수가 한 번만 일어난다.

제안 방식 II와 분석은 표 2와 같다. 제안 방식

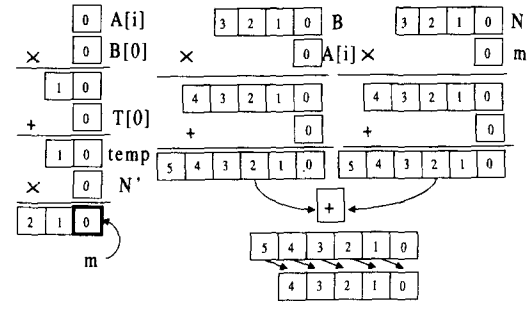

그림 3. 제안 방식 II
Fig 3 Proposed Method II

II은 $2s^2 + 3s$번의 곱셈, $3s^2 + 5s + 2$번의 덧셈, $5s^2 + 11s + 2$번의 읽기와 $s^2 + 5s + 1$번의 쓰기 횟수를 가진다. 안쪽 루프에서 한번 더 수행하므로 곱셈에서 다른 방식들보다 많은 수행을 하지만 올림수의 전달 길이가 짧아지므로 덧셈과 읽기, 쓰기에서는 가장 적은 횟수의 수행을 한다.

표 2의 6번 줄에서 오른쪽 계산 결과의 범위는 한 워드가 k 비트일 경우 $2k + 3$ 비트보다 작거나 같다. 예를 들어 $k = 16$ 비트 일 경우 결과값은 35비트이다. 32 비트 컴퓨터 기계어 일 경우 16 비트의 곱의 결과값은 32비트이다. 이때 상위 16비트는 DX 레지스터에 하위 16비트는 AX 레지스터에 입력이 된다. 따라서 다음과 같이 DX 레지스터의 값을 사용하지 않는 확장 32비트 레지스터에 저장하면 된다. 리스트 2에서 레지스터끼리의 읽기와 쓰기를 통해 수행되므로 실제로 메모리에서 읽고 쓰는 것보다 빠르다. 따라서, CIOS나 4.2의 제안 알고리즘보다 훨씬 빠른 수행을 한다.

표 2. 제안된 알고리즘 구현 방식 II.
Table 2 Analysis of the Proposed Method II

코 드	곱셈	덧셈	읽기	쓰기
1. for i = 0 to s-1				
2. temp = T[0] + B[0]*A[i]	s	s	3s	s
3. m = temp*n'[0] mod W	s		2s	s
4. (C, S) = temp + m*N[0]	s	s	3s	
5. for j = 1 to s				
6. (C, S) = T[j] + B[j]*A[i] + m*N[j] + C	$2s^2$	$3s^2$	$5s^2$	
7. T[j-1] = S				s^2
8. (C,S) = T[s-1] + C		s	s	
9. T[s-1] = S				s
10. T[s] = C				s
11 마지막 빼기(T - N)		2(s+1)	2(s+1)	s+1
합 계	$2s^2+3s$	$3s^2+5s+2$	$5s^2+11s+2$	s^2+5s+1

3.2 GF(2^m)상에서의 몽고메리 알고리즘의 구현

3.2.1 Koc 등이 제안한 방식의 구현

GF(2^m)상에서의 몽고메리 알고리즘 구현은 일반 정수 체계에서와 같이 다음과 같은 워드 단위로 계산이 수행이 된다.

$$a(x) = \sum_{i=0}^{s-1} A_i(x)x^{iw}$$

$$= A_i(x)x^{(s-1)w} + A_{s-2}(x)x^{(s-2)w} + \cdots + A_1(x)x^w + A_0(x)$$

w는 워드의 크기를 나타내며, s는 총 워드의 개수에 해당하며, $a(x)$를 총 k비트라고 하면 $k = s * w$가 된다.

