通信情報保護學會誌
第5卷 第1號, 1995. 3

멱승 알고리듬의 구현과 분석

황 효 선*, 임 채 훈**, 이 필 중**

요 약

멱승연산 알고리듬에는 사전계산없이 g와 n이 주어지면 g^n을 계산하는 것과 고정된 g에 대해 사전계산을 하여 그 결과를 메모리에 저장하여 두고 n이 주어지면 g^n을 계산하는 알고리듬이 있다. Binary method, window method 등의 알고리듬은 전자에 해당하는 것이고 BGMW method와 LL method 등은 후자에 해당하는 것이다. 이러한 여러 멱승연산 알고리듬을 PC 486과 586에서 C 언어로 구현하여 그 결과를 비교 분석하였다.

1. 서 론

모든 공개키 암호시스템은 멱승연산을 필요로 한다. 유한환 $\mathbf{Z}_N$을 기반으로 하는 공개키 암호시스템에서는 임의의 수 g, n에 대해 $g^n \bmod N$을 계산하는 모듈라 멱승연산이 필요하고, 유한체 또는 유한환에서 정의되는 타원곡선을 기반으로 하는 공개키 암호시스템에서는 타원곡선 위의 임의의 점 $\boldsymbol{P}$와 임의의 수 a에 대해 $a\boldsymbol{P}$를 계산하는 멱승연산(실제로 이 경우는 곱셈 연산이라 표현하는 것이 자연스럽다)이 필요하다. 원칙적으로 g^n은 g를 n번 반복하여 곱함으로써 계산할 수 있다. 하지만 n이 보통 512비트(10진수 155자리), 적어도 150비트(10진수 45자리) 임을 감안할 때, 현실적으로 위의 방법으로 멱승연산을 하는 것은 불가능하다. 또한 공개키 암호시스템에서 메세지의 암/복호화와 디지탈 서명의 서명/검증 과정의 속도는 멱승연산의 속도와 비례하기 때문에 효율적인 멱승연산 알고리듬이 필요하다. 일반적으로 멱승연산을 빠르게 구현하려는 시도에는 서로 다른 두 가지의 접근 방법이 있다. 첫번째 방법은 $\mathbf{Z}_N$에서 모듈라 곱셈 또는 타원곡선 위의 점들의 연산과 같은 기본연산의 속도를 향상시킴으로써 멱승연산의 속도를 향상시키는 것이다[1]. 두번째 방법은 적은 횟수의 기본연산으로 가능한 멱승연산 알고리듬을 개발하는 것이다. 다시 말해서 멱승연산 자체를 효율적으로 구현하는 알고리듬을 개발하려는 것이다. 궁극적으로 멱승연산을 효율적으로 구현하기 위해서는 위의 두 방법이 모두 이루어져야 한다. 본 논고에서는 위에서 두번째 시도, 즉 멱승연산 자체를 효율적으로 구현하는 알고리듬에 대해 언급하였다.

본 논고에서는 기존의 멱승연산 알고리듬을 그 특성에 따라 두 개로 분류하였다. 첫번째는 임의의 밑 g와 임의의 지수 n에 대해(모듈라 멱승에서

* 포항공과대학교 정보통신연구소
** 포항공과대학교 전자전기공학과

는 모듈라 수 N도 임의의 수임.) 멱승연산을 하는 알고리듬이다. 이 경우에는 사전계산(precompute)을 할 수 없기 때문에 일반적으로 $\log_2 n$번 이상의 제곱과 다수의 곱셈이 필요하다. 이러한 알고리듬들을 2장에서 설명한다. 두번째는 고정된 밑 g와 (모듈라 멱승에서는 모듈라 수 N도 고정됨) 임의의 지수 n에 대해 모듈라 멱승연산을 하는 알고리듬이다. 이 경우에는 정해진 값 g를 이용하여 사전계산을 해 둘 수 있기 때문에 사전계산된 값을 이용하면 매우 적은 횟수의 곱셈만으로 멱승연산을 수행할 수 있다. 이러한 알고리듬들을 3장에서 설명한다. 그리고 4장에서는 위의 알고리듬들을 PC에서 구현한 결과를 분석하였고, 5장에서는 결론을 내렸다.

본 논고 중의 컴퓨터 구현 결과는 $\mathbf{Z}_N$에서의 모듈라 곱셈을 기본연산으로 하여 멱승연산을 계산한 것이다. 그리고 알고리듬의 설명에서는 기본연산을 일반적인 곱셈(×)으로 표시하였다. 또 각 멱승연산 알고리듬이 필요로 하는 곱셈의 횟수를 셀때에는 제곱도 포함하였고, 제곱을 제외한 협의의 곱셈과 제곱을 포함한 광의의 곱셈을 혼용하여 표현한 경우도 있다.

2. 사전계산이 없는 멱승연산 알고리듬

이 장에서는 임의의 g와 임의의 n에 대해(모듈라 멱승의 경우 모듈라 수 N도 임의의 수임) 사전계산 없이 멱승 g^n을 계산하는 알고리듬을 알아본다.

2.1 Binary method (이진 방식)

가장 오래 전부터 알려진 멱승연산 알고리듬은 B.C. 200년경에 개발된 binary method이다[2]. 이 알고리듬은 repeated square and multiply 이라고도 하는데 이름에서 알 수 있듯이 제곱과 곱셈을 반복하여 멱승연산을 수행한다. 하나의 예로 binary method를 설명하겠다. n이 23인 경우 g^n을 계산하여 보자. 23을 2진수로 나타내면 $23 = 10111_{(2)}$이므로 다음의 식을 이용할 수 있다.

$$g^{23} = ((((g)^2)^2 \times g)^2 \times g)^2 \times g$$

이때 중간 계산 값을 보면 $g^{1_{(2)}}$, $g^{10_{(2)}}$, $g^{100_{(2)}}$, $g^{101_{(2)}}$, $g^{1010_{(2)}}$, $g^{1011_{(2)}}$, $g^{10110_{(2)}}$, $g^{10111_{(2)}}$이다. 위 중간 계산 값을 자세히 보면, g^i를 제곱하면 g^{2i}가 되어 2진수로 지수를 나타낼 경우 자리수가 한 자리 늘어나고 $g^i \times g$는 g^{i+1}이 되어 지수가 1 증가함을 알 수 있다. Binary method는 n을 2진수로 나타내어 그 최상위 자리의 1에 대해 g라 두고 이 값을 반복하여 제곱하면서 해당 비트가 1일 때만 g를 곱해준다. 이를 구체적으로 나타내면 아래와 같다. 먼저 n을 2진수로 나타내어서 $n = (n_{k-1}, n_{k-2}, \ldots, n_1, n_0)_{(2)}$이라 두자. 이 때 n_{k-1}는 1이다.

```
Binary_Exp(g,n)
    A := g ;
    for i=k-2 to 0 by -1
        A := A × A ;
        if n_i=1 then
            A := A × g ;
    return A ;
```

