고속 멱승을 위한 새로운 모듈라 감소 알고리듬

°하 재철*, 이 창순**, 문 상재*
* 경북대학교 전자전기공학부
** 경산대학교 정보처리과

A New Modular Reduction Algorithm for Fast Exponentiation

°Jae-Cheol Ha*, Chang-Soon Lee**, Sang-Jae Moon*

* School of Electronics and Electrical Eng., Kyungpook National University
** Dept. of Information Processing, Kyungsan University

본 연구는 정보통신연구관리단의 국책기술개발 사업 지원에 의해서 이루어졌습니다

요 약 문

본 논문에서는 right-to-left 형태의 멱승 연산에 적합한 고속 모듈라 감소 알고리듬을 제안하고 이를 여러 멱승 방식에 적용했을 경우의 계산 속도 및 메모리 사용효율을 기존의 방식들과 비교하였다. 분석 결과, 기존의 방식보다 고속으로 멱승을 수행할 수 있고 m-ary 방식이나 window 방식에서는 사용 메모리를 줄일 수 있다.

1. 서 론

RSA와 같이 소인수분해 문제에 근거한 암호시스템[1]이나 ElGamal 시스템 등과 같은 이산대수 문제에 근거한 암호시스템[2]에서 멱승 연산이 필요하다. 암호시스템에 사용되는 모듈라 멱승은 일반적으로 A^E mod N으로 표시할 수 있다. 여기서 A와 N은 안전도에 따라 가변적이나 512비트 이상을 사용한다. E가 RSA 시스템이나 ElGamal 시스템에서 비밀 키인 경우 512비트 이상, 그리고 이산 대수 문제에 근거한 디지탈 서명의 경우는 140비트 이상을 사용한다[3, 4]. 이와 같은 멱승 연산은 암호시스템에서 연산시간이 많으므로 이를 감소시키는 것이 시스템의 성능을 결정할 수 있는 중요한 요소가 될 수 있다.

고속 멱승 연산을 위해서는 기본 연산인 모듈라 곱셈을 고속으로 처리하거나 모듈라 곱셈수를 줄이는 효과적인 멱승 방식을 사용해야 한다. 전자에 해당하는 방법은 classical 알고리듬[5, 6], Yang 알고리듬[7], Barrett 알고리듬[8], Tanaka-Okamoto 알고리듬[9] 그리고 현재까지 가장 고속으로 알려진 Montgomery 알고리듬[10, 11] 등이 있다. 후자에 해당하는 방법으로는 이진 방식[5], 이진 잉여(binary redundant) 방식[12], m-ary 방식[5], window 방식[5] 등이 있다.

본 논문에서는 새로운 모듈라 감소 알고리듬을 제안한다. 제안 알고리듬은 right-to-left 형태의 멱승 방식에 적용할 경우 고속으로 모듈라 감소를 수행할 수 있다. 본 논문의 제 2장에서는 제안 알고리듬의 기본 개념을 설명하고, 제 3장에서는 이를 각 멱승 방식에 적용할 수 있는 알고리듬을 제시한다. 제 4장에서는 제안된 알고리듬을 사용했을 경우 기존 방식들과 비교 분석하고 마지막으로 결론을 맺는다.

2. 제안 모듈라 감소 알고리듬

멱승 $A^E \bmod N$ 연산시 A는 k비트이고 b진수로 나타내었을 때 n자리인 수라 가정한다. 즉, $A=\sum_{i=0}^{n-1}A_i b^i$, $A_i \in [0, b-1]$로 표현할 수 있다. 이진 멱승 방식에서 k비트인 E를 이진수로 변환한다. 즉, $E=\sum_{i=0}^{k-1}e_i 2^i$, $e_i \in \{0,1\}$로 표현할 수 있다. 이진방식은 left-to-right 방식과 right-to-left 방식으로 구분할 수 있다. 이를 비교한 것이 그림 1이다.

```
Binary(A, E, N)
S=1;
for i=k-1  to i=0 step -1
  S=S×S mod N;
  if( e_i=1) then S=S×A mod N;
 return(S);
```

(a) Left-to-right

```
Binary(A, E, N)
M=1;
S=A;
for i=0  to i=k-1 step 1
   if( e_i=1) then M=S×M mod N;
   S=S×S mod N;
return(M);
```

(b) Right-to-left

Fig. 1. Binary method.