GF(2^m)상에서의 몽고메리 알고리즘의 주된 연산은 GF(2)상에서의 곱셈(Mul_GF)과 덧셈인 XOR 연산이다. GF(2^m)상에서의 곱셈은 다음과 같은 함수로 구현된다.

```
Mul_GF(A, B)
{
  for ( i = 1; i < w +1; i++)
    if ((B)>>w-i)&0x0001) c = (c << 1) ^ A;
    else c = c << 1;
    H = (c >> w) & (2w-1);
    L = c & (2w-1);
}
```

A, B는 한 워드 값이며, 결과는 c에 임시로 저장되어 다시 상위 w비트는 H, 하위 w비트는 L에 나뉘어 저장된다. 다음의 방식의 비교 분석에서는 H, L에 대한 메모리 접근 횟수는 고려하지 않았다. 위의 함수는 다음에 구현되는 모든 방식에 적용된다.

GF(2^m)상에서 몽고메리 알고리즘 구현 시 일반 정수체계와 다른 점은 곱셈이나 덧셈의 경우 올림수의 전달이 하나를 넘지 않는다는 장점이 있다. 일반 정수의 경우 w비트의 두개의 값을 더하면 $w+1$비트의 크기가 되지만 GF(2^m)상에서는 비트 단위로 XOR연산을 수행하므로 w비트의 크기가 되며, 곱할 경우 일반 정수의 경우는 $2w$비트의 크기가 되지만 GF(2^m)상에서는 마지막의 덧셈에서 올림수가 발생하지 않으므로 $2w-1$크기가 된다. 따라서 올림수의 계산은 앞자리 하나만을 고려하면 된다. 주의할 점은 일반 정수 체계에서는 N의 값의 크기가 s개의 워드를 넘지 않았지만 GF(2^m)상에서는 $n(x)$가 $s + 1$개의 워드 크기를 가지므로 감소 단계에서 $n(x)$의 MSB에 해당하는 값의 덧셈을 수행하여야 한다.

3.2.2 제안 방식의 구현

제안 방식 I로 구현된 GF(2^m)상의 몽고메리 알고리즘은 다음과 같다. 제안 방식 I는 $2s^2 + s$번의 GF(2^m)상에서의 곱셈, $4s^2 + 2s$번의 XOR, $8s^2 + 3s$번의 메모리 읽기와 $4s^2 + 2s$번의 메모리 쓰기 횟수를 보였다.

```
void VER1_MMM_GF(N')
{
  for (i=0;i<s;i++)
  {
    T[0] = T[0] ^ Mul_GF(A[i], B[0]);
    M = Mul_GF(T[0],-N');
    for (j = 1; j< s; j++)
    {
      T[j] = T[j] ^ H;
      T[j] = T[j] ^ Mul_GF(A[i], B[j]);
    }
    T[s] = T[s] ^ H;
    T[0] = T[0] ^ Mul_GF(M, N[0]);
    for (j = 1; j< s; j++)
    {
      T[j] = T[j] ^ H;
      T[j-1] = T[j] ^ Mul_GF(M, N[j]);
    }
    T[s-1] = T[s] ^ H ^ M;
    T[s] = 0;
  }
}
```

제안 방식 II로 구현된 GF(2^m)상의 몽고메리 알고리즘은 다음과 같다. 제안 방식 II의 구현 시 올림수의 전달이 하나를 넘지 않으므로 안쪽 루프를 한 번 더 돌 필요가 없으므로 $s - 1$번만 루프를 돈다. 또한 올림수 계산 시 P라는 임시 메모리에 두어 두 GF(2^m)상에서의 곱셈에서 발생하는 올림수를 저장하였다. 제안 방식 II는 $2s^2 + s$ 번의 GF(2^m)상에서의 곱셈, $4s^2$ 번의 Xor, $8s^2$-s번의 메모리 읽

기와 $4s^2$ 번의 메모리 쓰기 횟수를 보였다.

```
void VER2_MMM_GF(N')
{
  for (i=0;i< s+1;i++) T[i] = 0;
  for (i=0;i<s;i++)
  {
    T[0] = T[0] ^ Mul_GF(A[i], B[0]);
    P = H;
    M = Mul_GF(T[0], N');
    Mul_GF(M, N[0]);
    H = H ^ P;
    for (j = 1; j< s; j++)
    {
      T[j] = T[j] ^ H;
      T[j] = T[j] ^ Mul_GF(A[i], B[j]);
      T[j+1] = T[j+1] ^ H;
      T[j-1] = T[j] ^ Mul_GF(M, N[j]);
    }
    T[s-1] = T[s] ^ H ^ M;
    T[s] = 0;
  }
}
```