알고리듬 1 : Binary method

Binary method는 n이 k비트인 경우 k-1번의 제곱 계산과 $W_H(n)$-1번의 곱셈이 필요하다. 여기서 $W_H(n)$은 n의 Hamming weight라고 하고, n을 2진수로 표현하였을 때의 1의 갯수를 나타낸다. 하지만 임의의 n에 대해 $W_H(n)$을 미리 알 수는 없다. 일반적으로 n을 2진수로 나타내었을 때 각 자리수가 약 1/2의 확률로 1이 됨을 가정하면 $W_H(n) \approx \lceil \log_2 n \rceil / 2$이라고 둘 수 있다. 그러므로 binary method는 최대 $2k$-2번, 평균 $(3/2) \times (k-1)$번의 곱셈이 필요하다. 그리고 binary method는 중간 계산값 A를 저장하기 위한 것 외

의 여분의 메모리를 필요로 하지 않는다.

Binary method는 구현이 쉽고 여분의 메모리를 필요로 하지 않는 장점이 있지만 상대적으로 많은 횟수의 곱셈을 필요로하므로 멱승연산 속도가 느리다.

2.2 m-ary method(m진 방식)

Binary method에서는 지수 n을 2진수로 나타내고, 제곱과 곱셈을 이용하여 멱승연산을 수행하였다. 그러나 2진수가 아닌 3진수, 4진수 등 일반적인 진수에 대해 같은 방법으로 멱승연산을 할 수도 있다[2]. Binary method가 제곱(2승, 지수를 2배함)과 g를 곱하는 것(지수를 1 증가함)을 반복한 것과 같이 m-ary method는 m승 연산(지수를 m배함)과 적당한 g^j, $j = 1...,m-1$,를 곱하는 것(지수를 j 증가함)을 반복하여 멱승연산을 수행한다. 이 때 g^j는 필요할 때마다 계산하는 것보다 미리 $\{g^j : j = 1...m-1\}$를 계산하여 두고 이용한다. 그리고 A^m 계산은 A를 단순히 m번 반복하여 곱하기보다 효율적인 binary method를 이용한다. 구체적인 알고리듬을 보자. 먼저 n을 m진수로 나타내어서 $n = (n_{k-1}, n_{k-2}, \ldots, n_1, n_0)_{(m)}$이라 두자. 이 때 n_{k-1}는 0이 아니다. 그리고 g^j, $j = 1...m-1$, 값은 미리 계산하여 메모리에 저장하여 두었다.

```
m-ary_Exp(g,n)
        A := g ;
        for i=k-2 to 0 by -1
                A := A^m ;
                if n_i≠0 then
                        A := A × g^(n_i) ;
        return A ;
```

알고리듬 2 : m-ary method

위의 알고리듬을 수행하기 위해 필요한 곱셈의 횟수를 알아보자. 먼저 g^j, $j = 1 \ldots m-1$,를 계산해 두기 위해 $m-2$번의 곱셈이 필요하다. 또 $k-1$번 반복하는 for문에서는 A^m을 계산하기 위해 $\lceil \log_2(m+1) \rceil + W_H(m) - 2$번과 $A \times g^{n_i}$ 계산에서 한 번의 곱셈이 필요하다. 이때 k는 n을 m진수로 표현할 때의 자리수이므로 $\lceil \log_m(n+1) \rceil$이고, 결국 $m - 2 + (\lceil \log_m(n+1) \rceil - 1) \times (\lceil \log_2(m+1) \rceil + W_H(m) - 2 + 1)$번의 곱셈이 필요하다. 하지만 위에서 n_i가 0일 확률이 $1/m$이므로 $A \times g^{n_i}$ 계산은 매 m번에 한 번 정도는 하지 않아도 된다고 기대된다. 그러므로 평균적으로 $m - 2 + (\lceil \log_m(n+1) \rceil - 1) \times (\lceil \log_2(m+1) \rceil + W_H(m) - 2 + (m-1)/m)$번 곱셈이 필요하다. 그리고 g^j, $j = 1 \ldots m-1$, 값을 저장하기 위해 $m-1$개의 수를 저장할 메모리가 필요하다. 예로서 n이 512비트수인 경우에 필요한 곱셈의 횟수를 보자. 아래의 표에서 m이 2인 경우가 binary method이다.

표 1 m-ary method 곱셈의 횟수(512비트 지수)

m	$W_H(m)$	최대횟수	평균회수
2	1	1022	766.5
3	2	970	862.3
4	1	767	703.3
5	2	883	839.0
6	2	796	763.0
7	3	915	889.0
8	1	686	664.8
9	2	812	794.1
10	2	778	762.6
11	3	897	883.5
12	2	720	708.2
13	3	839	828.4
14	3	816	806.4
15	4	930	921.3
16	1	649	641.3
32	1	642	638.8

위의 표로부터 m의 Hamming weight $W_H(m)$이 작을수록 m-ary method가 효율적임을 알 수 있다. 특히 m이 2의 멱승일 때가 매우 효율적인데 이러한 m-ary method의 장점을 더욱 개선한 멱승연산 알고리듬이 2.3절의 window method이다.

일반적으로 g^m 계산을 쉽게 할 수 있는 특별한 방법이 없을 경우 m-ary method는 m이 4, 8, 16 등과 같이 Hamming weight가 작을 때에만 유용하다. 하지만 유한체 $GF(2^n)$에서 정의된 타원곡선 중 anomalous curve의 경우는 타원곡선이 정의된 유한체 $GF(2^n)$를 normal bases로 나타낸다면 임의의 점 P에 대해 $16P$를 매우 쉽게 계산할 수 있다[3]. 이러한 특별한 경우에는 16-ary method를 사용하면 멱승의 속도를 빠르게 할 수 있다.

m-ary method는 g에 관계되는 몇 개의 수(g^j, $j = 1 \ldots m-1$)를 미리 계산하여 두고 이를 이용하여 멱승연산을 binary method보다 효율적으로 구현하였다.

