소수의 효율적 생성[1]

심 상규°, 이 은정*, 이 필중
포항공과대학교 전자전기공학과, *포항공과대학교 정보통신연구소

Efficient Generation of Primes

Sang Gyoo Sim° Eun Jeong Lee* Pil Joong Lee
orabi@postech.ac.kr ejlee@oberon.postech.ac.kr pjl@postech.ac.kr
Dept. Elec. & Electrical Eng. Pohang Univ. of Sci & Tech.(POSTECH),
*POSTECH Information Research Laboratories(PIRL)

요 약

공개키 시스템에서는 소수들이 필요하고, 이 소수들을 효율적인 방법으로 빠르게 생성할 수 있는 것이 중요하다. 본 논문에서는 소수들을 간단하면서도 효과적으로 생성할 수 있는 방법을 제안한다.

1. 서론

공개키 시스템은 크게 두 가지로 나뉘어 진다. 두 소수의 곱으로 이루어진 큰 정수의 소인수 분해의 어려움에 근거한 RSA 시스템[2]과 이산대수의 어려움에 근거한 ElGamal[3] 등의 DLP(Discrete Log Problem)형 시스템이 있다. 이들 공개키 시스템들에서 공통적인 것은 모두 소수들의 사용이 중요하다는 것이다. 특히, RSA 시스템에서는 곱하여져서 법을 이루는 두 개의 소수가 사용자의 비공개키에 포함되기 때문에, 사용자들에 해당하는 소수들을 달리 만들어야 한다. 뿐만 아니라 공개키 시스템에서는 여러모로 소수 사용의 필요성이 많이 있어서, 빠르고 효율적인 방법으로 소수를 생성할 수 있는 방법이 요구 되고 있다.

소수 생성의 한가지 방법은 홀수인 난수를 생성해서 소수 판별 알고리즘이 소수라고 판정할 때까지 난수를 계속해서 생성하는 방법이다. n 비트 길이의 소수 a 를 생성하는 알고리즘은 다음과 같다.

알고리즘 1.
단계 1. n 비트 난수 a 를 생성(a 는 홀수)
단계 2. a 에 대해서 소수 판별
단계 3. a 가 소수이면, a 를 출력, 아니면 단계 1 로.

[1] 본 연구는 정보통신연구관리단의 국책기술 개발사업(차세대 IC 카드를 사용한 정보보호 신기술 시스템 개발)의 지원에 의해 이루어졌다.

이것은 유사 코드로 다음과 같이 표현된다.

```
while ( primality_test( a ) == NOT_PRIME )
        a = pseudorandom_number_generator( n bit );
        a = a ∨ 1;
return ( a );
```

최종적인 소수 판별을 위해서 소수 판별 알고리즘 primality_test()를 사용하고 있다. 임의의 홀수가 소수인지를 판별하는 소수 판별 알고리즘에는 결정적인 방법과 확률적인 방법으로 나누어진다[1]. 결정적인 방법은 계산 시간이 오래 걸리는 단점이 있어, 그에 비해 상대적으로 적은 계산량으로 소수성을 확률적으로 판정하는 확률적인 방법이 주로 이용되고 있다. 특히, Miller-Rabin test[5] 등이 많이 이용되고 있다. 일반적으로 합성수가 소수로 잘못 판단될 확률이 2^{-100} 이하가 되도록 해야 한다. Miller-Rabin을 이용할 경우, 한 번을 적용하면 합성수를 소수로 판단하는 확률은 1/4이 되므로, 주어진 확률의 안전도를 만족시키기 위해 primality_test()는 내부적으로 Miller-Rabin test를 50회 반복한다. 확률적인 소수 판별 알고리즘이 결정적인 방법들에 비해 계산량이 적은 것이기는 하지만, 이것도 계산의 부담이 크다. 소수 생성을 빠르게 하기 위해서는 계산량이 많은 소수 판별 알고리즘을 적용하는 빈도를 줄여야 한다.