IV. 비교 분석

향상된 SOS 방식의 분석은 표 3과 같다

표 3. 향상된 SOS 방식의 분석.
table 3. Analysis of Improved SOS method

코 드	곱셈	덧셈	읽기	쓰기
1. for i = 0 to s-1				
2. (C, S) = T[i] + B[0]*A[i]	s	s	3s	
3. T[i] = S				s
4. for j = 1 to s-1				
5. (C, S) = T[i+j]+B[j]*A[i] + C	s(s-1)	2s(s-1)	3s(s-1)	
6. T[i+j] = S				s(s-1)
7. T[i + s] = C				s
8. for i = 0 to s-2				
9. m := T[i] * n'[0] mod W	s-1		2(s-1)	s-1
10 (C, S) = T[i] + m * N[0]	s-1	s-1	3(s-1)	
11 for j = 1 to s - 1				
12 (C, S) = T[i+j]+N[j]*m + C	$(s-1)^2$	$2(s-1)^2$	$3(s-1)^2$	
13 T[i+j] = S				$(s-1)^2$
14 ADD(T[i + s], C)		2(s-1)	2(s-1)	2(s-1)
15 m := T[s-1] * n'[0] mod W	1		2	1
16 (C, S) = T[s-1] + m * N[0]	1	1	3	
17 for j = 1 to s - 1				
18 (C, S) = T[s+j-1]+N[j]*m + C	s-1	2(s-1)	3(s-1)	
19 U[j-1] = S				s-1
20 (C, S) = T[2*s-1] + C		1	1	
21 U[s-1] = S				1
22 U[s] = T[2*s] + C		1	1	1
23 마지막 빼기(T - N)		2(s+1)	2(s+1)	s+1
합 계	$2s^2+s$	$4s^2+2s+2$	$6s^2+6s+2$	$2s^2+4s+1$

다음 표 4는 Koc 등이 제시한 5가지 알고리즘과 제안된 방식들을 곱셈, 덧셈, 메모리 읽기와 쓰기의 횟수에 대하여 비교한다.

표 4. 여러 방식들의 시간과 공간적 크기 비교
table 4. Comparison of serveral methods with time and space

방식	곱셈	덧셈	읽기	쓰기
SOS	$2s^2+s$	$4s^2+4s+2$	$6s^2+7s+3$	$2s^2+6s+2$
CIOS	$2s^2+s$	$4s^2+4s+2$	$6s^2+7s+2$	$2s^2+5s+1$
FIOS	$2s^2+s$	$5s^2+3s+2$	$7s^2+5s+2$	$3s^2+4s+1$
FIPS	$2s^2+s$	$6s^2+2s+2$	$9s^2+8s+2$	$5s^2+8s+1$
CIHS	$2s^2+s$	$4s^2+4s+2$	$6.5s^2+6.5s+2$	$3s^2+5s+1$
Improved SOS	$2s^2+s$	$4s^2+2s+2$	$6s^2+6s+2$	$2s^2+4s+1$
제안방식 I	$2s^2+s$	$4s^2+2s+2$	$6s^2+5s+2$	$2s^2+5s+1$
제안방식 II	$2s^2+3s$	$3s^2+5s+2$	$5s^2+11s+2$	s^2+5s+1