2.3 Window method

사전계산없이 멱승연산을 하는 경우 일반적으로 약 $\log_2 n$번의 제곱은 꼭 필요하다. 그러므로 사전계산이 없는 멱승연산 알고리듬에서는 약 $\log_2 n$번의 제곱 이외의 곱셈 횟수를 줄이는 것이 멱승연산을 빠르게 하기 위해 대단히 중요하다. Binary method와 같이 매번 하나의 비트만을 고려하여 곱셈을 하는 방법보다는 몇 개의 비트를 동시에 고려하여 곱셈을 한다면(이것은 m-ary method에 m이 2의 멱승인 4, 8, 16 등인 경우와 같다) 반드시 필요한 약 $\log_2 n$번의 제곱 이외에 곱셈은 줄일 수 있다. Window method는 g와 n이 주어지면 본격적인 멱승연산에 앞서 몇 개의 값을 미리 계산하여 저장함으로써 한 번의 곱셈으로 한 비트가 아닌 여러 개의 비트를 동시에 처리하는 것이 기본적인 idea이다. Window 크기를 w라 두면, 먼저 주어진 밑 g를 이용하여 g^1, g^2, ..., g^{2^w-1}을 계산하여 저장한다. 그리고 지수 n을 w비트씩 잘라서 반복적으로 w번의 제곱을 한 후 해당하는 w비트의 값 s에 따라 g^s를 곱해줌으로써 g^n을 계산할 수 있다. 위에서 설명한 것은 m이 2의 멱승인 m-ary method와 같다. 하지만 window method가 다른 점은 window를 잡는 방법이 일률적으로 w비트씩이 아니라는 것이다. window를 만들 때 window의 크기가 w 이하이면서 window의 첫 비트와 마지막 비트가 1이 되게 한다[2]. 이 idea를 적용하면 미리 계산하는 g^1, g^2, ..., g^{2^w-1}에서 지수가 짝수인 것들을 배제할 수 있다. 그 이유는 window의 마지막 비트가 항상 1이므로 짝수가 될 수 없기 때문이다. 또한 window의 첫 비트가 1이란 조건을 만족하기 위해 0인 비트가 계속되는 경우 이 0들을 빼고서 window를 구성한다. 그러므로 window의 갯수는 단순히 $\lceil \log_2 n \rceil / w \rceil$가 아니고, window들 사이에 생길 0의 갯수를 고려하여 약 $\lceil \log_2 n \rceil / (w+1) \rceil$이 된다. 결과적으로 window를 잡을 때 몇몇 조건을 줌으로써 미리 계산하는 과정과 본격적인 계산에서 필요한 곱셈의 횟수를 줄였다.

Window를 선택하는 방법을 구체적으로 설명하면 아래와 같다. 아래에서는 w를 4로 두었고 n은 262659474289536631839191이다. Window는 상위 자리수부터 잡는데 첫 비트와 끝 비트가 1이 되도록 하면서 크기가 4 이하가 되게 한다.

$$n = \underline{10110}\ 00\underline{111}\ 00\underline{100}\ 0000\underline{1}\ \underline{11010}$$
$$\underline{01010}\ 0\underline{1110}\ \underline{10100}\ 0000\underline{1}\ \underline{01111}$$
$$00000\ \underline{11111}\ 00\underline{110}\ 0\underline{1010}\ \underline{10111}$$

먼저 g가 주어지면 g^1, g^3, g^5, g^7, g^9, g^{11}, g^{13}, g^{15}를 계산하여 메모리에 보관한다. n에 대해 위에서 밑줄 그은 것과 같이 window를 만든다. 물론 위와 같은 window를 미리 만든 후에 멱승연산을 하는 것은 아니고 window를 만들면서 동시에 멱

승연산을 한다. 구체적인 방법을 설명해 보면 먼저 최초의 window 1011을 찾은 후 A에 $g^{1010_{(2)}}$를 대입한다. 다음으로 세 개의 0에 대해서는 $A := A^2$을 반복한다. 그리고 다음의 두번째 window 1101을 찾은 후 $A := A^2$을 이 window의 크기인 네 번만큼 반복하고 $A := A \times g^{1101_{(2)}}$을 한다. 이제 A에는 $g^{10110001101_{(2)}}$값이 저장되어 있다. 계속해서 두 개의 0에 대해 $A := A^2$을 두 번하고, 그 다음의 window는 1이므로 $A := A^2$과 $A := A \times g^{1_{(2)}}$을 하여 A에 $g^{1011000110100 1_{(2)}}$가 저장되게 한다. 같은 방법으로 마지막 비트까지 window를 만들면서 window 값 s에 해당하는 g^s를 A에 곱하는 것과 제곱을 반복하면 된다. 알고리듬은 아래와 같다. 이 때 $n = (n_{k-1}, n_{k-2}, \ldots, n_1, n_0)_{(2)}$이라 두고 n_{k-1}는 1이라고 가정하자.

```
Window_Exp(g,n)
    T[0] := g ;
    Temp[0] := g × g ;
    for i := 1 to 2^(w-1)-1 by 1
        T[i] := T[i-1] × Temp ;
    A := 1 ;
    i := k - 1 ;
    while i ≥ 0 do
        if n_i = 1 then
            j := MAX(i-w+1,0) ;
            while n_j = 0 do
                j := j + 1 ;
            exp = (e_i ··· e_j)_2 ;
            for k := j to i by 1
                A := A × A ;
            A := A × T[exp] ;
            i := j - 1 ;
        else
            A := A × A ;
    return A (= g^n) ;
```

알고리듬 3 : Window method

Window method로 멱승연산을 할 경우 $g^{홀수}$를 미리 계산하는 과정에서 2^{w-1}번의 곱셈이 필요하고 2^{w-1}개의 수를 저장해야 한다. 그리고 window의 평균 크기는 약 w - 1이므로 처음 w - 1 비트에 대해서는 제곱 계산이 필요없다. 그래서 약 $\lceil \log_2 n \rceil - (w-1)$번의 제곱 계산이 필요하다. 또한 예상되는 window의 갯수가 $\lceil \lceil \log_2 n \rceil / (w+1) \rceil$이므로 $\lceil \lceil \log_2 n \rceil / (w+1) \rceil$ - 1번의 곱셈이 필요하다. 정리하면 평균적으로 $2^{w-1} + \lceil \log_2 n \rceil - (w-1) + \lceil \lceil \log_2 n \rceil / (w+1) \rceil$ - 1번의 곱셈과 2^{w-1}개의 수를 저장할 메모리가 필요하다.

