論文 97-22-5-17

Montgomery 모듈라 곱셈을 변형한 고속 멱승

正會員 河 在 喆*, 文 相 在*

Fast Exponentiation with Modifed Montgomery Modular Multiplication

Jae-Cheol Ha*, Sang-Jae Moon* *Regular Members*

※본 연구는 정보통신연구관리단의 국책기술개발 사업 지원에 의해서 이루어졌습니다.

요 약

본 논문에서는 동일한 피승수를 가지는 두번의 모듈라 곱셈시 공통 계산 부분이 생기도록 Montgomery 곱셈을 변형하여 공통 부분을 한번만 계산할 수 있게 하므로써 멱승 계산속도를 개선하였고, 또한 m-ary나 window 멱승 방식에서는 저장용 테이블을 줄일 수 있게 하였다. 또한 지수 folding 기법을 사용한 멱승 방식에 적용할 경우에는 고속이면서 사용 메모리가 적어 IC카드와 같은 저메모리 환경에 효과적이다.

ABSTRACT

We modify the Montgomery modular multiplication to extract the common parts in common-multiplicand multiplications. Since the modified method computes the common parts in two modular multiplications once rather than twice, it can speed up the exponentiations and reduce the amount of storage tables in m-ary or window exponentiation. It can be also applied to an exponentiation method by folding the exponent in half. This method is well-suited to the memory limited environments such as IC card due to its speed and requirement of small memory.

Ⅰ. 서 론

소인수분해 문제(prime factorization problem)나 이산대수 문제(discrete logarithm problem)에 근거한 공개 키 암호시스템[1, 2]에서는 유한체 GF(p) 혹은 유한환 Z_N상에서 큰 수에 대한 모듈라 멱승 연산이 필요하다. 그러나 멱승 연산은 시간이 많이 소요되므로 공개 키 암호 시스템의 중요한 단점이 될 수 있다. 그러므로 공개 키 암호 시스템에서 멱승 시간을 줄이는 연구가 필요하다.

*경북대학교 전자전기공학부
論文番號:97086-0304
接受日字:1997年 3月 4日

일반적으로 모듈라 멱승은 $A^E \bmod N$으로 표시할 수 있으며 A와 N은 안전도에 따라 가변적이나 512비트 이상의 큰 정수를 사용한다. 지수 E는 RSA 시스템[1]이나 ElGamal 시스템[2]에서는 512비트 이상, 이산대수 문제에 근거한 디지탈 서명 시스템[3, 4]에서는 140비트 이상을 사용한다. 멱승 연산은 모듈라 곱셈의 반복으로 이루어져 있으므로 고속 멱승을 위해서는 모듈라 곱셈 횟수가 적은 멱승 방식을 선택하거나 모듈라 곱셈의 고속화가 필요하다. 멱승 방식으로는 이진 방식, m-ary 방식, window 방식 등이 대표적이다[5]. 모듈라 곱셈은 곱셈과 모듈라 감소 연산으로 나눌 수 있으며 모듈라 감소 방법으로는 고전적인 알고리듬[5, 6], Barrett 알고리듬[7] 그리고 Montgomery 알고리듬[8] 등이 있다.

본 논문에서는 멱승 연산 중 left-to-right 형태는 동일한 피승수에 대한 두번의 모듈라 곱셈을 수행하는 것에 기반하여 두번의 곱셈시 공통 계산 부분이 생기도록 Montgomery 모듈라 곱셈을 변형하였다. 공통 계산 부분은 한번만 계산할 수 있으므로 변형한 방법을 멱승 방식에 적용하면 계산속도를 개선할 수 있고 사용 메모리도 줄일 수 있다. 본 논문의 제 2장에서 Montgomery 모듈라 곱셈을 변형한 곱셈 방법을 기술하고 제 3장에서 이를 여러 멱승 방식에 적용한다. 제 4장에서는 기존 Montgomery 모듈라 곱셈과 변형한 방법을 멱승에 적용했을 경우의 계산 속도를 비교 분석하고 C 언어로 구현한 결과를 제시한다.

Ⅱ. Montgomery 모듈라 곱셈의 변형

모듈라 곱셈은 두 수의 곱셈과 그 결과에 대한 모듈라 감소로 이루어지는 데 이 두 과정을 순차적으로 수행할 수도 있고 곱셈을 하면서 모듈라 감소를 수행할 수도 있다. 멱승 연산에 사용되는 모듈라 감소 방법 중 현재까지 Montgomery 알고리듬이 속도면에서 가장 효과적인 것으로 알려져 있다[6]. 본 장에서는 Montgomery 모듈라 곱셈을 기술하고 고속 멱승을 위해 이를 변형한다.