제안하는 알고리듬은 right-to-left 이진방식에서 $e_i=1$일 때 동일한 S에 관하여 두번 모듈라 곱셈을 한다는 것을 이용한다. 즉, M=S×M mod N과 S=S×S mod N을 동시에 모듈라 감소를 함으로써 계산 효율을 높이자는 것이다. S×M mod N 과 S×S mod N은 다음과 같이 표현된다.

$$S\times M \bmod N = S(M_0 b^0 + M_1 b^1 + \cdots + M_{n-1} b^{n-1}) \bmod N$$

$$= (Sb^0 M_0 \bmod N + Sb^1 M_1 \bmod N + \cdots + Sb^{n-1} M_{n-1} \bmod N) \bmod N$$

$$= (Sb^0 \bmod N \cdot M_0 + Sb^1 \bmod N \cdot M_1 + \cdots + Sb^{n-1} \bmod N \cdot M_{n-1}) \bmod N \quad (1)$$

$$S\times S \bmod N = S(S_0 b^0 + S_1 b^1 + \cdots + S_{n-1} b^{n-1}) \bmod N$$

$$= (Sb^0 S_0 \bmod N + Sb^1 S_1 \bmod N + \cdots + Sb^{n-1} S_{n-1} \bmod N) \bmod N$$

$$= (Sb^0 \bmod N \cdot S_0 + Sb^1 \bmod N \cdot S_1 + \cdots + Sb^{n-1} \bmod N \cdot S_{n-1}) \bmod N \quad (2)$$

위의 식 (1)과 (2)에서 $S \cdot b^i \bmod N$연산이 동일하게 사용된다. 여기서 $i=0, \cdots, n-1$이다. 그러므로 S×M mod N에서 계산한 $S \cdot b^i \bmod N$을 S×S mod N에 사용할 수 있다. 물론 저장 메모리의 효율을 높이기 위해서는 S×M mod N과 S×S mod N의 부분 결과 값을 번갈아 가면서 구할 수도 있다.

한편, 각 $S_{bi}=S \cdot b^i \bmod N$은 다음과 같이 이전 계산 값들을 이용하여 계산할 수 있다.

$$S_{b0} = Sb^0 \bmod N = S$$

$$S_{b1} = Sb^1 \bmod N = Sb^0 b^1 \bmod N = S_{b0} b^1 \bmod N$$

$$S_{b2} = Sb^2 \bmod N = Sb^1 b^1 \bmod N = S_{b1} b^1 \bmod N$$

$$S_{bn-1} = Sb^{n-1} \bmod N = Sb^{n-2} b^1 \bmod N = S_{bn-2} b^1 \bmod N \quad (3)$$

식 (3)의 연산들은 이전 S_{bi}값을 한 자리수 이동하여 구할 수 있다. 각 S_{bi}를 구하는 연산에서는

몫이 항상 1 자리수이므로 n번의 작은 수 곱셈과 1번의 나눗셈이 필요하다. 그러므로 식 (3)을 구하는 데는 총 $n(n-1)$번의 작은 수 곱셈과 n-1번의 나눗셈이 필요하다. 여기서 N의 최상위 한 자리수를 모두 1로 하면(예를 들어 16진수인 경우 FFFF) 몫추정이 나눗셈 없이 해결된다. 또한 식 (1)의 최종 모듈라 감소 이전 값은 최대 $k+b+\lceil \log_2 n \rceil$ 비트이고 이는 n+2 자리이다. 이 연산도 몫 추정 기법 등을 이용하여 통해 간단히 수행할 수 있다. 결국 모듈라 수 N의 최상위 한자리를 모두 1로 한다면 나눗셈 없이 n+2자리 정도의 모듈라 감소는 간단히 수행할 수 있다.

모듈라 수 N의 최상위 한자리 수가 모두 1이라면 n+1자리의 모듈라 감소는 나눗셈 없이 쉽게 해결할 수는 있지만 부가적인 뺄셈이 필요하다. 이와 같은 경우의 모듈라 감소는 N을 diminished radix 형태로 표현하면 간단히 덧셈만으로도 해결할 수 있다[13, 14]. N의 diminished radix는 $N'=b^k-N$으로 표시할 수 있고 N'은 n-1자리 수가 된다. 그러므로 S_{bi}를 구할 때 S_{bi-1}의 최상위 자리수 값과 N'을 곱하여 S_{bi-1}값과 더하면 된다. 더한 수가 n자리수 이상이 되어 carry가 발생하면 한번 더 N'을 더하면 된다. 또한 S_{bi}를 구할 때 carry발생 여부를 먼저 점검하고 carry가 발생한다면 S_{bi-1}의 최상위 자리수에 1을 더하여 N'과 곱한 후 S_{bi-1}값과 더하여 모듈라 감소를 수행할 수도 있다. 이와 같은 기법을 사용하면 모두 덧셈으로 모듈라 감소를 할 수 있어 비교적 큰 자리수 덧셈보다 비효율적인 뺄셈을 생략할 수 있다. 이때의 계산량은 $(n-1)(n-1)$번의 작은 수 곱셈과 부수적인 덧셈연산이 필요하지만 carry발생여부를 판별하는 데 한번 정도의 곱셈이 필요하므로 이 방법 역시 약 $n(n-1)$번 정도의 작은 수 곱셈이 필요하다. 물론 N의 상위자리 중 모두 1인 자리수가 많을수록 더 효과적으로 모듈라 감소를 할 수 있다.