메모리가 충분한 경우 window method를 가장 효율적으로 구현하는 w값을 찾아보자. 이 값은 n의 비트 수에 의해 결정되는데, window method가 필요로 하는 곱셈연산의 횟수를 w에 대해 미분하여 최적의 w를 결정할 수도 있지만 실제로는 임의로 w를 연속적으로 대입하여 곱셈의 횟수를 최소화하는 w를 찾으면 된다. 일례로 n이 512비트 수인 경우에 window method가 필요로 하는 곱셈의 횟수는 $2^{w-1} + 512 - (w-1) + \lceil 521/(w+1) \rceil - 1$이다. $\lceil 521/(w+1) \rceil$를 $521/(w+1)$라 두고 이 식을 미분하면 $2^{w-1} \times \log_e 2 - 1 - 521/(w+1)^2$이다. 이 미분된 식을 0이 되게 하는 w를 구하면 약 5.31809717이다. 그러므로 앞뒤 정수인 5와 6을 직접 처음의 식에 대입하면 각각 609와 612가 되는데, 이 결과로 w가 5일 때 최소의 곱셈이 필요함을 알 수 있다. 또한 n이 1024비트 수인 경우에 필요한 곱셈의 횟수는 5, 6, 7을 직접 위의 식에 대입하면 1206, 1197, 1209를 얻을 수 있는데 이 결과로부터 w가 6일 때 최소의 곱셈이 필요함을 알 수 있다. 당연하게도 n이 커질수록 window의 크기를 늘리는 것이 효율적이다.

Window method는 m-ary method를 개선한 멱승연산 알고리듬으로 비교적 구현이 용이하고, 구현 속도도 사전계산이 없는 멱승연산 알고리듬 중에서는 매우 빠르다. 그리고 계산 중에

window의 크기가 5인 경우에는 15개, window의 크기가 6인 경우에는 31개의 수를 저장할 메모리가 필요하다.

3. 사전계산을 이용하는 멱승연산 알고리듬

이 장에서는 밑 g가 고정되어 있고 지수 n만이 변하는 경우(모듈라 멱승의 경우 N도 고정됨) g를 이용하여 사전계산을 하여 저장한 후 n이 주어지면 저장된 값을 이용하여 멱승 계산을 효율적으로 구현하는 알고리듬을 소개한다.

3.1 기본적인 idea

밑 g가 고정되어 있을 경우 멱승연산을 빠르게 하는 방법으로 가장 쉽게 생각할 수 있는 것은 n이 k비트수인 경우 $\{g^{2^i} : i = 0...k-1\}$을 계산하여 저장하여 두고, n이 주어지면 n의 i번째 비트가 1일 때만 계산해 둔 g^{2^i}를 곱합으로써 g^n을 계산하는 것이다. 위의 방법은 n이 512비트수 일때 512개의 수를 저장해야 하고 필요한 곱셈의 횟수도 약 $W_H(n) - 1 \approx 512/2 - 1 = 255$번이다. 이 방법을 **사전계산을 이용한 b진수 멱승연산 알고리듬**이라고 하자. 만약 곱셈의 횟수를 줄여 멱승을 빠르게 계산하려면 적당한 양수 b에 대해 $\{g^{ib^j} : i = 0...k-1, j = 1...b-1\}$을 사전에 계산하여 저장한 후 n이 주어지면 n을 b진수로 나타내고 n의 i번째 자리수가 j이면 g^{ib^j}를 곱함으로써 g^n을 계산한다. 이 방법을 **사전계산을 이용한 빠른 b진수 멱승연산 알고리듬**이라고 하자. 반면 곱셈의 횟수는 늘어나서 속도가 느려지더라도 메모리를 적게 사용하면서 멱승연산을 하려면 역시 적당한 양수 b에 대해 $\{g^{b^i} : i = 0...k-1\}$을 사전에 계산하여 저장한 후 n이 주어지면 n을 b진수로 나타내고 n의 i번째 자리수가 j이면 g^{b^i}를 j번 반복하여 곱함으로써 g^n을 계산한다. 이 방법을 **사전계산을 이용한 느린 b진수 멱승연산 알고리듬**이라고 하자. **사전계산을 이용한 b진수 멱승연산 알고리듬**은 **사전계산을 이용한 빠른 b진수 멱승연산 알고리듬**과 **사전계산을 이용한 느린 b진수 멱승연산 알고리듬**에서 b가 2인 경우이다. **사전계산을 이용한 빠른 b진수 멱승연산 알고리듬**은 최대 k-2번, 평균 $(b-1)/b \times (k-2)$번의 곱셈이 필요하고 $(k-1) \times (b-1)$개의 수를 저장해야 한다. 또한 **사전계산을 이용한 느린 b진수 멱승연산 알고리듬**은 최대 $(b-1) \times (k-1)-1$번, 평균 $(b-1)/2 \times (k-1)-1$번의 곱셈이 필요하고 $(k-1)$개의 수를 저장해야 한다.

이러한 사전계산을 이용한 두 알고리듬은 idea가 간단하고 구현도 쉽지만 다음 절에서 설명할 BGMW method나 LL method과 비교해 보면 필요한 곱셈의 횟수와 메모리 요구량이 너무 많다. 다음 절에서 소개될 방법들은 기본적으로 **사전계산을 이용하는(빠른/느린) b진수 멱승연산 알고리듬**을 개선한 것이다.

3.2 BGMW Method

Brickell 등은 밑 g가 고정되어 있을 경우 메모리 요구량은 **사전계산을 이용하는 느린 b진수 멱승연산 알고리듬**과 같으면서 필요한 곱셈의 횟수는 **사전계산을 이용하는 빠른 b진수 멱승연산 알고리듬**보다 조금(b번) 많은 멱승연산 알고리듬을 개발하였다[4]. 이 알고리듬은 n을 b진수로 나타내고 멱승연산을 하는 일반적인 방법에서도 유용하지만, n을 basic digit set을 이용한 b진수로 나타내고 멱승연산을 하는 경우에도 유용하다. 구체적인 멱승 방법은 아래와 같다.