2.1 Montgomery 모듈라 곱셈

임의의 큰 수 X를 기수(radix) b로 표현하면 $X=\sum_{i=0}^{n-1} X_i b^i$와 같이 n자리로 나타낼 수 있다. 여기서 $X_i \in \{0, 1 \cdots b-1\}$이다. n자리인 두 수 A와 B의 모듈라 곱셈 $A\times B \bmod N$은 A와 B를 곱하는 곱셈과 이 결과를 N에 대해 모듈라 감소하는 연산으로 이루어진다. n자리인 두 수의 곱셈은 n^2번의 기수 b보다 작은 두 수의 곱셈으로 수행될 수 있다.(이하에서는 기수 b보다 작은 두 수의 곱셈을 작은 수 곱셈이라 한다.) 모듈라 감소는 A와 B를 곱한 결과 C에 대해 $C-QN$을 구하는 것이다. 여기서 Q는 C를 N으로 나눈 몫이다. 그러므로 고전적인 모듈라 감소에서는 큰 수에 대한 나눗셈이 필요하며 이 연산은 덧셈이나 곱셈에 비해 구현하기가 어렵다.

Montgomery 알고리듬에서는 먼저 N보다 크고 N과 서로 소인 R을 선택하는데 일반적으로 R에 대한 모듈라 연산(mod R)이나 나눗셈(div R)을 자리수 이동으로 간단히 계산할 수 있도록 b^n으로 한다. 이 알고리듬은 나눗셈을 하는 고전적인 방법과는 달리 임의의 정수 X의 R에 대한 N-residues를 $XR \bmod N$으로 정의하고 이 집합내에서 빠른 모듈라 감소를 수행한다. Montgomery 모듈라 감소 알고리듬은 N-residue 상에서 $0\le T\le RN$인 임의의 정수 T에 대해 $TR^{-1} \bmod N$을 계산하는 것과 같다. 여기서 T는 N-residues 상의 A와 B를 곱한 결과로 볼 수 있다. 이 알고리듬의 핵심은 T에 대해 N의 일정한 배수를 더하여 하위 n자리가 0이 되도록 하는 것이다. 하위 n자리가 0인 수를 R로 나누면 $TR^{-1} \bmod N$가 된다.

한편, Dusse와 Kaliski는 Montgomery 알고리듬을 개선하여 N-residue로 변환된 A와 B를 곱하면서 모듈라 감소를 수행하는 모듈라 곱셈 방법을 제안하였다[9]. (이하에서는 이를 Montgomery 곱셈이라 한다.) Montgomery 곱셈은 $REDC(A\times B)=ABR^{-1} \bmod N$으로 표현할 수 있으며 이 알고리듬을 나타낸 것이 그림 1이다. 여기서 N_0'은 $-N_0^{-1} \bmod b$이며 Euclid의 최대공약수 알고리듬[5]으로 사전에 계산할 수 있다.

Montgomery 곱셈을 사용하여 한번의 모듈라 곱셈을 수행할 때에는 두 수를 N-residues로 변환한 후 변환된 두 수에 대한 Montgomery 곱셈을 수행한다. 이 결과 값은 N-residue 상의 수이므로 일반수로 변환하는 사후 계산과정이 필요하다. 그러므로 Montgomery 곱셈은 residue 변환에 따른 부가연산 때문에 모듈라

```
Montgomery Multiplication : C=REDC(A×B)
    C=0;
    for i = 0 to n-1 step 1 {
        C=C + B×A_i;
        m=C_0×N'_0 mod b;
        C=(C + N×m)/b; }
    if(C≥N) C = C - N ;
    return(C);
```

그림 1. Montgomery 모듈라 곱셈
Fig. 1 Montgomery modular multiplication.

곱셈의 반복하는 횟수가 적거나 모듈라 수가 자주 변할 때는 비효율적이다. 그러나 멱승시에는 N-residues 상에서 모듈라 곱셈이 반복되므로 매우 효과적이다. Montgomery 곱셈은 $2n^2+n$번의 작은 수 곱셈이 필요하며 고전적인 방법이나 Barrett 알고리듬에 비해 고속이면서 구현이 용이하다[6].

2.2 Montgomery 모듈라 곱셈의 변형

이진 멱승 방식으로 A^E 연산시 먼저 k 비트인 E를 이진수로 변환한다. 즉, $E=\sum_{i=0}^{k-1} e_i 2^i$, $e_i \in \{0, 1\}$로 표현할 수 있다. 이진 방식은 지수를 상하위 비트 중 어느 쪽에서부터 탐색하는가에 따라 left-to-right 방식과 right-to-left 방식으로 구분할 수 있는데 Montgomery 곱셈을 사용한 right-to-left 이진 멱승 방식을 나타낸 것이 그림 2이다.

```
Right-to-left binary exponentiation : C=A^E mod N
    C= R mod N;
    S= AR mod N;
    for i=0  to k-1 step 1
        if( e_i = 1) then { C=REDC(S×C); S=REDC(S×S);}
        else S=REDC(S×S);
    REDC(C×1);
    return(C);
```

그림 2. Right-to-left 이진 멱승 방식
Fig. 2 Right-to-left binary exponentiation.