곱셈에서는 제안 알고리즘 II를 제외한 모든 알고리즘이 $2s^2 + s$ 번의 수행을 가진다. 곱셈 단계와 감소 단계에서 각각 s^2번과 중간값 m을 계산할 때 s번씩 이루어진다. 제안 알고리즘 II에서는 안의 루프가 한번 더 수행함으로 $2s$번이 많아졌다. 덧셈, 읽기와 쓰기에서는 제안된 알고리즘 II이 각각 $3s^2 + 5s + 2$, $5s^2 + 11s + 2$와 $s^2 + 5s + 1$로 하한을 가진다. 이는 곱셈 단계와 감소 단계에서 루프의 인덱스 값이 0일 때 항상 올림수의 값이 0임으로 인덱스 값이 0일 경우에는 올림수를 더하는 것을 피함으로써 수행 횟수를 줄였다. 또한 고급 프로그램 언어의 타입 선언을 이용하고, 피연산자와 모듈러스의 MSB에 0을 더하여, 곱셈 단계와 감소 단계에서 루프를 한 번 더 수행함으로써 올림수의 전달을 줄임으로써 올림수의 덧셈과, T값의 읽기와 쓰기 횟수를 줄였다.

이 알고리즘들의 실제 수행을 측정하기 위하여 C 언어를 이용하여 Intel 펜티엄프로-200MHz CPU, RAM 64Mbyte PC에서 구현하였다. 한 자리의 크기가 각각 1, 4, 6 비트 즉 기저로 2, 16, 64진수를 가지는 512, 1024 비트를 크기의 메시지에 대하여 실험을 하였으며 그 결과는 표 5에 나타나 있다. 표 5에서의 시간 단위는 밀리초이며, 각 값은 10000번 수행의 평균값이다. 표 5에서 보듯이 제안 알고리즘 II가 가장 빠른 시간을 보여준다.

표 5. C로 구현된 방식들의 실제수행시간 (단위 =ms).
table 5. Excution time of methods in C-language(unit=ms)

메시지길이	512 bits			1024 bits		
워드 크기	1bit	4bits	5bits	4bits	8bits	16bits
SOS	1.386	0.088	0.056	0.352	0.088	0.047
CIOS	1.312	0.083	0.054	0.329	0.082	0.046
FIOS	2.252	0.120	0.078	0.479	0.120	0.065
FIPS	2.618	0.164	0.107	0.655	0.164	0.088
CIHS	1.627	0.103	0.067	0.408	0.103	0.057
Improved SOS	1.400	0.087	0.056	0.348	0.087	0.047
제안방식I	1.299	0.081	0.053	0.325	0.081	0.045
제안방식II	0.975	0.061	0.040	0.242	0.061	0.025

다음 표 6은 각각의 방식들을 어셈블리 언어를 이용하여 구현하여 실제 수행 시간을 측정한 것이다. 이때 사용된 어셈블리 코드는 인텔 펜티엄 코드를 이용하여 작성되었으며, 한 워드의 크기가 16 비트이다. 시간 측정은 메인 함수는 C 언어로 작성되어 extern 함수로 선언하여 호출하여 함수 호출 시간이 포함된 시간이다.

표 6. 어셈블리 언어로 구현된 방식들의 수행 시간(단위 = ms)
table 6. Execution time fo methods in assembly language (unit = ms)

	512비트	1024 비트	1536비트
SOS	0.008	0.019	0.040
CIOS	0.007	0.016	0.039
FIOS	0.010	0.025	0.059
FIPS	0.012	0.031	0.068
CIHS	0.009	0.022	0.043
Improved SOS	0.008	0.018	0.040
제안방식 I	0.007	0.016	0.038
제안방식 II	0.005	0.012	0.024

다음 표 7은 GF(2^m)상에서 몽고메리 알고리즘을 각각의 방식에 대하여 C 언어로 구현한 후, 위의 표 5에서와 같은 환경에서 수행하여 측정한 시간을 나타낸다. 결과를 살펴보면, 한 워드의 크기 $w = 1$ 비트일 경우에는 일반 정수 체계에서는 빠르지만 한 워드의 크기가 커질 수록 수행 시간은 더 길어짐을 알 수 있다. 이는 GF(2^m)상에서 곱셈 함수 *Mul_GF()*를 살펴보면, 실제 시프트되는 단위가 1 비트임으로 한 워드의 크기가 커지면 *Mul_Gf()*의 수행 시간이 길어짐으로 같은 크기의 워드일 경우 모듈러 곱셈의 전체 수행 시간이 길어지게 된다. 따라서 실제적으로 효과적인 GF(2^m)상에서 곱셈 방법의 구현이 필요하게 된다.