밑 g와 지수 n이 주어졌을 때 적당한 k개의 수 $x_0, x_1, x_2, \ldots, x_{k-1}$에 대해 지수 n이

$$n = \sum_{i=0}^{k-1} a_i x_i, \quad a_i \in [0,h) \quad \text{for } i = 0, \ldots, k-1$$

을 만족한다고 하자. 만약 $\{g^{X_i} : i = 0...k-1\}$을 사전계산하여 메모리에 저장하여 두었다면, 다음의 식

$$g^n = \prod_{i=0}^{k-1} g^{a_i X_i} = \prod_{i=0}^{k-1} (g^{X_i})^{a_i} = \prod_{d=1}^{h} (\prod_{a_i=d} g^{X_i})^d = \prod_{d=1}^{h} C_d^d$$
$$= C_h \times (C_h \times C_{h-1}) \times (C_h \times C_{h-1} \times C_{h-2}) \times$$
$$\ldots (C_h \times C_{h-1} \times \ldots \times C_2 \times C_1)$$

을 이용하여 g^n을 계산할 수 있다. 위에서 $C_d = \prod_{a_i=d} g^{X_i}$이다. 구체적인 알고리듬은 아래와 같다.

```
BGMW_Exp(g,n)
    B := ∏_{a_i=h} g^{X_i} ;
    A := B ;
    for d = h - 1 to 1 by -1
        B := B × ∏_{a_i=d} g^{X_i} ;
        A := A × B ;
    return A (= g^n) ;
```

알고리듬 4 : BGMW method

위의 알고리듬은 최대 $k+h-2$번의 곱셈만이 필요하다. 위의 알고리듬이 적은 횟수의 곱셈만으로 멱승연산을 할 수 있는 이유를 보자. 먼저 지수 a_i가 일치하는 g^{X_i}들을 모두 곱하여 하나의 수 C_d로 만든 후에 d승을 하였다. 또한 C_d의 d승도 각각의 C_d(= 1, 2, ..., h)에 대해 따로 d승을 하지 않고 C_h부터 C_1까지를 차례대로 B에 곱하면서 그 중간값 B를 A에 다시 곱하였다. 즉 C_h의 경우 B에 포함되어 h번 반복하여 A에 곱해짐으로써 C_h^h이 계산되었다.

3.2.1 일반적인 b진수 이용

양수 b가 주어지면 n을

$$n = \sum_{i=0}^{k-1} a_i b^i, \quad a_i \in [0,h] \quad \text{for } i = 0, \ldots, k-1$$

와 같이 b진수로 나타낼 수 있다. 따라서 g가 주어지면 $\{g^{b^i} : i = 0...k-1\}$을 미리 계산해 두고 앞에서 설명한 idea로 멱승연산을 할 수 있다. 이때 $h = b - 1$이고, k개의 수를 저장하기 위한 메모리가 필요하다.

n이 512비트 수인 경우 위의 알고리듬을 구현하기 위해 필요한 곱셈의 횟수를 최소화하는 b의 값을 찾아보자. 이 경우 $k = \lceil \log_b 2^{512} \rceil$, $h = b - 1$이므로 멱승 계산시 필요한 곱셈의 횟수는 $k + h - 2 = \lceil \log_b 2^{512} \rceil + b - 3$이다. 하지만 a_i가 $1/b$의 확률로 0이라고 가정하면 필요한 곱셈의 평균 횟수는 $(b-1)/b \times \lceil \log_b 2^{512} \rceil + b - 3$이다. 이때에도 위의 식을 미분하거나, 또는 단순히 몇 개의 b를 연속적으로 대입하여 곱셈을 최소화하는 b를 찾을 수 있다. 찾은 최적의 b는 26이다. 이 때 최대 132번, 평균 127.81번의 곱셈이 필요하다. 여기서 최대 곱셈의 횟수는 모든 a_i 값이 0이 아닌 경우에 필요한 곱셈의 횟수이다. 또 사전계산된 값 109개를 저장하기 위해 6976byte가 필요하다. 하지만 512비트수 n을 26진수로 바꾸는 과정에서 시간이 걸리는 것을 고려하여 b를 32로 선택할 수도 있다. b가 32이면 n을 32진수로 바꾸는 것이 단지 n을 5비트씩 취하기만 하면 되므로 진수 변환에 필요한 시간이 많이 단축된다. 이 경우에는 최고 132번, 평균 128.78번의 곱셈이 필요하고, 103개의 수를 저장하기 위해 6592byte의 메모리가 필요하다. b가 26인 경우와 32인 경우 중 어느 방법이 더 빠른 속도로 멱승연산을 하는지 알기 위해서는 구현하여 이를 실행시켜보아야 한다. 이론적으로는 $b = 26$인 경우가 평균적으로 한 번의 곱셈이 적게 필요하므로 빠르지만, $b = 32$인 경우는 n의 진수 변환 과정이 없는 것이 장점이다.

3.2.2 Basic digit set을 이용한 b진수

여기서는 b가 주어졌을 때, b에 대한 basic digit set의 엄밀한 정의와 만드는 방법에 대한 구

체적인 설명은 생략하겠다[5]. 먼저 어떤 set M과 수 h에 대해

$$D(M,h) = \{sm : m \in M,\ s = 0, 1, \ldots, h\}$$

가

모든 $a \in [0,b-1]$에 대해
$s'm' \in D(M,h)$ s.t.
$a = s'm' \pmod b$가 있다.

을 만족하면 $D(M,h)$를 b에 대한 basic digit set이라 한다. 일반적인 b진수가 $\{0, 1, \ldots, b-1\}$로 모든 수를 표현할 수 있는 것처럼 우리는 임의의 수 n을 $D(M,h)$의 수를 이용한 b진수로 표현할 수 있다. 즉 임의의 n을

$$n = \sum_{i=0}^{k-1} d_i b^i,$$

$$d_i = s_i m_i \in D(M,h) \quad \text{for } i = 0, \ldots, k-1$$

로 나타낼 수 있다. 여기서 b^i를 위의 BGMW 알고리듬에서 x_i에 대응하면 멱승연산을 할 수 있다.

이해를 돕기 위해 basic digit set을 예를 들어 설명해 보자. 첫번째 예는 $M = \{\pm1, \pm2\}$이고 $h = 17$, $b = 53$인 경우이다. 이 때 모든 $\alpha = 0, 1, \ldots, 52$에 대해 $\alpha \equiv \beta \pmod{53}$을 만족하는 β가 집합

$$\begin{aligned} D(M,h) &= \{sm : m \in \{\pm1, \pm2\}, \\ &\qquad s = 0, 1, \ldots, 17\} \\ &= \{\pm1, \pm2, \pm3, \pm4, \ldots, \pm15, \pm16, \\ &\quad \pm17, \pm18, \pm20, \pm22, \pm24, \\ &\quad \pm26, \pm28, \pm30, \pm32, \pm34\} \end{aligned}$$

에 존재한다. 그러므로 $k = \lceil \log_b n \rceil$에 대해 n을

$$n = \sum_{i=0}^{k-1} d_i b^i, \quad d_i \in D[M,h] \quad \text{for } i = 0, \ldots, k-1$$