모듈라 곱셈시 계산량을 보면, S×M mod N 연산시에는 최종 모듈라 감소시 2n번의 작은 수 곱셈을 포함하여 약 $2n^2+n$번의 작은 수 곱셈만이 필요하다. 그러나 S×S mod N 연산시에는 위의 결과를 이용할 수 있으므로 모듈라 곱셈을 수행하는 데 n^2+2n번의 곱셈만이 필요하다. 결국 S×M mod N 과 S×S mod N을 동시에 수행할 경우에는 $3n^2+3n$번의 작은 수 곱셈만이 필요하다. 그리고 $e_i=0$일 경우에는 S×S mod N을 독립적으로 수행하는 데 여기에는 $2n^2+n$번의 작은 수 곱셈이 필요하다.

일반적으로 n자리수인 두 수에 관한 곱셈을 위해서는 n^2번의 작은 수 곱셈이 필요하다. 모듈라 감소시에는 가장 빠른 알고리듬으로 알려진 Montgomery 방식에서 n^2+n번의 작은 수 곱셈이 필요하다. (이 경우에도 $-N_0^{-1}=1 \bmod b$가 되는 N_0를 선택하면 n^2으로 줄일 수 있음). 그러므로 1회의 모듈라 곱셈시에는 모두 $2n^2+n$ 정도의 작은 수 곱셈이 필요하다. Montgomery 알고리듬으로 멱승시 $e_i=1$이면 2회의 곱셈과 모듈라 감소가 필요하므로 총 $4n^2+2n$의 작은 수 곱셈이 필요하다.

제안 방식에서는 나눗셈을 없애기 위해 N의 최상위 한 자리를 모두 1로 함을 전제로 한다. 이는 시스템 전체의 안전도에는 큰 영향이 없다. 제안 방식으로 이진 멱승을 수행하면 $e_i=1$이면 총 $3n^2+3n$의 작은 수 곱셈이 필요하다. 그러므로 Montgomery 알고리듬을 이용하는 것보다 $e_i=1$인 경우의 연산량을 약 0.75배로 줄일 수 있다. 그리고 $e_i=0$인 경우의 연산량은 $2n^2+n$으로 Montgomery 모듈라 감소와 동일하다.

예를 들어 E가 512비트의 수이고 0과 1의 개수가 동일한 가정하에서 $b=2^{16}$이면 n=32이다. 일반적으로 이진방식은 768번의 곱셈이 필요하므로 $768(2n^2+n)$번의 작은 수 곱셈이 필요하다. 즉, 1597440번이다. 제안 방식에서는 $e_i=0$일 때 $256(2n^2+n)$번과 $e_i=1$일 때 $256(3n^2+3n)$번의 작은 수 곱셈이 필요하므로 $256(5n^2+4n)$번이 요구된다. 즉 1335552번이다. 그러므로 역시 전체 계산량을 16.4%정도

개선할 수 있어 약 642회의 모듈라 곱셈 시간으로 처리할 수 있다. 한번 계산한 S_{bi}는 두 모듈라 감소에만 사용하므로 별도의 계산 테이블이나 많은 메모리를 요구하지 않는다.

계산량을 일반식으로 기술하면 left-to-right 방식에서는 $1.5k(2n^2+n)$으로 표현할 수 있고 제안 방식에서는 $0.5k(5n^2+4n)$와 같다. 결국 제안하는 알고리듬은 right-to-left 형태의 멱승 연산에서 동일한 변수를 두 개이상의 다른 값과 곱하여 모듈라 감소를 할 경우 이 두 연산의 중복되는 부분을 한번으로 감소함으로써 전체 계산량을 줄인다는 것이다.