표 7. GF(2^m) 상에서 C-언어 구현된 방식들의 실제 수행 시간 (단위 = ms)
table 7. Execution time of method in C on GF(2^m) (unit = ms)

메시지길이	512 bits			1024 bits		
워드 크기	1bit	4bits	5bits	4bits	8bits	16bits
SOS	0.962	0.095	0.082	0.379	0.161	0.065
CIOS	0.912	0.091	0.080	0.366	0.160	0.064
FIOS	0.892	0.090	0.079	0.359	0.158	0.064
FIPS	0.853	0.086	0.076	0.342	0.153	0.063
CIHS	0.911	0.091	0.081	0.366	0.161	0.064
Improved SOS	0.959	0.095	0.084	0.380	0.164	0.064
제안방식I	0.898	0.091	0.080	0.365	0.159	0.063
제안방식II	0.891	0.090	0.078	0.359	0.153	0.062

각 방식의 수행 시간을 비교해 보면, FIPS 방식이 훨씬 빠르며, 이는 FIPS 방식을 살펴보면 GF(2^m)상의 모듈러 곱셈의 구현 시 실제 중간 계산값을 두 개의 임시 저장소만을 이용하므로 덧셈 연산을 올림수의 전달이 거의 없는 XOR연산을 사용하므로 임시 저장소가 컴파일 시 레지스터에 저장되어 읽기와 쓰기가 수행되므로 수행 시간이 다른 방식들에 비해 빠르다. 그러나 한 워드의 크기가 커짐에 따라 제안 방식 I와 II의 수행 시간이 FIPS 방식과 거의 비슷하며, 제안 방식 II의 경우 수치가 같다. 이는 제안 방식이 메모리의 접근 횟수가 다른 방식에 비해 적으므로 워드의 크기가 커짐에 따라 빠른 수행을 한다.

V. 결 론

공개키 암호 시스템은 안전성을 위해 512비트 이상의 큰 수에 대하여 모듈러 지수승을 수행함으로써 처리 속도의 지연이라는 단점이 있다. 본 논문에서는 모듈러 지수 연산에서 사용되는 모듈러 곱셈 알고리즘들 중 간단하고 보다 효과적인 알고리즘인 몽고메리 알고리즘을 변형하여 빠른 수행의 몽고메리 알고리즘을 구현하였다. 제안된 알고리즘 구현 방식에서 곱셈의 하한값은 $2s^2 + s$이며 제안된 방식 II를 제외하고는 모든 방식이 하한값을 가졌다. 덧셈은 $3s^2 + 5s + 2$, 메모리 읽기는 $5s^2 + 9s + 2$, 메모리 쓰기는 $s^2 + 4s + 1$로 제안된 방식 II가 가장 효율적이고 빠른 수행을 한다. 제안된 방식 II은 피연산자(B), 모듈러스(N)의 MSB에 0을 하

나 더 추가하여 루프를 한번 더 돌림으로써 올림수의 발생으로 인한 메모리 읽기와 쓰기의 횟수를 줄였으나 이로 인해 곱셈의 횟수가 $2s$만큼 많아지게 되었다. 그러나, 이는 곱셈의 수행시간과 메모리 읽기, 시간을 비교해 보았을 때 곱셈의 수행시간이 훨씬 적으므로 실제적으로는 더 빠른 수행을 가져왔다. 또한 $GF(2^m)$상에서도 몽고메리 알고리즘을 제안된 방식을 이용하여 구현하였다. 전체적으로 일반 정수 체계에서 보다는 느린 수행을 가져왔으며 워드의 크기가 클 경우 제안 방식 II이 효과적인 수행을 하였다.