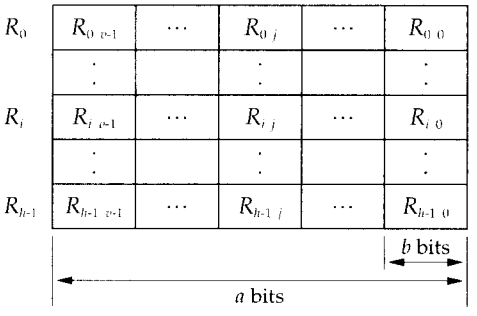
와 같이 표현할 수 있다. 이 때 n은 b진수로 k자리 수이지만 basic digit set으로 표현할 경우에는 최상위 자리에서 자리올림이 일어날 수 있기 때문에 $k+1$자리로 나타내어야 한다. 그러므로 주어진 g에 대해 $\{g^{\pm1\times b^i}, g^{\pm2\times b^i}, g^{\pm1\times b^k} : i = 0 \ldots k-1\}$을 사전계산해 둔다면 g^n을 쉽게 계산할 수 있다. 특히 $n = 512$비트 수인 경우, $k = 90$, $h = 17$이므로 최대 106번의 곱셈이 필요하고, d_i가 1/53의 확률로 0이 됨을 가정하면 평균적으로 104.28번의 곱셈이 필요하다. 그리고 사전계산된 362개의 수를 저장하기 위해 2398byte의 메모리가 필요하다.

또다른 예로 $M = \{\pm1, \pm2, \pm23\}$이고 $h = 16$, $b = 67$인 경우가 있다. 이 때에는 $n = 512$비트이면, $k = 86$, $h = 16$이므로 최대 100번의 곱셈과 사전계산된 512개의 수를 저장하기 위한 32768byte의 메모리가 필요하다.

3.3 LL method

Window method($w = 5$)나 BGMW method($b = 32$)는 연속된 5개의 비트를 동시에 처리하여 속도 향상을 꾀하였다. 하지만 LL method는 연속되지 않은 여러 개의 비트를 동시에 처리하여 속도 향상을 하였다[6].

먼저 k비트수 n이 주어지면 $a = \lceil n/k \rceil$ 비트인 블럭 h개로 나누어 $R_0, R_1, \ldots R_{h-1}$이라 하자. 또 이 h개의 각 블럭을 $b = \lceil a/v \rceil$ 비트인 블럭으로 나누어 $R_{i0}, R_{i1}, \ldots, R_{iv-1}$이라 하자.



이제 $g_i = g^{m_i}$를 계산하고 이를 이용하여 아래의 값을 사전계산하여 저장하자.

$$G[0][I] = g_{h-1}^{e_{h-1}} g_{h-2}^{e_{h-2}} \cdots g_0^{e_0},$$
$$I = (e_{h-1}\, e_{h-2} \cdots e_0)_{(2)}$$
$$G[j][I] = (G[j-1][I])^{2^h} = (G[0][I])^{2^{jh}}.$$

그러면 g^n은 다음과 같이 계산할 수 있다.

$$g^n = \prod_{i=0}^{h-1} g_i^{R_i} = \prod_{i=0}^{h-1} \prod_{j=0}^{v-1} (g_i^{2^{jh}})^{R_{ij}}$$
$$= \prod_{i=0}^{h-1} \prod_{j=0}^{v-1} \prod_{k=0}^{b-1} ((g_i^{2^{jh}})^{R_{ij}^k})^{2^k} = \prod_{k=0}^{b-1} (\prod_{j=0}^{v-1} \prod_{i=0}^{h-1} (g_i^{2^{jh}})^{R_{ij}^k})^{2^k}$$
$$= \prod_{k=0}^{b-1} (\prod_{j=0}^{v-1} G[j][I_{jk}])^{2^k}$$

위에서 $R_{ij} = (R_{ij}^{b-1} R_{ij}^{b-2} \cdots R_{ij}^{0})_{(2)}$이고 $I_{jk} = (R_{h-1j}^{k} R_{h-2j}^{k} \cdots R_{0j}^{k})_{(2)}$이다. 이를 구체적으로 나타내면 아래와 같다.

```
LL_Exp(g,n)
    A := 1 ;
    for k := b - 1 to 0 by -1
        A := A × A ;
        for j := v - 1 to 0 by -1
        A := A × G[j][I_jk] ;
    return A (= g^n) ;
```

알고리듬 5 : LL method

위의 멱승연산에는 최대 $a + b - 2$번의 곱셈연산이 필요하고 $(2^h - 1) \times v$개의 사전계산된 값을 저장해야 한다.

n이 512비트인 경우 LL method를 구현하기 위해 h와 v를 정해보자. 여기서는 메모리 요구량이 적당한 두 경우를 설명하겠다.

Case 1은 h와 v를 모두 5로 하였다. 그리고 b는 주로 21이고 19와 18인 block도 있다. 이 경우 $(2^h - 1) \times v = 31 \times 5 = 155$개의 수를 저장할 메모리가 필요하다. 그리고 필요한 곱셈의 횟수는 최대 $a + b - 2 = 103 + 21 - 2 = 122$번이고 평균 $(21 \times 4 \times 31/32 + 18 \times 31/32 + 1 \times 1/4) + 21 - 2 = 118.1$번이다. Case 2는 앞에서 알고리듬을 설명한 것과는 달라 보이지만 같은 원리로 설명된다. 단 마지막 두 비트를 적당히 처리하기만 하면 된다. 이 경우에는 마지막 두 비트를 처리하기 위한 하나의 수를 포함하여 $(2^5 - 1) \times 3 + 1 = 94$개의 수를 저장할 메모리가 필요하다. 그리고 필요한 곱셈의 횟수는 마지막 두 비트를 처리하기 위한 두 번의 곱셈을 포함하여 최대 $a + b - 2 + 2 = 102 + 34 - 2 = 136$번이고 평균 $102 \times 31/32 + 34 - 2 = 132.81$번이다.

19	21	21	21	21
19	21	21	21	21
18	21	21	21	21
18	21	21	21	21
18	21	21	21	21

Case 1 : 5×5

34	34	34
34	34	34
34	34	34
34	34	34
34	34	34
		2

Case 2 : 5×3

4. 컴퓨터 구현 결과와 비교 분석

멱승연산 알고리듬을 PC에서 C 언어로 구현하였다. 구현한 멱승연산 알고리듬은 binary method, window method, BGMW method에 b가 26과 32인 경우 그리고 LL method에서 5×3인 경우이다. 이 때 멱승연산을 위한 기본연산은 모듈라 곱셈이다. 즉 모듈라 멱승 $g^n \bmod N$을 계산하였고, N은 512비트수이고 g와 n은 N 미만의 임의의 수이다. C 언어로 구현된 알고리듬은 Microsoft C 7.00과 WATCOM C 9.5로 컴파일하여 PC 486/50MHz와 PC Pentium(586)/60MHz에서 수행하였다.