가장 원시적인 결정적 소수 판별 알고리즘은 trial division이라 불리는 것으로 발생된 난수가 그 수의 제곱근 이하의 소수들에 대해서 나누어지지 않으면 그 수를 소수라고 판별할 수 있다. 이런 방법으로 소수를 판별하기 위해서는 미리 많은 양의 소수를 저장하고 있어야 하고, 많은 소수들에 대해서 나눗셈을 하는 것은 시간적으로 비효율적이므로 작은 수의 소수 판별에만 사용된다. 그러나, 큰 수의 소수 판별의 경우에도 작은 소수들로 나누어 보아 나뉘지 않는 것 만을 Miller-Rabin test를 통해 확인하면 알고리즘 1의 단점을 개선할 수 있다. 즉, 작은 소수들의 리스트를 만들고, 리스트에 포함된 소수들로 난수를 나누어 보아서 나누어지면 새로운 난수를 만들고, 어떤 소수들에 대해서도 나누어지지 않는 경우에만 소수 판별 알고리즘을 적용한다. 예를 들어, 2000보다 작은 소수들에 대해서 나머지를 검사해보는 알고리즘으로 다음과 같이 기술된다. 먼저, prime[i]는 2보다는 크고 2000보다 작은 소수를 크기 순으로 저장한 값이라고 하자.

알고리즘 2.
단계 0. prime[0]=3, ... , prime[301]=1999
단계 1. n비트 난수 a생성(a는 홀수), i = 0
단계 2. i = 0, 1, ... , 301 에 대해서 a가 prime[i]로 나누어지면 단계 1로
단계 3. a에 대하여 소수 판별
단계 4. a가 소수이면 a를 출력, 아니면 단계 1로.

이 알고리즘를 유사 코드로 표현하면 다음과 같다.

```
prime[302] = { 3, 5, 7, 11, ... ,.1997, 1999 };
while( flag== NOT_PRIME)
        a = pseudorandom_number_generator( n bits );
        a = a ∨ 1;
        flag = PRIME;
        for ( i = 0; i < 302; i++ )
                r = a mod prime[i];
                if ( r == 0 )
                        flag = NOT_PRIME;
                        break;
        if ( flag == PRIME )
                flag = Primality_Test( a );
return ( a );
```

알고리즘 2는 알고리즘 1보다는 개선이 되었지만, 소수 판별 후에 a가 합성수이면 매번 새로운 난수를 만들어서 모든 prime[i]에 대해서 나눗셈을 다시 계산해야 한다. 특히, a는 다정도 정수(multi-precision integer)이기 때문에 불필요하게 많은 나눗셈은 알고리즘을 비효율적이게 한다.

이런 점을 개선하기 위해, 본 논문에서는 소수 판별 알고리즘을 수행하는 회수를 최대한 줄여서, 소수를 생성하는 데 까지 걸리는 시간을 최소화할 수 있는 알고리즘들을 제안한다. 2절에서는 개선시킨 알고리즘들을 제안하고, 3절에서는 앞서 살펴본 1, 2의 알고리즘들과 제안한 알고리즘들을 비교한 결과를 보여 준다.

2. 소수 생성 알고리즘의 제안

● 제안 1

알고리즘 2에서는 소수 리스트에 속한 소수들 중의 하나로 나누어 떨어진다면, 매번 새로운 난수를 만들고, 이에 대해서 나눗셈 과정을 반복하게 된다. a mod prime[i]의 연산은 prime[i]가 다정도 정수가 아니어서 다정도 정수끼리의 나눗셈에서 필요한 몫 추정 과정이 없고, 그 연산은 다정도 정수와 하나의 정수를 곱하는 것과 비슷한 속도로 연산이 가능하다. 그러나, a가 소수 판별 알고리즘에 의해서 합성수로 판별되었을 때마다 새로운 난수 a를 생성해서 모든 i에 대해 a mod prim[i]를 수행하는 것은 그다지 효율적인 방법이 못된다. 알고리즘 2를 개선해보도록 하자.