그림 2에서 보는 바와 같이 e_i가 1일 때에는 동일한 S에 대해 C=REDC(S×C)과 S=REDC(S×S)을 수행한다. 이 연산을 공통 피승수 곱셈(common-multiplicand multiplication)이라 한다. 제안 모듈라 곱셈의 기본 개념은 이 두 연산에서 S를 공통적으로 곱하므로 공통적인 계산 부분을 추출하여 한번만 계산하자는 것이다. 이러한 개념은 논문 [10]에서도 유사하게 적용된 바 있는 데 이 경우에는 나눗셈을 피하기 위해 상위 1~2자리가 모두 "1"인 N을 택하는 조건이 부가되는 단점이 있었다. Montgomery 곱셈을 그대로 사용할 경우에는 공통적인 계산 부분을 추출하기가 어렵다. 그 이유는 중간 계산값들이 각각 C와 S에 의존하여 변하기 때문이다. 그러므로 Montgomery 곱셈을 다음과 같이 변형한다.

$$\begin{aligned}
REDC(A\times B) &= ABR^{-1} \bmod N \\
&= ABb^{-n} \bmod N \\
&= A(B_{n-1}b^{n-1}+B_{n-2}b^{n-2}+\dots \\
&\quad +B_0b^0)b^{(m-n)}b^{-m} \bmod N \\
&= (B_{n-1}(Ab^{m-1} \bmod N) \\
&\quad +B_{n-2}(Ab^{m-2} \bmod N)+\dots \\
&\quad +B_0(Ab^{m-n} \bmod N))b^{-m} \bmod N \\
&= (B_{n-1}A_{R0}+B_{n-2}A_{R1}+\dots \\
&\quad +B_0A_{Rn-1})b^{-m} \bmod N \\
&= Tb^{-m} \bmod N \qquad (1)
\end{aligned}$$

여기서 T는 $B_{n-1}A_{R0}+B_{n-2}A_{R1}+\dots+B_0A_{Rn-1}$이며, $0\le i\le n-1$일 때 $A_{Ri}=Ab^{m-1-i} \bmod N$이다. 만약 $m-1-i$가 0보다 크면 A_{Ri}값은 A를 $m-1+i$ 자리만큼 왼쪽으로 이동시킨 값이며 이때 모듈라 감소는 하지 않는다. 그러나 $m-1-i$가 0보다 작을 경우의 A_{Ri} 값은 이전 값을 이용하여 반복적으로 구한다. 즉, $A_{Ri}=A_{Ri-1}b^{-1} \bmod N$와 같다.

한편, m값은 계산 시간과 사용 메모리를 고려해 0과 $n-1$의 사이 값으로 선택할 수 있지만 T의 자리수가 $n+2$자리임을 고려할 필요가 있다. 만약 m이 0이면 T는 $n+2$자리이고, 1이면 Tb^{-1}는 $n+1$자리가 되어 이를 다시 n자리로 감소시키는 부가적인 연산이 필요하다. m이 2이상이면 전체 계산량은 동일하나 m이 커질수록 사용하는 메모리가 많아진다. 그러므로 메모리 및 계산 측면에서 최적 m은 2이다. 따라서, m을 2로 하여 공통 피승수 곱셈 REDC(S×C)와 REDC(S×S)를 나타내면 다음과 같다.

$$\begin{aligned}
REDC(S\times C) &= (C_{n-1}S_{R0}+C_{n-2}S_{R1}+C_{n-3}S_{R2}\dots \\
&\quad +C_0S_{Rn-1})b^{-2} \bmod N \\
&= (C_{n-1}Sb+C_{n-2}S+C_{n-3}Sb^{-1} \bmod N \dots
\end{aligned}$$

$$+C_0Sb^{-n+2})b^{-2} \bmod N \quad (2)$$

$$\begin{aligned}REDC(S\times S) &= (S_{n-1}S_{R0}+S_{n-2}S_{R1}+S_{n-3}S_{R2}\cdots \\ &\quad +S_0S_{Rn-1})b^{-2} \bmod N \\ &= (S_{n-1}Sb+S_{n-2}S+S_{n-3}Sb^{-1} \bmod N \cdots \\ &\quad +S_0Sb^{-n+2})b^{-2} \bmod N \quad (3)\end{aligned}$$

식 (2)와 (3)에서 S_{Ri}가 공통적으로 사용되므로 한번만 계산할 수 있다. REDC(S×C)를 계산한 후 REDC(S×S)를 계산할 경우에는 모든 S_{Ri}를 저장해야 하지만 S_{R0}, $C_{n-1}S_{R0}$, $S_{n-1}S_{R0}$, S_{R1}순으로 번갈아 가면서 계산하면 S_{Ri}를 하나의 임시 메모리에 저장하여 사용할 수 있다. 여기서 S_{R0}는 S를 한자리 왼쪽으로 이동한 것이며 S_{R1}은 S이므로 간단히 계산할 수 있고 나머지 S_{Ri}는 다음과 같이 계산한다.

$$S_{R2}=S_{R1}b^{-1} \bmod N,\ S_{R3}=S_{R2}b^{-1} \bmod N,\ \cdots,$$
$$S_{Rn-1}=S_{Rn-2}b^{-1} \bmod N \quad (4)$$