3. 고속 멱승을 위한 적용

멱승 방법에는 이진 방식, 이진 잉여 방식, m-ary 방식, window 방식 등이 있으며 대부분 left-to-right 형태로 구현한다. 그 이유는 left-to-right 형태가 right-to-left 형태보다 비교적 구현이 용이하고 속도가 비슷하거나 혹은 빠르기 때문이다. 제안한 모듈라 감소는 left-to-right 형태에는 적용하기가 어렵다. 본 장에서는 각 멱승 방식을 right-to-left 형태로 변형하여 제안 알고리듬을 적용하고자 한다.

3.1 이진 잉여 방식

이진 잉여 방식은 멱승 연산을 수행할 경우 1의 개수를 줄이기 위해 E의 이진 잉여 표현(binary redundant representation) 방법을 이용한다. 예를 들어 지수가 47로 주어지면 다음과 같이 지수를 변형한다. $E=47=1011111=110000\bar{1}=1100000-1$이 된다. 여기서 $\bar{1}$은 -1을 의미한다. 이진 잉여 표현으로 지수를 변환하여 멱승을 수행하면 지수 E에 대한 1과 $\bar{1}$의 개수를 일반적인 이진수로 표현할 때의 1의 수보다 평균적으로 약 33%가량 줄일 수 있다[12].

그러므로 k비트의 멱승 연산을 위해서는 k번의 모듈라 제곱과 $0.5k\times0.67$번의 모듈라 곱셈이 필요하다. 예를 들면 512비트 지수 연산시 제곱을 위한 512번의 모듈라 제곱이 필요하며 곱셈을 위한 $256\times0.67=170$번의 연산이 필요하다. 그러므로 총 682번 정도의 곱셈 연산이 소요된다. 이진 잉여 방식을 나타낸 것이 그림 2(a)이다. 제안한 모듈라 감소 알고리듬을 적용하기 위해서는 이를 right-to-left 방식으로 변환해야 한다. 본 논문에서는 그림 3(b)와 같은 알고리듬을 제시하며 여기에 새로운 모듈라 감소 알고리듬을 적용할 수 있다. 단지 $e_i=1$일 때의 값과 $e_i=\bar{1}$일 때의 값을 별도로 구해서 이 두 값을 나누면 된다. 이는 원래 right-to-left 이진 잉여 방식에서 e_i가 0이 아닐 경우의 계산량을 약 0.75배 줄일 수 있다.

```
Bin-Red(A, E, N)
 A1 = A^-1 mod N
 S=1;
 for i=k-1  to i=0 step -1
    S=S×S mod N;
    if( ei = 1) then S=S×A mod N;
    if( ei = 1̄) then S=S×A1 mod N;
  return(S);
```

(a) Left-to-right

```
Bin-Red(A, E, N)
 M=1;  M1 = 1;
 S=A;
 for i=0  to i=k-1 step 1
    if( ei = 1) then  M=S×M mod N;
    if( ei = 1̄) then  M1=S×M1 mod N;
    S=S×S mod N;
 M = M · M1^-1 mod N;
 return(M);
```

(b) Right-to-left

Fig. 2. Binary redundant method.

제안 right-to-left 이진 잉여 방식의 계산량을 보면 제곱 연산시 $(k-0.33k)(2n^2+n)$번의 작은 수 곱셈과 $e_i=1$나 $e_i=\overline{1}$시 $0.33k(3n^2+3n)$번의 작은 수 곱셈이 필요하므로 총 $k(2n^2+n)+0.33k(n^2+2n)$의 작은 수 곱셈이 필요하다. 512 비트 지수 연산시 계산량을 보면 제곱 연산시 $(512-170)(2n^2+n)$번의 작은 수 곱셈과 $e_i=1$나 $e_i=\overline{1}$시 $170(3n^2+3n)$번의 작은 수 곱셈이 필요하므로 총 $512(2n^2+n)+170(n^2+2n)$의 작은 수 곱셈이 필요하다. 그러므로 총 1244650번의 작은 수 곱셈이 필요하므로 Montgomery 알고리듬을 사용한 원래 이진 방식을 22.1% 개선할 수 있다. 이는 598번의 모듈라 곱셈과 동일한 계산량이 된다.

그러나 이 알고리듬은 역수를 구해야 하는 단점이 있다. 구현 결과에 의하면 역수 계산은 extended Euclidean 알고리듬을 사용하면 약 10번의 모듈라 곱셈정도가 소요되는 것으로 알려져 있다. 이진 잉여 방식에서 E를 이진 잉여로 표현할 때는 LSB부터 MSB순으로 표현한다. 그러므로 left-to-right 방식은 E를 이진 잉여 표현으로 변환한 것을 저장한 후 멱승을 수행해야 한다. 그러나 제안 방식에서는 이진 잉여로 표현하면서 동시에 멱승을 할 수 있으므로 이진 잉여로 표현하는 데 필요한 메모리를 줄일 수 있다.