먼저, 생성된 난수 a에 대해서 remainder[i] = a mod prime[i]를 계산해 놓는다. 만약 remainder[i] = 0 이면 prime[i]로 나누어지는 것이므로, a를 2만큼 증가시킨다. a가 2만큼 2증가하면 remainder[i]는 $Z_{prime[i]}$에서 2만큼 증가된다. 이런 방법으로 계속하면 모든 prime[i]에 대해서 나누어지지 않는 a를 얻을 수 있고, 그 때의 값에 대해 소수 판별 알고리즘을 적용한다. a가 합성수인 경우에는 a를 2만큼 더 증가시키고, remainder[i]도 $Z_{prime[i]}$에서 2만큼 증가시킨다. 이렇게 개선시켰을 경우 prime[i]를 갱신하는 계산

은 다정도 정수가 포함되지 않는 연산이므로 더욱 빠른 수행이 가능해질 것이다.

알고리즘 3. 제안 알고리즘(1)
단계 0. prime[0]=3, ... , prime[301]=1999.
단계 1. n 비트 난수 a 생성(a는 홀수), i = 0
단계 2. i = 0, 1, ... , 301 에 대해서 remainder[i] = a mod prime[i]
단계 2.1. remainder[i] = 0 이면,
a = a +2, 모든 j(j ≤ i)에 대해서 remainder[j] = (remainder[j] + 2) mod prime[j]
단계 3. i = 0, 1, ... , 301 에 대해서 remainder[i] = 0 이면 단계 3.1
단계 3.1. a = a+2, 모든 i = 0, 1, ... , 301 에 대해서 remainder[i] = (remainder[i] +2) mod prime[i]
단계 4. a에 대하여 소수 판별
단계 5. a가 소수이면 a를 출력하고 알고리즘을 끝낸다.
단계 6. a = a + 2, i = 0, 1, ... , 301 에 대해 remainder[i] = (remainder[i] + 2) mod prime[i], 단계 3 으로

유사 코드는 다음과 같다.

```
prime[302] = { 3, 5, 7, 11, ... , 1997, 1999 };
a = pseudorandom_number_generator( n bits );
a = a ∨ 1;
for ( i = 0; i < 302; i++ )
        remainder[i] = a mod prime[i];
        if ( remainder[i] == 0 )
                for (j = 0; j < i; j++ )
                        remainder[j] = ( remainder[j] + 2 ) mod prime[j];
                remainder[i]=2;
                a = a + 2;
while( flag == NOT_PRIME )
        for ( i = 0; i < 302; i++ )
                if ( remainder[i] == 0 )
                        for ( j = 0; j < 302; j++ )
                                remainder[j] = ( remainder[j] + 2 ) mod prime[j];
                        i = 0;
                        a = a + 2;
        flag = Primality_Test( a );
        if ( flag == NOT_PRIME )
                a = a + 2;
                for ( j = 0; j < 302; j++ )
                        remainder[j] = ( remainder[j] + 2 ) mod prime[j];
        if (( flag == PRIME )&&( |a| > n ))
                flag = NOT_PRIME;
return ( a );
```

알고리즘 3에서는 다정도 정수 a를 직접 나누는 계산은 prime[i]들에 대해서 각각 한 번씩 이루어지면 되고, 그 후에는 나머지인 remainder[i]만 갱신면 된다. 첫번째 for 구문의 remainder[j] = (remainder[j] + 2) mod prime[j]의 연산에서, remainder[j]는 1999 보다는 작은 정수이므로 16 비트 이

상을 워드 크기로 하는 컴퓨터들에서는 하나의 워드로 저장된다. remainder[j]+2 역시 하나의 워드에 저장되어서 이 연산은 하나의 워드를 하나의 워드로 나누는 연산만으로 가능하다. 또, 가장 최악의 경우에 주어진 i에 대해서 remainder[j] ≠ 0 (j < i)이고, remainder[i] = 0인 경우이므로, 첫 번째 for 구문에서 a = a + 2가 실행되는 경우는 최대 302 번까지 실행된다.