REDC(S×C)와 REDC(S×S)를 동시에 계산하는 공통 피승수 곱셈은 그림 3과 같다. 여기서 첫번째 for 문의 첫번째와 두번째 줄이 S_{Ri}를 계산하는 공통 부분이며 두번째 for문은 m이 2이므로 $Tb^{-2} \bmod N$을 수행하는 부분이다.

```
Common-multiplicand modular multiplication :
X=REDC(S×C)=SCR^-1 mod N,  Y=REDC(S×S)=SSR^-1 mod N
   T=S;
   X=C_{n-1}Sb+C_{n-2}S ;          Y=S_{n-1}Sb+S_{n-2}S ;
   for i=n-3 to 0 step -1{
      m=T_0N_0' mod b;
      T=(T + mN)/b;
      X= X+TC_i;                   Y=Y+TS_i; }
   for i=1 to 0 step -1{
      m_X=X_0N_0' mod b;           m_Y=Y_0N_0' mod b;
      X=(X + m_XN)/b;              Y=(Y + m_YN)/b; }
   if (X≥N) X=X-N;                 If (Y≥N) Y=Y-N;
   return (X,Y);
```

그림 3. 공통 피승수 모듈라 곱셈
Fig. 3 Common-multiplicand modular multiplication

변형한 모듈라 곱셈의 계산량을 분석하면 다음과 같다. 먼저 하나의 S_{Ri}값을 계산하는데 n+1번의 작은 수 곱셈이 필요하므로 모든 S_{Ri}값을 계산하는 데 (n−2)(n+1)번의 작은 수 곱셈이 필요하다. 그러므로 REDC(S×C)를 계산하는 데 $2n+(n-2)(2n+1)+2(n+1) \approx 2n^2+n$번의 곱셈이 필요하다. 그러나 REDC(S×S)를 계산할 때에는 공통 계산 부분을 생략할 수 있으므로 $2n+(n-2)n+2(n+1) \approx n^2+2n$번의 곱셈이 필요하다. 변형한 방법의 한번 모듈라 곱셈 계산량은 Montgomery 곱셈과 동일하나 공통의 피승수를 가지는 두 번의 모듈라 곱셈은 $3n^2+3n$번의 곱셈으로 처리할 수 있다. 반면 Montgomery 곱셈으로는 $4n^2+2n$번이 필요하므로 변형한 방법은 이진 방식에서 e_i가 1일 경우의 계산량을 약 0.75배로 줄일 수 있다.

Ⅲ. 고속 멱승을 위한 적용

멱승 방법에는 이진 방식, m-ary 방식, window 방식 등이 있으며 left-to-right 방식과 right-to-left 방식으로 구분할 수 있다. 그러나 left-to-right 형태는 공통 피승수 곱셈이 없으므로 변형한 모듈라 곱셈을 적용하기가 어렵다. 본 장에서는 right-to-left 형태의 m-ary, window 멱승 방식 그리고 지수 folding 기법을 사용한 멱승 방식에 변형한 모듈라 곱셈 방법을 적용한다.

3.1 m-ary 멱승 방식

m-ary 멱승에서는 먼저 지수 E를 t개의 자리수를 가지는 m진수로 나타낸다. 즉, $E=\sum_{i=0}^{t-1} E_i m^i$, $E_i \in \{0, m-1\}$로 표현할 수 있다. Montgomery 곱셈을 사용한 right-to-left m-ary 멱승 방식을 나타낸 것이 그림 4이다[5]. Right-to-left 방식의 기본 원리는 A^{m^i}를 계산한 후 각 자리의 계수인 E_i에 해당하는 만큼 멱승을 수행하는 것이다. 그림 4의 두번째 for문에서 보는 바

```
m-ary exponentiation : C=A^E mod N
   for i=1 to m-1 step 1 { T[i]=R mod N; }
   S=AR mod N;
   for i=0  to t-1 step 1 {
      if( E_i≠0) then { T[E_i]=REDC(S×T[E_i]); S=REDC(S×S);}
      else S=REDC(S×S);
      for j=0 to log_2m-1 step 1 {S=REDC(S×S);} }
   for i=m-2 to i=1 step -1 { T[i]=REDC(T[i]×T[i+1]); }
   for i=m-2 to i=1 step -1 { T[m-1]=REDC(T[m-1]×T[i]); }
   C=REDC(T[m-1]×1)
   return (C);
```

그림 4. Right-to-left m-ary 멱승 방식
Fig. 4 Right-to-left m-ary exponentiation.

와 같이 E를 t개의 자리수로 표현할 경우 약 t(m-1)/m번의 공통 피승수 곱셈이 생긴다. 이 방식에서는 평균 $\lceil \log_2(E) \rceil + v + 2(m-2)$번의 모듈라 곱셈이 필요하다[5]. 여기서 v는 E를 m진수로 표현했을 때 0이 아닌 E_i의 개수이며 t(m-1)/m과 같다.