기본연산인 모듈라 곱셈은 Classical algorithm, Barrett algorithm, Montgomery algorithm, Selby algorithm, Takagi algorithm, Yang algorithm으로 구현하였다[11].

표 2는 이 여섯 모듈라 곱셈을 이용하여 binary method로 모듈라 멱승연산을 구현한 결과이다. 다른 멱승연산 알고리듬에 대해서도 각 모듈라 곱셈 연산 알고리듬 간의 상대적인 속도비는 비슷하다. 이 때 표 2에서 WATCOM C 9.5로 컴파일하여 PC에서 수행한 것은 그 결과가 예상과 일치한다. 그러나 Microsoft C 7.00으로 컴파일하여 PC(특히 586/60MHz)에서 수행한 것은 그 결과가 예상과 일치하지 않았다. 제일 빠를 것으로 기대한 Montgomery algorithm으로 모듈라 곱셈을 구현한 경우보다 Barrett algorithm으로 모듈라 곱셈을 구현한 경우가 더 빠르게 모듈라 멱승연산을 수행하였다. 이에 위 두 모듈라 곱셈 알고리듬을 Microsoft C 7.00으로 컴파일하여 PC 586에서 수행하여 속도를 확인하였다. 그 결과 모듈라 곱셈의 속도도 Barrett algorithm이 더 빨랐다. Microsoft C 7.00으로 컴파일하여 PC 286/12MHz, 386/20MHz, 486/50MHz에서 수행한 결과는 항상 Montgomery algorithm으로 모듈라 곱셈을 구현한 경우가 더 빠른 수행 속도를 보였다[1]. PC 586에서 이와 같은 의외의 결과가 생긴 것은 컴파일러와 하드웨어 간에 특별한 관계가 있을 것이라고 추측하지만 정확한 이유는 알 수 없었다.

표 4는 여섯 모듈라 곱셈 알고리듬 중 속도가 빠른 Barrett algorithm과 Montgomery algorithm에 대해서 binary method, window method, BGMW method에서 b가 26과 32인 경우 그리고 LL method에서 5×3인 경우를 구현한 결과이다. 표 4에서 '/'의 좌측이 Barrett algorithm, '/'의 우측이 Montgomery algorithm으로 구현한 모듈라 곱셈을 이용한 멱승연산의 수행시간이다. 그리고 표 3은 각 멱승 알고리듬이 필요로 하는 곱셈과 제곱의 평균 횟수와 메모리 요구량을 나타내었다. 곱셈의 횟수는 제곱을 포함한 전체 곱셈의 횟수이고 제곱은 그 중 제곱의 횟수만을 나타낸 것이다. 그리고 위의 횟수에는 사전계산에 필요한 곱셈의 횟수는 제외되어 있다. 즉 사전계산된 정보가 있을 때 이를 이용하여 멱승연산을 수행하는 과정에서 필요한 곱셈의 횟수이다. 메모리 요구량에는 저장해야 하는 512비트 수의 갯수를 나타내었다. 여기서 RAM은 프로그램 수행 중에 계산, 저장하여 이용하는 512비트 수의 갯수를, ROM은 사전계산하여 저장해 둔 512비트 수의 갯수를 나타낸다. 그리고 입력 g와 출력 g'', 그리고 계산 중에 일시 저장해야 하는 수를 위해 3 ~ 4개의 512비트 수를 저장하기 위한 메모리도 필요한데 이 갯수는 괄호 안에 넣었다. 실제 BGMW method는 n을 26 또는 32진수로 변환하여 저장해 두면서 위에서 언급한 512비트 수를 저장하기 위한 것 이외의 메모리도 필요로 하지만 이 메모리는 위에서는 포함되지 않았다.

표 4의 결과는 표3의 사실과 일치한다. LL method(5×3)가 BGMW 32보다 곱셈의 횟수가 많음에도 불구하고 더 빠른 것은 곱셈 중에 제곱이 33번이나 포함되어 있기 때문이다. 일반적으로 제곱은 곱셈보다 약 25% 정도 빠르다. 그리고 BGMW 26과 BGMW 32의 경우는 본문에서 언급한 바와 같이 n을 26진수로 바꾸는 시간이 한 번의 곱셈을 하는 시간보다 더 많이 걸렸음을 알 수 있다. 다음으로 컴파일러와 PC 기종에 따른 결과를 비교해 보자. 먼저 컴파일러나 PC 기종에 관계없이 각 멱승연산 알고리듬의 상대적인 속도비는 비슷하다. 그러나 PC 486/50MHz에서는 Microsoft C 7.00으로 컴파일한 것이 WATCOM C 9.5로 컴파일한 것보다 빠르지만 PC 586/60MHz에서는 그 반대이다. 이는 Microsoft C 7.00는 16비트 컴파일러이고 WATCOM C 9.5는 32비트 컴파일러라는 것과 PC 486/50MHz과 PC 586/60MHz의 하드웨어의 차이에 기인한 것임을 추측할 수 있다.

논문 [7]에서는 IBM PC 486 DX-66MHz에서 Turbo C 2.0으로 모듈라 멱승연산을 구현한 결과가 있다. Binary method는 약 15.05초, Window method는 약 11.95초가 걸려 본 논고에서 PC 486/50MHz로 수행한 결과와는 약 10배 정도의 차이는 있지만 두 멱승연산의 상대적인

속도비는 본 논고의 결과와 비슷하다. 위 논문의 결과가 매우 느린 이유는 컴파일러의 선택에 있다고 생각된다. Microsoft C 7.00과 WATCOM C 9.5는 최적화 기능이 매우 우수한 컴파일러이다.

논문 [8]에서는 PC 386/33MHz에서 WATCOM C/386 9.0으로 모듈라 멱승연산을 구현한 결과가 있다. 모듈라 곱셈은 Classical algorithm, Barrett algorithm, Montgomery algorithm으로 구현하였고 멱승은 16-ary method로 구현하였다. N이 512비트, n이 512비트인 경우 Classical algorithm에서 2.95초, Barrett algorithm에서 2.83초, Montgomery algorithm에서 2.55초의 수행시간을 보였다. 기종이 다르므로 본 논고의 결과와 엄밀한 비교는 할 수 없지만 각 모듈라 곱셈 알고리듬 간의 상대적인 속도비가 본 논고에서 WATCOM C 9.5로 컴파일하여 PC 486에서 수행한 것과 비슷하다.