두 번째의 for 구문에서는 remainder[i] = 0인 i를 없애는 것인데, 결론적으로는 a가 어떤 prime[i]로도 나누어지지 않게 하는 것이다. 홀수들 중에서 3, 5, 7의 배수가 되는 정수들을 가려낼 경우에 54% 정도가 걸러지고, 100보다 작은 소수들에 대해서는 76%, 256보다 작은 소수들에 대해서는 80%, 2000의 경우에는 85%를 걸러 낼 수 있다. 임의의 n에 대해서는 n보다 작은 소수들을 사용해서 배수가 되는 홀수를 걸러내면 남는 비율은 1.12/ln(n)로 알려져 있다[4]. 알고리즘 3은 2000보다 작은 소수들에 대해서 나눗셈을 적용하고, 나누어지지 않는 홀수들에 대해서 소수 판별 알고리즘을 사용하도록 했다. 이렇게 하면, 작은 소수들의 나눗셈을 적용하지 않는 경우에 비해서 ln(2000)/1.12 ≈ 6.787배 정도로 소수 판별 알고리즘의 적용 회수를 줄일 수 있다. 실제 구현에서 prime[i]의 가장 큰 값을 얼마로 정할 것인가는 컴퓨터의 환경에 따라서 정하면 된다.

● 제안 2

알고리즘 3에서 소수 판별 알고리즘을 적용했는데 소수가 아니었을 경우에는 a = a + 2를 반복하여 적용하게 되고, 이때마다 remainder[i]가 갱신된다. 특히 prime[0](= 3)에 대해서 remainder[0]은 3번에 한 번 꼴로 0이 된다. 이렇게 되면, 모든 i에 대해서 remainder[i]를 갱신해야 하는데, 그 경우의 빈도가 너무 자주 일어난다. 이런 점을 해결하는 알고리즘을 만들어 보자.

홀수인 난수 a를 생성해서 3으로 나눈다. 3으로 나누어 떨어질 경우 a는 계속 홀수여야 하므로 a를 2만큼 증가시킨다. 다음에는 a를 5로 나누어 5의 배수이면 2와 3의 최소공배수인 6을 더한다. 이렇게 하면, 2로도 나누어지지 않고(홀수), 3으로도 나누어 지지 않는 성질이 유지된다. 다음에 7로 나누어지면, 이미 2, 3, 5로 나누어 봤으므로, 그들의 최소공배수인 30을 a에 더한다. 그러면, 2, 3, 5로 나누어지지 않는 성질이 유지되면서, 7로 나누어지지 않게 된다. 이런 방법으로 해나가면, 특정 몇 개의 소수들에 대해서 배수가 되지 않는 홀수 a를 얻을 수 있고, 소수 판별을 통해서 a가 합성수라고 판별되었을 경우에는 나눗셈을 시도했던 모든 소수들의 최소공배수를 계속 더하여 나가면, a가 그 소수들로 나누어지지 않는 성질을 계속 유지시킬 수 있어서 더이상의 나눗셈이 필요 없어진다. 이때에 나누어 보는 소수를 너무 많이 잡으면 a에 더하여지는 수가 매우 커지므로 이 점을 고려하여 그 개수를 정하여야 한다.