3.2 Window 멱승 방식

Window 방식은 m-ary 방식을 효율적으로 개선한 것으로 적당한 크기의 window를 잡고 멱승을 수행한다. 그러나 window를 일률적으로 같은 비트씩 잡는 것이 아니라 window의 크기가 w이하이면서 시작과 끝이 1이 되게 한다. Montgomery 곱셈을 사용한 right-to-left window 방식을 나타낸 것이 그림 5이다. 이 방식은 right-to-left m-ary방식의 원리와 비슷하게 window가 생기는 위치에서 A에 관한 멱승을 수행한 후 각 window값에 해당하는 만큼 멱승을 수행하도록 한 것이다[5]. 그림 5의 while문내에서 보는 바와 같이 window가 발생할 때마다 공통 피승수 곱셈이 발생한다. Window 방식에서 평균 window 개수가 $\lceil \log_2 E/(w+1) \rceil$ 개이므로 평균 $\lceil \log_2 E/(w+1) \rceil + \lceil \log_2 E \rceil + 2(2^{w-1}-1) + 3$번의 모듈라 곱셈이 필요하다.

```
Window exponentiation :  C=A^E mod N
for i=0 to 2^(w-1)-1 step 1 { T[i]= R mod N; }
S=AR mod N;
i=0 ;
while i≤k do{
    if (e_i=1) then
        j=MIN(k,i+w-1) ;
        while e_j=0 do
            j=j-1 ;
        e=(e_i..e_j)_2 /2 ;
        T[e]=REDC(S×T[e]); S=REDC(S×S);
        for k=i+1 to j step 1 { S=REDC(S×S); }
        i=j+1 ;
    else
        S=REDC(S×S);
        i=i+1;
}
for i=2^(w-1)-2 to 0 step -1 { T[i]=REDC(T[i]×T[i+1]); }
for i=2^(w-1)-2 to 1 step -1 {T[2^(w-1)-1]=REDC(T[2^(w-1)-1]×T[i]); }
T[2^(w-1)-1]=REDC(T[2^(w-1)-1]×T[2^(w-1)-1]);
T[0]=REDC(T[2^(w-1)-1]×T[0]);
C=REDC(T[0]×1);
return (C);
```

그림 5. Right-to-left window 멱승 방식
Fig. 5 Right-to-left window exponentiation.

3.3 지수 folding을 이용한 멱승 방식

본 절에서는 지수 folding 기법을 사용한 멱승 방식을 제시하고 변형한 모듈라 곱셈을 적용한다. 멱승에서 folding 기법이란 지수 E를 반으로 접어서 연산함으로써 계산 효율을 높이는 방법이다. 이 방식은 Schnorr가 제시한 서명 검증식 계산방법[3]에 기초하여 right-to-left 방식으로 변환한 것이다. 즉, k비트인 지수 E는 $E_H 2^{k/2} + E_L$로 표현할 수 있고 $C = A^E \bmod N = (A^{E_H})^{2^{k/2}} \cdot A^{E_L} \bmod N$으로 계산할 수 있다. 이 멱승 방식의 기본 개념은 E_H와 E_L을 이진수로 나타내었을 때 공통적으로 "1"인 것에 대한 멱승은 한번만 할 수 있다는 것이다.

멱승 연산시에는 식(5)와 같이 E_H와 E_L를 AND시켜 공통으로 "1"인 부분만 추출한다. 그리고 E_{com}과 E_H을 exclusive-or(XOR)시키고 E_{com}과 E_L을 XOR한다.

$$E_{com} = E_H \wedge E_L,\quad E_{HXOR} = E_{com} \oplus E_H,\quad E_{LXOR} = E_{com} \oplus E_L \tag{5}$$

이 경우 E_{com}, E_{HXOR} 및 E_{LXOR}는 각각 E를 반으로 접었을 때 상위 반절과 하위 반절이 공통적으로 "1"인 수, 상위 반절만이 "1"인 수, 하위 반절만이 "1"인 수를 의미하며 이진수로 표현하면 다음과 같다. 여기서 $e_{com_i}, e_{HXOR_i}, e_{HXOR_i} \in \{0, 1\}$이다.

$$E_{com} = \sum_{i=0}^{\frac{k}{2}-1} e_{com\_i} \cdot 2^i,\quad E_{HXOR} = \sum_{i=0}^{\frac{k}{2}-1} e_{HXOR\_i} \cdot 2^i,$$

$$E_{LXOR} = \sum_{i=0}^{\frac{k}{2}-1} e_{LXOR\_i} \cdot 2^i \tag{6}$$

따라서 E_H와 E_L는 다음과 같이 표현할 수 있다.

$$E_H = E_{com} + E_{HXOR},\quad E_L = E_{com} + E_{LXOR} \tag{7}$$

지수 folding 기법을 이용하여 멱승을 수행하는 과정은 그림 6과 같다. 그림 6의 첫번째 for문에서 $A^{E_{com}} \bmod N$, $A^{E_{HXOR}} \bmod N$ 그리고 $A^{E_{LXOR}} \bmod N$을 먼저 구한 후, $A^{E_H} = A^{E_{com}} \cdot A^{E_{HXOR}} \bmod N$과 $A^{E_L} = A^{E_{com}} \cdot A^{E_{LXOR}} \bmod N$을 계산한다. 그리고 두번째 for문에서 최종적으로 $A^E = (A^{E_H})^{2^{k/2}} \cdot A^{E_L} \bmod N$을 계산한다. 멱승 연산에 필요한 계산량을 분석함에 있어 E의 Ham-