3.2 m-ary 방식

이진 방식이 지수를 이진수로 표현하여 멱승 연산을 하는 반면 m진수로 확장하여 계산할 수도 있다. 먼저 E를 l개의 자리수를 가지는 m진수로 나타낸다. 여기서 각 자리수 값 E_i은 $0 \le E_i < m$이다. Left-to-right 방식에서는 먼저 $1 \le i < m$을 만족하는 A^{E_i}를 구하여 저장한다. 멱승 연산은 m승 연산과 A^{E_i}를 곱하는 과정을 반복하여 수행한다. 그러나 right-to-left 방식에서는 조금 더 복잡하다. 이 방식의 기본 원리는 E를 m진수로 표현할 때의 위치값을 먼저 계산하고 후에 각 자리수 값 E_i에 해당하는 만큼 멱승을 수행한다. 이 두 방법을 비교하면 그림 3과 같다. 그림 3(b)의 right-to-left 방식에서 첫번째 for문 내의 $S=S\times S \bmod N$과 $S=S^{m-1} \bmod N$은 $S^m \bmod N$을 계산하기 위한 과정이나 모듈라 감소를 효과적으로 하기 위해 분리하였다. 두번째 및 세번째 for문은 $A^E=T[1]\cdot T[2]^2\cdot\cdots\cdot T[m-1]^{m-1}$을 수행하는 과정이다.

```
m-ary(A, E, N)
 for i=1 ; i<m step 1
   T[i]=A^Ei mod N;
 C=A ;
 for i=l-2 to 0 step -1
    C=C^m mod N;
    if(Ei ≠0)
      then C=C×T[Ei] mod N;
 return (C);
```

(a) Left-to-right

```
m-ary(A, E, N)
 for i=1 to m-1 step 1
   T[i]=1;
 S=A;
 for i=0  to i=l-1 step 1
   if( Ei≠0) then T[Ei]=S×T[Ei] mod N;
   S=S×S mod N;
   S=S^(m-1) mod N;
 for i=m-2 to i=1 step -1
   T[i]=T[i]×T[i+1] mod N;
 for i=m-2 to i=1 step -1
   T[m-1]=T[m-1]×T[i] mod N;
 return (T[m-1]) ;
```

(b) Right-to-left

Fig. 3. m-ary method.

필요한 계산량을 살펴보면, 먼저 left-to-right 방식에서는 평균적으로 필요한 곱셈수는 m-2+(⌈ $\log_m(E)$⌉ -1)×(⌈ $\log_2(m+1)$⌉ +$W_H(m)$-2+(m-1)/m)이다. 여기서 $W_H(m)$은 m의 Hamming weight이다. 반면 제안 알고리듬을 적용하기 전의 right-to-left 방식에서는 약 ⌈ $\log_2(E)$⌉ + v +2(m-2)번의 모듈라 곱셈이 필요하다. 여기서 v는 E를 m진수로 표현했을 때 0이 아닌 자리수이다. Left-to-right 방식에서 수행 속도는 m의 선택에 따라 달라지지만 $m=2^4$로 하여 멱승을 수행하면 512비트의 지수에 대해 약 641번의 모듈라 곱셈이 필요하다. 그리고 m-1개의 A^{E_i} 값을 저장할 임시 메모리가 필요하다. 위와 같은 조건이라면 제안 알고리듬을 적용하기 전의 right-to-left 방식에서는 $m=2^4$, l=128, v=128·(m-1)/m=120이므로 약 660번의 모듈라 곱셈이 필요하다.

제안한 알고리듬을 적용할 경우 필요한 작은 수의 곱셈수는 $(k-v)(2n^2+n)+v(3n^2+3n)+2(m-2)(2n^2+n)$ 이다. 그러므로 $(512-120)(2n^2+n)+120(3n^2+3n)+28(2n^2+n)$이고 n=32라면 1250040번이 된다. 이것은 약 601번의 모듈라 곱셈을 수행하는 결과와 같다. 즉, 이는 left-to-right 이진 방식을 21.8%, left-to-right m-ary 방식을 9.4% 개선한 것이 된다.