알고리즘 4. 제안 알고리즘(2)
단계 1. n비트 난수 a를 생성(a는 홀수)
단계 2. a가 3의 배수이면, a = a + 2
단계 3. a가 5의 배수이면, a = a + 6

단계 4. a가 7의 배수이면, a = a + 30
단계 5. a가 11의 배수이면, a = a + 210
단계 6. a가 13의 배수이면, a = a + 2310
단계 7. a가 17의 배수이면, a = a + 30030
단계 8. a가 19의 배수이면, a = a + 510510
단계 9. a에 대하여 소수 판별
단계 10. a가 소수이면, a를 출력하고 알고리즘을 마친다.
단계 11. a = a + 9699690, 단계 9로

```
prime[7] = { 3, 5, 7, 11, 13, 17, 19 };
while ( flag == NOT_PRIME )
        a = pseudorandom_number_generator( n bit );
        a = a ∨ 1;
        for (i = 0, temp = 2; i < 7; i++ )
                if ( a mod prime[i] == 0 )
                        a = a + temp;
                temp = temp × prime[i];
        while ( flag == NOT_PRIME )
                flag = primality_test( a );
                if ( flag == NOT_PRIME )
                        a = a + temp;
        if (( flag == PRIME )&&( |a| > n ))
                flag = NOT_PRIME;
return(a);
```

temp = 2는 초기값으로 2를 저장하고 있고, 나눗셈을 시도하는 소수들을 하나씩 곱하여서 그 이전 소수들의 최소공배수 값을 갱신하면서 저장하게 되고, 최종적으로는 2 × 3 × ... × 19 = 9699690 = $94016a_{(h)}$을 저장하게 된다. 계속 더하는 과정에서 a의 길이가 n을 넘으면, 새로운 a를 난수값으로 생성한다.

알고리즘 3과 비교하여 보자. 알고리즘 3에서 빈번하게 일어났던 remainder[i]들의 갱신은 나눗셈 몇 번 이후 일어나지 않지만, 작은 소수를 20보다 작은 수로 정하였기 때문에 나눗셈을 통과하고 남아서 소수 판별에 적용되는 홀수는 1.12/ln(20) ≈ 37.4% 정도가 된다. 결과적으로 1.12/ln(2000) ≈ 14.7%인 알고리즘 2에 비해서 소수 판별 알고리즘을 2.54배 정도 더 많이 수행하여야 한다.

16비트 워드를 사용하는 컴퓨터에서 구현하는 것을 생각해 보자. temp의 최종값이 되는 $94016a_{(h)}$은 16비트를 넘어서 다정도 정수 a에 더하는 연산을 처리하려면 추가적인 구문들이 필요해진다. 더하여지는 수를 16비트가 되게 나누어 보는 소수들을 17까지로 줄이고, 2, 3, ... , 17의 최소공배수를 매번 더하는 것이 아니라 2 × 3 × ... × 13 = 30030 = $754E_{(h)}$를 더하고, 17에 대해서는 a mod 17 값을 변수 r에 저장해서 r = 0인지를 검사하도록 하자.

알고리즘 5. 알고리즘 4의 변형

```
while ( flag == NOT_PRIME )
```

```
a = random_number_generator( n bits );
a = a ∨ 1;
if ( a mod 3 == 0 )          a = a + 2;
if ( a mod 5 ==0 )           a = a + 6;
if ( a mod 7 ==0 )           a = a + 30;
if ( a mod 11 == 0 )         a = a + 210;
if ( a mod 13 == 0 )         a = a + 2310;
if ( r = a mod 17 == 0 )     a = a + 30030;          r = 8;
while(( flag = primality_test( a )) == NOT_PRIME )
        a = a + 30030;
        r = ( r + 8 ) mod 17;
        if ( r == 0 )
                a = a + 30030;
                r = 8;
if (( flag == PRIME )&&( |a| > n ))
        flag = NOT_PRIME;
return ( a );
```

r = a mod 17 = 0인 경우에는 a는 30030만큼 증가하고, r = 30030 mod 17 = 8로 갱신된다. a에 더하여지는 모든 값은 16비트 값이 되었고, 알고리즘 4에 비해서는 나눗셈을 검사하는 소수의 개수가 하나 줄었다. 그러나, 16비트 구현에는 더욱 유리해졌고, a가 증가하는 폭이 줄어서 증가 폭이 큰 경우보다는 a를 선택할 수 있는 경우가 늘어났다.