ming weight는 k/2라 가정한다. 그러면 E_H와 E_L의 Hamming weight는 k/4이며 E_{com}, E_{HXOR}, E_{LXOR}의 Hamming weight는 각각 k/8이 된다. 그러므로 k + 3k/8 +3번의 모듈라 곱셈이 필요하다. 이 방식에서는 첫번째 for문에서 E_{com}, E_{HXOR} 및 E_{LXOR}가 "1"일 때 공통 피승수 모듈라 곱셈을 3k/8번 사용할 수 있다.

```
Folding : C=A^E mod N
C_com = C_L = C_H = R mod N;
S=AR mod N;
for i=0 to k/2-1 step 1 {
    if( e_com_i = 1)  { C_com = REDC(S×C_com); S=REDC(S×S);}
    else if( e_HXOR_i = 1) { C_H = REDC(S×C_H);S=REDC(S×S);}
    else if( e_LXOR_i = 1) { C_L = REDC(S×C_L);S=REDC(S×S);}
    else S=REDC(S×S); }
C_H = REDC(C_H×C_com);
C_L = REDC(C_L×C_com);
for i=0 to k/2-1 step 1 { C_H = REDC(C_H×C_H); }
C=REDC(C_H×C_L);
C=REDC(C×1);
return(C);
```

그림 6. 지수 folding 기법을 사용한 멱승
Fig. 6 The exponentiation by folding the exponent in half.

Ⅳ. 비교 분석 및 구현 결과

4.1 멱승 방식별 비교 분석

멱승 알고리듬의 계산량을 비교할 때에는 모듈라 곱셈수를 고려한다. 그러나 모듈라 곱셈은 기수 b보다 작은 수들의 곱셈으로 이루어지므로 결국 멱승에 필요한 작은 수 곱셈을 분석함으로써 전체 계산량을 비교할 수 있다. 본 논문에서는 멱승 A^E mod N을 구현하는 데 A, E 및 N을 모두 512비트로 가정하였다. A와 N의 기수 b는 2^{16}으로 하였는데 그 이유는 C 언어 컴파일러가 16비트의 곱셈을 지원하기 때문이다. 이 경우 A나 N을 기수 b로 표현하면 자리수 n은 32가 된다.

이진 멱승 방식에서 E가 k비트이면 left-to-right 방식과 right-to-left 방식 모두 약 1.5k번의 모듈라 곱셈이 필요하다[5]. 그러므로 이 방식에 Montgomery 곱셈을 사용하면 총 $1.5k(2n^2+n)$번의 작은 수 곱셈이 필요하다. 그러나 right-to-left 멱승 연산에서는 k/2번의 공통 피승수 곱셈이 생기므로 변형한 방법을 사용하면 $0.5k(3n^2+3n)+0.5k(2n^2+n)$번의 곱셈이 필요하다. 이는 E가 512 비트일 경우 768번의 모듈라 곱셈을 646번으로 감소시킨 것과 동일하다.

Left-to-right m-ary 멱승 방식에서는 m의 선택에 따라 계산 테이블 및 곱셈수가 달라지게 되는 데 $m=2^4$로 하면 약 641번의 모듈라 곱셈이 필요하며 m-1개의 저장용 테이블이 필요하다[10]. Right-to-left m-ary 방식에서는 $\lceil \log_2(E) \rceil + v + 2(m-2)$번의 모듈라 곱셈과 m−1개의 저장용 테이블이 필요하다[5]. $m=2^4$이면 $t=128$, $v=128\cdot(m-1)/m=120$이므로 약 660번의 모듈라 곱셈이 필요하다. 그러나 right-to-left 멱승 연산에서 v번의 공통 피승수 곱셈이 생기므로 변형한 모듈라 곱셈을 적용하면 $(k-v)(2n^2+n)+v(3n^2+3n)+2(m-2)(2n_2+n)$번의 작은 수 곱셈이 필요한 데 이는 약 603번의 모듈라 곱셈을 수행하는 결과와 같다. 그러나 $m=2^3$이면 $t=171$, $v=171\cdot(m-1)/m=150$이다. 이때 계산량은 $(512-150)(2n_2+n)+150(3n_2+3n)+12(2n_2+n)$번이고 이는 약 602번의 모듈라 곱셈과 같다. 결국 E가 512비트인 경우 m-ary 방식에서는 $m=2_3$으로 하는 것이 $m=2_4$로 하는 것에 비해 계산 속도는 비슷하나 계산 테이블을 절반 정도로 줄일 수 있다.

Left-to-right window 방식에서는 512 비트 멱승시 window 크기가 5일 때 최적이며 평균 609번의 모듈라 곱셈이 필요하다[10]. 이 경우 2^{w-1}개의 저장용 테이블이 필요하다. Right-to-left 방식에서는 $\lceil \log_2 E/(w+1) \rceil + \lceil \log_2 E \rceil + 2(2^{w-1}-1)+3$번의 모듈라 곱셈이 필요한 데 이는 631번의 모듈라 곱셈과 동일한 계산량이다. 그러나 right-to-left 멱승 연산에서는 window 개수만큼 공통 피승수 곱셈이 생기므로 변형한 모듈라 곱셈을 적용하면 $(\lceil \log_2 E \rceil - \lceil \log_2 E/(w+1) \rceil)(2n^2+n) + (\lceil \log_2 E/(w+1) \rceil)(3n^3+3n) + 2(2^{w-1}-1)+3)(2n^2+n)$번의 작은 수 곱셈이 필요하다. 이는 590번의 모듈라 곱셈을 수행한 것과 같다. 그러나 window 크기를 4로 하면 $(512-103)(2n^2+n)+103(3n^2+3n)+17(2n^2+n)$이고 이것은 약 583번의 모듈라 곱셈을 수행하는 결과와 같다. 지수가 512비트이고 변형한 곱셈을 사용하는 경우에는 window 크기가 4일 때 속도도 빠르고 메모리 사용을 절반 정도로 감소할 수 있다. 그 이유는 window 크기가 작으면 평균 window 개수는 증가하므로 window가 생기는 위치의 멱승 값