그러나 E를 표현하는 진수가 $m=2^3$이면 l=170, v=170·(m-1)/m=149이다. 이때 계산량은 $(512-149)(2n^2+n)+149(3n^2+3n)+12(2n^2+n)$이고 n=32라면 1247413번이 된다. 이것은 약 600번의 모듈라 곱셈을 수행하는 결과와 같다. 이는 left-to-right 이진 방식을 22.0% 개선한 것이 된다. 결국 m-ary방식에서는 $m=2^3$으로 하는 것이 $m=2^4$로 하는 것에 비해 계산 효율은 비슷하나 메모리 사용 측면에서는 계산 테이블을 반정도로 줄일 수 있어 효과적이다.

3.3 Window 방식

Window방식은 m진 방식을 효율적으로 개선한 방식으로 m진 방식에서와 같이 적당한 크기의 윈도우를 잡고 이에 대해서 m진 방식과 비슷한 방식으로 멱승을 수행한다. 그러나 일률적으로 m비트씩 잡는 것이 아니라 윈도우를 만들 때 윈도우의 크기가 w이하이면서 윈도우의 시작과 끝이 1이 되게 한다. 이 방식은 고속이면서 구현이 용이하여 많이 이용되는 방식이다. Left-to-right 방식을 나타낸 것이 그림 4(a)이고 제안 알고리듬을 적용할 수 있도록 right-to-left형태로 변형하면 그림 4(b)와 같다. Right-to-left 방식의 기본 원리는 m-ary 방식에서와 비슷하며 E를 window로 잡을 때의 위치값을 먼저 계산하고 후에 각 window에 해당하는 만큼 멱승을 수행한다. Right-to-left 방식의 while문 내의 T[e]=S×T[e] mod N과 S=S×S mod N에서 제안 알고리듬을 적용하면 모듈라 감소를 효과적으로 할 수 있다. 두번째 및 세번째 for문은 $A^E=T[1]\cdot T[2]^3\cdot\cdots\cdot T[2^{w-1}-1]^{2^{w-1}}$을 수행하는 과정이다.

계산량을 보면, 먼저 left-to-right 방식에서 평균적으로 필요한 전체 계산량은 2^{w-1} + ⌈ $\log_2E$⌉ -(w-1)+⌈ ⌈ $\log_2E$⌉ /(w+1)⌉ -1번의 모듈라 곱셈이다. 또한 2^{w-1} 개의 수를 저장할 메모리가 필요하다. 반면 제안 알고리듬을 적용하기 전의 right-to-left 방식의 while문 내에서 window 개수는 ⌈ $\log_2E/(w+1)$⌉ 이므로 ⌈ $\log_2E/(w+1)$⌉ 번의 곱셈과 ⌈ $\log_2E$⌉ 번의 제곱이 필요하다. 그러므로 평균적으로 ⌈ $\log_2E/(w+1)$⌉ +⌈ $\log_2E$⌉ +3(2^{w-1}-2)번의 곱셈이 필요하다. 예를 들어 512비트인 경우에는 윈도우 크기를 5로 하면 left-to-right 방식에서는 평균 곱셈수가 609번이고 right-to-left 방식에서는 평균 곱셈수가 640번이 된다.

제안한 알고리듬을 적용할 경우 필요한 작은 수 곱셈은 (⌈ $\log_2E$⌉ -⌈ $\log_2E/(w+1)$⌉) $(2n^2+n)$+(⌈ $\log_2E/(w+1)$⌉) $(3n^2+3n)$+3(2^{w-1}-2) $(2n^2+n)$이다. 512비트 멱승시 $(512-86)(2n^2+n)$ + $86(3n^2+3n)+42(2n^2+n)$이고 n=32라면 1245888번이 된다. 이것은 약 599번의 모듈라 곱셈을 수행하는 결과와 같다. 이는 left-to-right 이진 방식을 22.1% 개선한 것이 된다.

```
Window(A, E, N)
  T[0]=A ;
  Temp=A×A mod N;
  for i=1 to 2^(w-1)-1 step 1
      T[i]=T[i-1]×Temp mod N;
  S=1;
  i=k-1 ;
  while i≥0 do
        if E_i=1 then
            j=MAX(i-w+1,0) ;
            while E_j=0 do
                  j=j+1 ;
            e=(E_i...E_j)_2 /2 ;
            for k=j to i step 1
               S=S×S mod N;
            S=S×T[e] mod N;
            i=j-1 ;
        else
            S=S×S mod N;
            i=i-1;
  return (S);
```

(a) Left-to-right

```
Window(A, E, N)
  for i=1 to 2^(w-1)-1 step 1
       T[i]=1;
  S=A;
  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_j...E_i)_2 /2 ;
            T[e]=S×T[e] mod N;
            S=S×S mod N;
            for k=i+1 to j step 1
               S=S×S mod N;
            i=j+1 ;
        else
            S=S×S mod N;
            i=i+1;
  for i=2^(w-1)-2 i=1 step -1
    T[i]=T[i]×T[i+1]×T[i+1] mod N;
  for i=2^(w-1)-2 i=1 step -1
    T[2^(w-1)-1]=T[2^(w-1)-1]×T[i] mod N;
  return (T[2^(w-1)-1]) ;
```

(b) Right-to-left

Fig. 4. Window method.