아래의 표에 실린 결과는 각 멱승연산을 100회 수행하여 그 수행 시간의 평균과 평균에 대한 표준편차의 백분율이다. 일례로 평균이 1882msec, 표준편차가 71.516인 경우 1882(3.8)이라고 나타내었다.

5. 결 론

공개키 암호시스템의 구현 속도는 멱승연산의 속도와 매우 밀접한 관계가 있기 때문에 멱승연산을 효율적으로 구현하는 것은 공개키 암호시스템을 효율적으로 구현하기 위해 필수적이다. PC로

표 2 모듈라 곱셈에 따른 결과

Binary	Microsoft C		WATCOM C	
	PC 486	PC 586	PC 486	PC 586
Classical	1882(3.8)	1009(3.8)	1629(4.9)	717(4.9)
Barrett	1402(2.4)	647(4.5)	1526(2.1)	586(4.5)
Montgomery	1402(2.1)	698(3.8)	1344(2.5)	560(4.4)
selby	1667(2.2)	831(2.8)	1646(2.1)	805(3.4)
Takagi	1618(2.7)	837(3.4)	1454(2.2)	623(4.4)
Yang	2231(3.7)	1139(3.9)	1973(3.9)	851(8.9)

단위 : msec/1회(표준편차의 백분율)

표 3 평균 곱셈 횟수와 메모리 요구량

N=512비트	곱셈(제곱포함)/제곱 평균 횟수 -사전계산 제외	RAM/ROM 요구량 - 단위 : 수의 갯수
Binary	766.00/511	0(4)/ 0
Window	609.00/508	15(4)/ 0
BGMW 26	127.81/ 0	0(3)/109
BGMW 32	128.78/ 0	0(3)/103
LL 3×5	132.81/ 33	0(3)/ 94

표 4 멱승연산 알고리듬의 구현 속도

N=512비트 Barr./Montg.	Microsoft C		WATCOM C	
	PC 486	PC 586	PC 486	PC 586
Binary	1402(2.4)/1402(2.1)	647(4.5)/698(3.8)	1526(2.1)/1344(2.5)	586(4.5)/560(4.4)
Window	1098(1.6)/1099(1.6)	502(3.8)/544(2.8)	1197(1.9)/1054(2.1)	456(5.6)/440(1.7)
BGMW 26	280(1.7)/285(2.0)	135(2.3)/144(2.1)	294(1.7)/269(1.9)	117(2.7)/114(2.5)
BGMW 32	263(1.9)/266(1.8)	124(2.3)/134(2.1)	282(1.7)/255(1.8)	110(4.2)/108(2.6)
LL 3×5	260(1.6)/262(1.8)	121(2.7)/130(2.3)	279(1.6)/252(1.6)	108(2.6)/106(2.6)

단위 : msec/1회(표준편차의 백분율)

모듈라 멱승연산을 구현한 결과를 보면 모듈라 수 N과 지수 n이 512비트인 경우 PC 486에서 약 1초, 특히 밑 g에 대해 사전계산을 할 수 있다면 약 0.3초 이내에 할 수 있다. 이는 공개키 암호시스템을 PC 486에서 구현할 경우 디지탈 서명 등을 수 초 내에 할 수 있음을 말한다. 또한 지수 512비트보다 작은 경우 그 구현 시간은 표 3에서보다 우수할 것이므로 PC에서 공개키 암호시스템을 구현할 경우 시간적인 제약은 별로 없을 것이라고 할 수 있다.

참 고 문 헌

[1] 황효선, 임채훈, 이필중, *PC에서 모듈라 곱셈 연산의 구현과 비교 분석*, 통신정보보호학회지 4권 3호, pp. 34-59, 1994. 9.

[2] D. E. Knuth, *The Art of Programming, Vol. 2 : Seminumerical Algorithms*, 2nd Ed. Chap. 4, Addison-Wesley, 1981.

[3] Neal Koblitz, *CM-Curve with Good Cryptographic Properties*, Advances in Cryptology-CRYPTO '91, Lecture Notes in Computer Science 576, pp. 279-287, Springer-Verlag(1992).

[4] E. F. Brickell, D. M. Gordon, K. S. McCurley and D. Wilson, *Fast Exponentiation with Precomputation*, Proc. Eurocrypt '92, Balatonfured, Hungary(1992), pp. 193-201.

[5] David W. Matula, *Basic Digit Set for Radix Representation*, Journal of the ACM, 29(1982), pp. 1131-1143.

[6] Chae Hoon Lim and Pil Joong Lee, *More Flexible Exponentiation with Precomputation*, Advances in Cryptology-CRYPTO '94, Lecture Notes in Computer Science 839, pp. 95-107, Springer-Verlag(1994).

[7] 최성민, 김병천, 정지원, 원동호, *모듈라 멱승연산 알고리즘의 구현에 관한 연구*, 한국통신정보보호학회 종합학술발표회 논문집 Vol. 3, No. 1, pp. 108-118, 1993.

[8] Antoon Bosselaers, Rene Govaerts and Joos Vandewalle, *Comparision of three modular reduction functions*, PreProceedings of Crypto '93, Santa Barbara, California, August 1993.

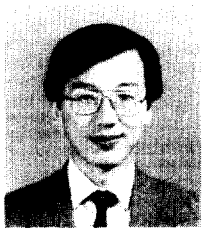
□ 著者紹介

황 효 선 (정회원)

1969년생
1992년 2월 포항공과대학 수학과 학사
1994년 2월 포항공과대학 수학과 석사
1994년 ~ 현재 포항공과대학교 정보통신연구소 전임연구원

임 채 훈 (정회원)

1963년생
1989년 2월 한국 데이타 통신(주) 기술본부 근무
1989년 3월 서울 대학교 전자공학과 학사
1992년 2월 포항공과대학 전자전기공학과 석사
1994년 ~ 현재 포항공과대학 전자전기공학과 박사과정 재학중



이 필 중 (李 弼 中) 종신회원

1951년 12월 30일생
1974년 2월 서울대학교 전자공학과 학사
1977년 2월 서울대학교 전자공학과 석사
1982년 6월 U.C.L.A. System Science, Engineer
1985년 6월 U.C.L.A. Electrical Engineering, Ph.D.
1980년 6월 ~ 1985년 8월 Jet Propulsion Laboratory, Senior Engineer
1985년 8월 ~ 1990년 2월 Bell Communications Research, M.T.S.
1990년 2월 ~ 현재 포항공과대학 전자전기공학과, 부교수