을 계산하는 연산은 많아지나 변형 모듈라 곱셈으로 보상할 수 있고, 반면 저장용 테이블이 감소하게 되어 테이블을 멱승하는 계산량이 줄어들기 때문이다.

Folding 기법을 이용한 멱승은 left-to-right 방식이나 right-to-left 방식으로 구현할 수 있으며 각각 k +3k/8 +1와 k +3k/8 +3번의 모듈라 곱셈이 필요하다. 그러나 Right-to-left 방식에서는 3k/8번의 공통 피승수 곱셈이 생기므로 변형한 모듈라 곱셈을 사용하면 $3k/8(3n^2+3n)+(k/8+k/2+3)(2n^2+n)$번이 필요하다. 이는 약 615번 정도의 모듈라 곱셈으로 처리할 수 있는 계산량이 된다. 상기한 각 멱승 방식의 모듈라 곱셈수와 512비트 데이타를 저장하기 위한 테이블 수를 요약하면 표 1과 같다. 단, 필요한 테이블 수는 ( )로 표시하였다.

일반적으로 멱승 방식에서 left-to-right 형태가 right-to-left 형태보다 계산 측면에서 유리하다. 그러나 left-to-right 형태에서는 공통 피승수 곱셈이 발생하지 않는다. 반면, Right-to-left 형태에서는 공통 피승수 곱셈이 생기므로 이를 효과적으로 처리한다면 left-to-right 형태보다 고속 멱승을 실현할 수 있다. 특히, 이진 방식이나 folding 기법을 이용한 방식에서는 공통 피승수 곱셈의 발생확률이 높으므로 개선효율이 커지게 된다. 이 방식들은 특별한 데이타 저장용 테이블이 필요하지 않으면서 고속이므로 IC카드와 같은 제한적인 메모리를 사용하는 환경에 적합하다. 반면, m-ary나 window 방식은 이들보다는 고속이지만 데이타 저장용 테이블이 필요하므로 메모리가 확보된 컴퓨터나 네트워크 시스템에 활용할 수 있다.

4.2 구현 결과

상기한 멱승 방식들을 PC 및 workstation에서 C 언어로 구현하였다. 고전적인 알고리듬, Montgomery 곱셈 그리고 변형한 방법으로 모듈라 곱셈을 구현하여 각 멱승 방식에 적용하였다. PC는 Pentium(586)/133MHz을 사용하였고 DOS 환경 하에서 Borland C로 컴파일하였다. Workstation은 Solaris 2.5.1환경 하

표 1. 모듈라 곱셈수 및 테이블 비교

Table 1. Comaprison in the number of modular multiplications and required storage tables.

Modular multiplication \ Exponentiation	Binary	m-ary	Window	Folding
Montgomery(L-to-R)	768	641(15)	609(16)	705
Montgomery(R-to-L)	768	660(15)	631(16)	707
Modified (R-to-L)	646	602(7)	583(8)	615

표 2. PC-586(133MHz)에서의 멱승 구현 속도

Table 2. Experimental speed for each exponentiation on PC-586(133MHz). [sec]

Modular multiplication \ Exponentiation	Binary	m-ary	Window	Folding
Conventional(L-to-R)	1.559	1.319	1.244	1.437
Montgomery(L-to-R)	0.771(768)	0.638(636)	0.617(615)	0.708(706)
Montgomery(R-to-L)	0.770(768)	0.657(655)	0.632(630)	0.710(708)
Modified (R-to-L)	0.645(643)	0.598(596)	0.580(578)	0.617(615)

표 3. SUN Ultra2(200MHz)에서의 멱승 구현 속도

Table 3. Experimental speed for each exponentiation on SUN Ultra2(200MHz). [sec]

Modular multiplication \ Exponentiation	Binary	m-ary	Window	Folding
Conventional(L-to-R)	0.450	0.375	0.356	0.412
Montgomery(L-to-R)	0.306(768)	0.255(640)	0.245(614)	0.280(702)
Montgomery(R-to-L)	0.306(768)	0.263(660)	0.252(632)	0.281(705)
Modified (R-to-L)	0.255(640)	0.239(600)	0.231(580)	0.243(610)

에서 SUN Ultra2/200MHz를 사용하였다. 표 2와 3은 PC 및 workstation에서 한번의 멱승을 수행하는데 소요된 평균 시간을 나타낸 것이다. 이 결과는 세가지 모듈라 곱셈을 적용하여 멱승을 1000번씩 수행 후 이를 평균한 것이다. 표의 ( )는 Montgomery 곱셈을 사용한 이진 방식을 기준으로 한 상대적인 모듈라 곱셈 수를 나타낸 것이다.