● 제안 3

알고리즘 3의 단점은 작은 i에 대해서 remainder[i] = 0이 되는 경우가 자주 일어나서 모든 i에 대해서 remainder[i] 값들을 갱신해야 하는 경우가 많다는 것이었다. 그리고 알고리즘 4의 단점은 나눗셈을 시도해보는 소수가 너무 적어서 상대적으로 소수 판별 알고리즘을 수행하는 빈도가 늘어난다는 것이었다. 그러나, 알고리즘 3은 나눗셈을 검사하는 소수들이 많이 있어서 소수 판별 알고리즘을 적용하는 회수가 적은 장점이 있고, 알고리즘 4는 작은 소수들에 대해서는 더이상 나눗셈을 확인하지 않아도 된다는 장점이 있었다. 그러면, 알고리즘 3과 4의 장점들을 가지면서 단점들을 보완하는 방법을 찾아보자. 먼저, 알고리즘 4를 대신해서 16비트 구현에 더욱 유리한 알고리즘 5를 생각해보자. 알고리즘 5에서 변수 r에 a mod 17의 값을 저장하였었다. 그러나, 이 부분에서 알고리즘 3에서와 마찬가지로 prime[i]들에 대해서 remainder[i] = a mod prime[i]를 저장하여 이용한다면 나눗셈을 확인하는 소수가 늘어나서 소수 판별 알고리즘을 적용하는 회수가 줄어들 것이다. 또, 알고리즘 5의 장점대로 작은 소수들에 대해서는 더 이상 나눗셈을 확인하지 않아도 되므로, 알고리즘 3에서처럼 작은 prime[i]들에 대해서 remainder[i] = 0인 경우가 빈번하게 일어나지도 않을 것이다.

알고리즘 6. 제안 알고리즘(3)
단계 0. prime[0] = 17, prime[1] = 19, prime[2] = 23, ... , prime[296] = 1999

단계 1. n 비트 난수 a를 생성(a는 홀수)
단계 2. a가 3의 배수이면, a = a + 2
단계 3. a가 5의 배수이면, a = a + 6
단계 4. a가 7의 배수이면, a = a + 30
단계 5. a가 11의 배수이면, a = a + 210
단계 6. a가 13의 배수이면, a = a + 2310
단계 7. i = 0, ... ,296 에 대해서 remainder[i] = a mod prime[i]
단계 7.1. remainder[i] = 0 이면, a = a + 30030, 모든 j(j ≤ i)에 대해서 remainder[j] = (remainder[j] + 2) mod prime[j]
단계 8. i = 0, ... , 296 에 대해서 remainder[i] = 0 이면 단계 8.1
단계 8.1. a = a + 30030, 모든 i = 0, ... , 296 에 대해서 remainder[i] = (remainder[i] + 30030) mod prime[i]
단계 9. a에 대하여 소수 판별
단계 10. a가 소수이면 a를 출력하고 알고리즘을 끝낸다.
단계 11. a = a +30030, i = 0, ... , 296 에 대해 remainder[i] = (remainder[i] + 30030) mod prime[i], 단계 8로

```
prime[297] = { 17, 19, 23, ... , 1999 };
while ( flag == NOT_PRIME )
        a = random_number_generator( n bits );
        a = a ∨ 1;
        if ( a mod 3 == 0 )                 a = a + 2;
        if ( a mod 5 ==0 )                  a = a + 6;
        if ( a mod 7 ==0 )                  a = a + 30;
        if ( a mod 11 == 0 )                a = a + 210;
        if ( a mod 13 == 0 )                a = a + 2310;
        for ( i = 0; i < 297; i++ )
                remainder[i] = a mod prime[i];
                if ( remainder[i] == 0 )
                        for ( j = 0; j < i; j++ )
                                remainder[j] = ( remainder[j] + 30030 ) mod prime[j];
                        remainder[i] = 30030 mod prime[i];
                        a = a + 30030;
        while ( flag == NOT_PRIME )
                for ( i = 0; i < 297; i++ )
                        remainder[i] = ( remainder[i] + 30030 ) mod prime[j];
                        if ( remainder[i] == 0 )
                                for ( j = 0; j < 297; j++ )
                                        remainder[j] = ( remainder[j] + 30030 ) mod prime[j];
                        i=0;
                flag = primality_test( a );
                if ( flag == NOT_PRIME )
                        a = a + 30030;
                        for ( i = 0; i < 297; i++ )
                                remainder[i] = ( remainder[i] + 30030 ) mod prime[j];
return( a );
```