Window 크기를 4로 하면 512 비트 멱승시 $(512-103)(2n^2+n)+ 103(3n^2+3n)+ 18(2n^2+n)$이고 n=32라면 1214464번이 된다. 이것은 약 584번의 모듈라 곱셈을 수행하는 결과와 같다. 이는 left-to-right 이진 방식을 24.0% 개선한 것이 된다. 결국 제안 알고리듬을 사용할 때에는 window 크기가 4일 때 최적이다. 이는 계산 측면에서도 효율적이며 메모리 사용 측면에서도 반정도 감소할 수 있어 효과적이다. 그 이유는 window 크기가 작으면 평균 window 개수는 증가하므로 window의 위치값을 계산하는 연산이 많아지는 반면 상대적으로 중간값 저장용 테이블이 감소하고 테이블간의 곱셈수를 줄여주기 때문이다.

4. 성능 및 효율성 분석

제안하는 모듈라 감소 알고리듬의 특징은 멱승 방식을 right-to-left 형태로 변환하면 동일한 하나의 변수에 대해 서로 다른 두개의 변수를 각각 모듈라 곱셈을 하는 성질을 이용하고 있다. 사용되는 공통 변수를 다른 변수들과 곱하여 모듈라 감소를 하는 과정에서 생기는 중첩되는 계산 부분을 생략할 수 있다는 것이다. 본 장에서는 제안 알고리듬을 기존의 모듈라 감소 알고리듬과 비교 분석하고 멱승 방식에 적용할 때의 효율성을 속도와 메모리 사용 측면에서 비교한다.

모듈라 감소 알고리듬으로는 classical 알고리듬[5, 6], Yang 알고리듬[7], Barrett 알고리듬[8], Tanaka-Okamoto 알고리듬[9] 그리고 현재까지 가장 고속으로 알려진 Montgomery 알고리듬[10, 11] 등이 있다. 한번의 모듈라 감소를 수행할 때 기존의 방식과 제안하는 알고리듬을 비교한 것이 표 1이다.

제안 알고리듬에서는 몫추정을 위한 나눗셈을 없애기 위해 모듈라 수의 최상위 한자리를 모두 1로 함을 가정한다.

Table 1. Comparison of modular reduction algorithms.

	Proposed	Classical	Barrett	Montgomery	T-O	Yang
Multiplication	n^2+n	n^2+2n	n^2+4n	n^2+n	n^2+2n	$1.25n^2$
Division	·	n	·	·	2	2n
Tables for k bits	·	·	1	1	n	·
Advantage	High speed, Simple Small memory size	Low speed	Simple	High speed Simple	High speed Simple	Small memory size
Disadvantage	Specified modulus	Complex		Precomputation Non-standard	Lookup table	Low speed Complex

Classical 알고리듬과 Yang 알고리듬에서는 일반적인 모듈라 수를 사용할 경우 나눗셈이 필요하고 구현이 복잡한 점이 단점으로 지적되고 있다. Barrett 알고리듬은 비교적 고속이면서 간단히 구현할 수 있다. Tanaka-Okamoto 방식은 고속이면서 구현도 용이하다. 그러나 이 알고리듬은 k비트의 사전 계산 테이블이 n개 사용된다는 것이 단점이다. 반면 Montgomery 알고리듬은 잉여류 변환을 이용한 비표준 연산법이나 현재까지 가장 빠른 모듈라 감소 알고리듬으로 알려져 있다.

반면 제안하는 알고리듬은 n+1자리에 대한 몫 추정과 곱셈이 필요하지만 모듈라 수를 특정하게 하면 나눗셈을 피할 수 있고 중간값을 저장하는 메모리의 크기는 최대 n+2자리만 소요된다. 이 경우 n+1 자리의 수를 모듈라 감소하는 방법 중 diminished radix 기법은 큰 수의 뺄셈을 덧셈으로 대치할 수 있어 효과적이다. 최상위 한자리수가 모두 1이 되는 특정한 모듈라 수를 구하는 문제는 RSA 암호시스템이나 ElGamal 형태의 암호시스템에서 어려운 문제는 아니며 안전도에도 별다른 영향이 없는 것으로 알려져 있다.