표에서 보는 바와 같이 Montgomery 곱셈은 고전적인 알고리듬에 비해 PC에서는 약 0.5배, workstation에서는 0.68배의 시간이 소요된다. Montgomery 곱셈을 각 멱승 방식에 적용했을 경우 이론적인 소요시간과 구현 결과가 거의 일치하였다. 또한 변형한 곱셈을 적용했을 경우에는 Montgomery 곱셈을 사용하는 것보다 고속 멱승을 실현할 수 있었고 그 비율도 표 1의 분석결과와 비슷하였다. 변형한 방법은 PC보다는 workstation에서 개선비율이 높았는데 이는 데이타의 읽고 쓰는 억세스 시간의 차이에 의한 것 같다. Montgomery 곱셈을 한번 수행하는 데 PC에서는 약 1.0026 msec, workstation에서는 약 0.3984 msec 소요되었다. 공통 피승수 곱셈으로 2회의 모듈라 곱셈을 수행시 PC에서 약 1.5170 msec, workstation에서는 0.7969 msec 소요되었다. 다른 PC나 workstation에서도 수행한 결과, 멱승 연산의 속도는 시스템 기종의 성능, 컴파일러의 선택, 혹은 최적화 기능의 추가 등에 따라 약간씩 달랐지만 모듈라 곱셈 방식에 따른 상대적인 비는 비슷하였다.

V. 결 론

본 논문에서는 동일한 피승수를 가지는 두번의 모듈라 곱셈 연산시 공통적으로 계산되는 부분이 생기도록 Montgomery 모듈라 곱셈을 변형하였다. 변형한 방법을 right-to-left 형태의 멱승에 적용하면 기존의 방식보다 고속 연산이 가능하다. 특히, IC카드와 같은 저메모리 환경하에서는 지수 folding 기법을 사용한 멱승 방식에 변형한 곱셈 방법을 사용하는 것이 효과적이다. m-ary나 window 방식에서는 기존 방식보다 고속이면서 사용 메모리를 절반 정도로 줄일 수 있는 장점이 있다. 따라서 이 방법은 RSA나 ElGamal 형 공개 키 암호시스템을 효과적으로 구현할 수 있다.

참 고 문 헌

1. T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," *IEEE Trans. Inform. Theory*, Vol. 31, No. 4, pp. 469-472, July 1985.
2. R. Rivest, A. Shamir and L. Adleman, "A method for obtaining digital signatures and public key cryptosystems," *Comm. of ACM*, Vol. 21, No. 2, pp. 120-126, Feb. 1978.
3. C. P. Schnorr, "Efficient signature generation for smart cards," *Advances in Cryptology-CRYPTO '89* Proceedings, Springer-Verlag, pp. 239-252, 1990.
4. National Institute Standard Technology, *Specifications for a Secure Hash Standard(SHS)*, FIPS YY Draft, Jan. 1992.
5. D. E. Knuth, *The Art of Programming, Vol. 2: Seminumerical Algorithms*, 2nd Ed. Addison-Wesley, 1981.
6. A. Bosselaers, R. Govaerts and J. Vandewalle, "Comparison of three modular reduction functions," *Advances in Cryptology, Proc. CRYPTO '93*, pp. 175-186, 1993.
7. Paul Barrett, "Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor," *Advances in Cryptology, CRYPTO '86*, pp. 311-323, 1986.
8. Peter L. Montgomery, "Modular multiplication without trial division," *Math. of Comp.* Vol. 44, No. 170, pp. 519-521, April 1985.
9. S. R. Dusse and B. S. Kaliski, "A cryptographic library for the Motorola DSP 56000," *Advances in cryptology, Proc. EUROCRYPT '90*, pp. 230-244, 1990.
10. 하재철, 이창순, 문상재, "고속 멱승을 위한 새로운 모듈라 감소 알고리듬," *한국통신 정보보호학회 논문집*, 제 6권 1호, pp. 151-159, 1996. 11.

河 在 喆(Jae-Cheol Ha) 정회원
1989년 2월 : 경북대학교 공과대학 전자공학과(공학사)
1991년 6월 : 육군 통신장교
1993년 8월 : 경북대학교 대학원 전자공학과(공학석사)
1994년~현재 : 경북대학교 대학원 전자공학과 박사과정
※주관심분야 : 부호이론, 정보보호, 디지탈 통신

文 相 在(Sang-Jae Moon) 정회원
1972년 2월 : 서울대학교 공과대학 공업교육과(전자공학, 공학사)
1974년 2월 : 서울대학교 대학원 전자공학과(통신공학, 공학석사)
1984년 6월 : 미국 UCLA(통신공학, 공학박사)
1984년 6월~1985년 6월 : UCLA Postdoctor 근무
1984년 6월~1985년 6월 : 미국 OMNET 컨설턴트
1974년~현재 : 경북대학교 공과대학 전자전기공학부 교수
※주관심분야 : 정보보호, 디지탈 통신, 정보통신망