공개키 암호 시스템에서 쓰이는 모듈러 지수 연산은 아주 큰 수에 대하여 모듈러 곱셈을 반복적으로 수행된다. 수행 속도를 높이기 위해 하드웨어 구현이 대부분이지만 빨라지는 일반용 컴퓨터의 프로세서의 발달로 소프트웨어로의 구현도 할 수 있게 되었다. 그러므로 가장 빠른 모듈러 곱셈 알고리즘인 몽고메리 알고리즘을 분석하여 보다 빠른 수행을 위한 구현 방식을 통해 처리 시간을 줄일 뿐 아니라, 스마트 카드와 같은 마이크로 프로세스에서의 구현에도 사용 할 수 있다

참 고 문 헌

[1] W. Diffie, M. Hellman, "New Directions in Crytography," *IEEE Trans. on Info. Theory*, vol. IT-22(6) pp.644-654, 1976

[2] Rivest, R. L., Shamir, A. and Adleman, L., " A Method for Obtaining Digital Signatures and Public-key Cryptosystems," *Communication of the ACM*, 21, 120-126. 1978

[3] T. Elgamal, " A Public-Key Cryptosystem and Signature Scheme Based on Discrete Logarithms," *IEEE Trans. Information Theory*, vol. IT-31, No. 4, 1985, pp. 469-472

[4] D. Barret, "Implementing the Rivest Shamir and Adleman Pulblic Key Encryption Algorithm on a Standard Digital Signal Processor," *Advances in Cryptology-CRYPTO* '86, A. M. Odlyzko, ed., Lecture Notes in Computer Science 263, Springer-Verlag, 1987, pp. 311 - 323

[5] E, Eldridge, "A Faster Modular Multiplication Algorithm," *Intern. J. comput. Math.*, vol. 40, 1991, pp. 63-68,

[6] E. Eldridge and C. D. Walter, "Hardware Implementation of Montgomery's Modular Multiplication Algorithm,", *IEEE Trans on Computers*, vol. 42, No. 6, Jane 1993, pp. 693-699

[7] E. F. Brickell, "A Fast Modular Multiplication Algorithm with Application to Two-Key Cryptography," *Advances in Cryptology-CRYPTO* '82, Chaum et al., eds., New York:Plenum 1983, pp. 51-60

[8] Hui, L. C. K. and Lam, K. Y., "Fast Square-and-Multiply *Exponentiation for RSA*," *Electronics Letter*, 30(17), pp. 1396-1397, 1994.

[9] Antoon Bosselaers, Rene Govaerts, and Joos Vandewalle, "Comparison of Three Modular Reduction Functions," *Advances in Cryptology-CRYPTO* '93, pp. 175-186, August, 1993.

[10] P. L. Montgomery, "Modular Multiplicatio Without Trial Division," *Mathemat. of Computat*, vol. 44, pp. 519-521, 1985

[11] R, J. McEliece, *Finate Field for Computer Scientists and Engineers*, Kluwer Academic Publishers, 1987.

[12] R. Lidl and H. Niederreiter, *Introduction to Finite Fields and Their Application*, Cambridge University Press, Cambridge, UK, 1994.

[13] Koc, C. K. and Acar, T. "Montgomery Multiplication in GF(2k)," *In Proceedings of Third Workshop on Selected Area in Cryptography*, pp. 95-106, queens University, Kinston, Ontario, Canada, August 15-16 1996.

[14] Koc, C. K., Acar, T. and Kaliski, jr. B. S., "Analyzing and Comparing Montgomery Multiplication Algorithms," *IEEE Micro*, pp. 26-33, June, 1996.

□ 著者紹介

안 준 언(Joon-Eun Ahn) **정회원**

1997년 2월 : 경북대학교 컴퓨터공학과 졸업
1997년 3월 : 경북대학교 컴퓨터공학과 석사
1999년 2월 : 경북대학교 컴퓨터공학 석사졸업

유 기 영(Kee-Young Yoo) **정회원**

1976년 2월 : 경북대학교 수학교육학과 졸업(이학사)
1978년 2월 : 한국 과학기술원 전산학과졸업(공학석사)
1992년 2월 : 미국 Renss elaer Polytechnic Institute 졸업 (이학박사)
1978년~현재: 경북대학교 컴퓨터공학과에 재직.
관심분야 : 병렬처리, DSP array processor설계, 병렬 컴파일러, 암호학 등임.