제안하는 모듈라 감소 알고리듬은 단순히 곱셈만을 수행할 때에는 Montgomery 방식과 속도면에서 비슷하다. 그러나 제안 알고리듬은 위에서 언급한 바와 같이 right-to-left 형태의 멱승 연산에 적용시에 효과적이다. 멱승 연산에 적용할 때의 비교 분석한 것은 표 2와 같다. 제안 알고리듬과 비교하기 위해 각 멱승 방식에 사용된 모듈라 감소 알고리듬은 Montgomery 알고리듬을 사용함을 전제로 하고 이진 방식을 기준으로 비교하였다. 표에서는 예로서 512비트의 A, E 및 N을 사용할 때 비교하였다. 여기서 $b=2^{16}$, n=32임을 가정하고 계산하였다. 제안 알고리듬을 m-ary 방식에 적용할 때에는 $m=2^3$을 사용하고 window 방식에서는 window 크기를 4로 하였다.

Table 2. Comparison of modular exponentiation methods using a new modular reduction algorithm(|E| =512).

Method	Binary		Binary redundant		m-ary		Window	
	L-to-R	Proposed	L-to-R	Proposed	L-to-R ($m=2^4$)	Proposed ($m=2^3$)	L-to-R |w|=5	Proposed |w|=4
Modular multiplication	768	642	682	598	641	600	609	584
Improved ratio(%)	0%	16.4%	11.2%	22.1%	16.5%	22.0%	20.7%	24.0%
Remarks	·	·	Inverse	Inverse	15 tables for 512 bits	7 tables for 512 bits	15 tables for 512 bits	15 tables for 512 bits

표에서 보는 바와 같이 제안한 알고리듬을 적용할 경우 기존의 멱승 방법보다 고속으로 수행할 수 있다. 이진 방식에서는 특별한 메모리가 필요하지 않으면서 간단하게 구현할 수 있어 IC카드 등과 같은 시스템에서 효과적이다. 이진 잉여 방식에 적용시는 역수 계산이 필요하지만 E를 이진 잉여 표현으로 바꾸면서 순차적으로 멱승을 수행할 수 있는 것이 효과적이다. m-ary 방식이나 window방식에 적용할 경우 속도 개선은 물론 최적으로 수행될 때 저장하는 테이블의 수가 거의 절반으로 감소시킬 수 있는 점이 특징이다.

5. 결 론

본 논문에서는 고속 멱승을 위한 새로운 모듈라 감소 알고리듬을 제안하였다. 이 알고리듬은 right-to-left 형태의 멱승 연산시 기존의 방식보다 고속 연산이 가능하다. 제안한 알고리듬은 몫 추정시 나눗셈 혹은 뺄셈을 피하기 위해 모듈라 수의 최상위 자리수가 모두 1이라는 가정이 있기는 하지만 이를 구현하는 데는 별 어려움이 없다. 특히 IC 카드와 같이 제한적인 메모리를 사용하는 경우에는 이진 방식을 적용하는 것이 효과적이다. 또한 m-ary나 window 방식에서는 기존 방식보다 사용 메모리를 절반 정도로 줄일 수 있는 장점이 있다. 그러므로 제안 모듈라 감소 알고리듬은 RSA나 ElGamal 형태의 암호시스템에 효과적으로 적용할 수 있다.

참 고 문 헌

[1] 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.

[2] 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.

[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] H. Morita and C. H. Yang, "A modular multiplication algorithm using lookahead determination," *IEICE Trans. Fundamentals,* Vol. E76-A, No. 1, Jan. 1993.

[8] 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.

[9] Tanaka K. and Okamoto E. "On modular exponentiation using a signal processor," *Natl. Conv. Conf. Rec. on Information and Systems, IEICE.* 15, Nov. 1987.

[10] Peter L. Montgomery, "Modular multiplication without trial division," *Math. of Comp.* Vol. 44, No.170, pp.519-521, April 1985.

[11] 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.

[12] C. N. Zhang, "An improved binary algorithm for RSA," *Computer Math. Applic.* Vol. 25, No. 6, pp. 15-24, 1993.

[13] S. B. Mohan and B. S. Adiga, "Fast algorithms for implementating RSA public key cryptosystem," *Electronics Letters*, Vol. 21, No. 7, p. 761, 1985.

[14] C. H. Lim and P. J. Lee, "Sparse RSA secret keys and their generation," *SAC'96,* pp. 117-131, 1996.