2, 3, ... , 13 의 소수에 대한 나눗셈은 알고리즘 5 에서처럼 한 번 씩 수행되면서 그들 소수의 배수가 되지 않도록 특정한 값들을 a에 더해 준다. 일단 그들 소수의 배수가 아니면, 알고리즘 3 에서 처럼 a

에 소수 2, 3, 5, ... , 13 의 최소공배수인 30030 을 더해나가며 prime[i]들의 배수인지를 검사한다.

결과적으로 여섯 개의 작은 소수들에 대한 나눗셈 검사는 알고리즘 4 를 채용하고, 비교적 큰 소수들에 대한 나눗셈 검사는 알고리즘 3 을 채용하게 되었다. 이렇게 해서, 알고리즘 6 은 앞서 지적한 알고리즘 3 과 4 의 장점들을 가지며 단점들을 동시에 보완했다. 특히, remainder[j] = (remainder[j] + 30030) mod prime[j]의 연산에서 remainder[j]가 1999 보다 작은 수이고, 30030 을 더한 후 나누어도 이 연산은 16 비트 워드 단위 안에서 가능하다. 이렇게 해서 16 비트 구현에서 유리하도록 구성된 알고리즘 5 의 장점도 유지하여, 16 비트 구현에서는 더욱 효과적인 수행이 가능해 질 것이다.

3. 결과와 분석

16 비트를 기본 연산 단위로 하여 C 언어로 구현하였고, Visual-C++ Ver. 5.0 으로 실행을 시켰다. 비교한 알고리즘은 1, 2, 3, 5, 6 이고, Pentium-200MHz PC(P-200)와 Sparc-20 60MHz(S-20)에서 128 비트와 160 비트 소수를 1000 개 생성하는데 필요한 평균 시간을 구하였다. 그 결과는 표 1 에 정리하였는데, 표 1 을 살펴보면, 알고리즘 6 이 가장 빠르고, 제안한 알고리즘 5 는 특별히 빠르지는 않은 것으로 나타났다.

	실행 환경	알고리즘 1	알고리즘 2	알고리즘 3	알고리즘 5	알고리즘 6
128 bits	P-200	0.502	0.310	0.222	0.479	0.134
	S-20	1.507	0.636	0.448	1.271	0.382
160 bits	P-200	1.057	0.639	0.464	0.964	0.258
	S-20	3.163	1.326	0.912	2.806	0.747

표 1. 알고리즘들의 평균 수행 시간 비교(sec)

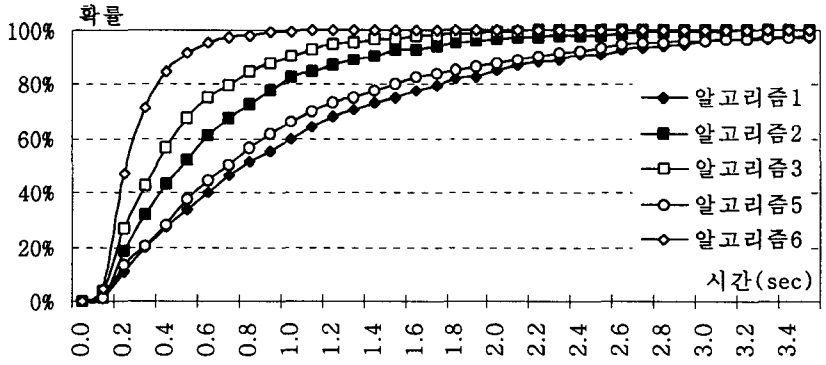

그림 1. 알고리즘의 수행 시간과 확률에 대한 비교

그림 1 은 Pentium-200MHz 에서 160 비트를 생성할 때, 하나의 소수가 생성되어 나오기 까지 걸리는 시간의 확률을 누적 그래프로 그린 것이다. 알고리즘 5 는 알고리즘 1 보다는 개선되었지만, 크게 개선되지는 않았음을 보여 주고 있다. 그러나 알고리즘 3 과 알고리즘 5 를 결합한 알고리즘 6 의 곡선은 나

머지 곡선들보다 빠르게 증가하는 것을 볼 수 있다. 이것은 알고리즘 6을 사용했을 때, 다른 알고리즘들 보다 소수를 빠르게 생성할 수 있다는 것을 보여주는데, 그 결과는 위의 표 1과 일치하는 내용이다.

4. 결론

본 논문에서는 난수를 생성해서 소수 판별 알고리즘에 적용시키는 과정을 반복함으로써 소수를 얻는 방법을 개선시키는 알고리즘들을 제안했다. 제안한 알고리즘들은 소수 판별 알고리즘의 수행 시간이 오래 걸린다는 것을 기본적인 가정으로 하고 있으며, 소수 판별 알고리즘을 적용시키는 횟수를 감소시키는데 그 주안점이 있었다. 그 문제점을 해결하는데 세 가지 해결 방법을 제안했다. 첫째, 일정한 작은 소수들로 나누어 보는 과정을 반복하여 그 소수들로 나누어지지 않는 경우에만 소수 판별 알고리즘을 적용시키는 방법이었다. 둘째, 작은 소수들의 배수가 되지 않게 설정하여 반복적으로 작은 소수들의 최소공배수 만큼씩 더해가는 방법을 제안했다. 그리고, 그 두 가지 해결방법의 단점들을 보완하는 세 번째 방법을 제안했고, 이 방법이 앞선 두 가지 방법들 보다 더 효율적인 알고리즘이라는 것도 살펴 보았다.

RSA 알고리즘에서 안전한 소수들의 곱을 법으로 사용하기 위해 여러 가지 조건들[6]에 대해서 연구되고, 그 조건들을 만족시키는 소수들을 생성시키는 방법들에 대해서도 연구되고 있다. 그러나, 아주 강력한 안전성을 요구하지 않는 한, 소수 판별 알고리즘만을 거친 난수를 소수로 사용하여도 충분한 안전성을 보장한다는 주장도 있다.

본 논문에서 제안된 알고리즘을 활용한다면, 작은 소수 뿐만 아니라, RSA 시스템의 공개 법을 이루는 비공개 소수를 구하는 데에도 사용할 수 있을 것이다.

5. 참고 문헌

[1] 최 영주, "솟수 판별법(primality test)", *통신정보보호학회지*, 제 2 권 제 1 호, 1992. 3. pp. 49-51.

[2] R. Rivest, A. Shamir and I. Adleman, "A method for obtaining digital signatures and public key cryptosystems", *Comm. ACM*, 21(2), 1978, pp.120-126.

[3] T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logatithms", *IEEE Trans. Info. Theory*, IT-31, 1985, pp.469-472.

[4] Bruce Schneier, *Applied cryptography*, 2nd Edition, John Wiley & Sons, 1996, pp. 258-261.

[5] D. Knuth, *The Art of Computer Programming: Vol. 2, Seminumerical Algorithms*, 2nd Ed, Addision-Wesley, 1981.

[6] Robert D. Silverman, "Fast Generation of Random, Strong RSA Primes", RSA Laboratories' *CryptoBytes*, Vol. 3, Num. 1, Spring 1997, pp. 9-13.