고속 복수 소수 생성을 위한 새로운 방법

*정정수, *허석중, *노종선,
*서울대학교
전기컴퓨터공학부&INMC,
*{integer, hsjbest}@ccl.snu.ac.kr,
jsno@snu.ac.kr,

**임대운
**동국대학교 IT학부
정보통신공학전공,
**daewoonlim@gmail.com

***김태성
***SK Telecom
***helios@sktelecom.com

New Method for Fast Multiple Prime Number Generation

*Jung-Soo Chung, *Seok-Joong Heo,
*Jong-Seon No
*Seoul National Univ.

**Dae-Woon Lim
** Dongguk Univ.

***Tae-Sung Kim
***SK Telecom

요 약

본 논문은 공개키 암호화 시스템에서 필요한 소수를 생성하는 방법에 관한 것이다. 서로 공통 원소가 없는 두 개의 작은 소수 집합 (small prime set, SPS)을 이용해서 두 개의 비트 배열을 생성하고, 이를 이용해서 복수의 소수를 생성한다. 일반적인 비트 배열 알고리즘과 다르게 비트 배열의 모든 원소에 대해서 소수 여부를 판별하고, 두 개의 SPS 가 서로의 원소를 포함하지 않은 특징을 이용하며 하나의 비트 배열을 재활용해서 계속 소수를 생성할 수 있다는 점에 그 특징이 있다. 두 개의 비트 배열을 사용함으로써 고속 복수 소수 생성 장치는 기존의 비트 배열 알고리즘의 비효율성을 개선하고 복수의 소수 생성에 특화될 수 있으며, 한번 계산한 비트 배열을 지속적으로 재활용함으로써 평균 소수 생성 시간을 줄이는 효과가 있다.

I. 서론

최근 전자상거래의 발달과 유무선 데이터 통신의 증가에 따라 기밀성에 대한 필요가 증가하고 있다. 이를 위해 여러 가지 암호화 방식들이 제안되었으며, 이들 중 공개키 암호화 방식이 간편한 암호관리와 안전한 통신을 가능하게 한다는 점에서 널리 쓰인다. 공개키 암호화 방식 중 RSA (Rivest, Shamir, and Adleman) 공개키 암호화 방식이 가장 선호되며 널리 쓰이고 있다.

RSA 공개키 암호화 방식은 큰 소수의 인수분해가 어렵다는 사실에 근거한다. 즉 두 개의 소수 p, q 에 대해서 두 수의 곱인 $n=pq$ 를 인수분해 해서 p 와 q 를 찾아 내기 매우 어려운 특징이 있다. 즉 RSA 공개키 암호화 방식을 무력화하기 위해서는 효율적인 인수분해 알고리즘이 필요하게 되고, 이 같은 공격을 방지하기 위해 매우 큰 수를 이용함으로써 인수분해를 어렵게 한다. 최근에는 RSA 공개키 암호화 방식의 기밀성을 높이고자 암호화에 사용되는 소수들의 크기를 512 비트 이상의 큰 수를 사용한다.

본 논문은 공개키 암호화 시스템에서 필요한 소수를 생성하는 방법에 관한 것이다. 서로 공통 원소가 없는 두 개의 작은 소수 집합 (small prime set, SPS)을 이용해서 두 개의 비트 배열을 생성하고, 이를 이용해서 복수의 소수를 생성한다. 일반적인 비트 배열 알고리즘과 다르게 비트 배열의 모든 원소에 대해서 소수 여부를 판별하고, 두 개의 SPS 가 서로의 원소를 포함하지 않은 특징을 이용하며 하나의 비트 배열을 재활용해서 계속 소수를 생성할 수 있다는 점에 그 특징이 있다. 두 개의 비트 배열을 사용함으로써 고속 복수 소수 생성 장치는 기존의 비트 배열 알고리즘의 비효율성을 개선하고 복수의 소수 생성에 특화될 수 있으며, 한번 계산한 비트 배열을 지속적으로 재활용함으로써 평균 소수 생성 시간을 줄이는 효과가 있다.

II. 배경지식

소수는 자기 자신과 1 이외에 어떤 약수도 갖지 않는 수를 의미한다. 예를 들면, 2, 3, 5, 7, 11, 13, 17,… 등은 모두 소수이다. 자연수 n 이 소수인지 아닌지를 판정하려면, $2\le p\le \sqrt{n}$ 인 범위에 있는 모든 소수 p 로 n 을 나누어 보아, 나누어지지 않으면 소수이고, 나누어지면 합성수이다. 즉, 소수는 약수로 1 과 자신을 가진 자연수이며 합성수는 약수가 3 개 이상인 자연수이다.

일반적으로 소수를 생성하는 방법은 임의의 홀수를 생성한 후, 이를 소수 검사방법 (primality test)를 이용해 검사하고 실패할 경우 또 다시 임의의 홀수를 생성하거나 이전 것에 2 를 더한 수에 대해 다시 검사하는 방법을 반복하는 방법이 있다. 소수 검사방법으로는 결정적 검사와 확률적 검사가 있다.

결정적 소수 검사방법은 확률 1 로 임의의 수가 소수임을 확인하는 방법으로 나눗셈 테스트, Pocklington 테스트, Jacobi sum 테스트 등이 있다. 나눗셈 테스트는 임의의 수보다 작은 소수들로 나누어 보는 방법으로 실제로 사용하기에는 너무 느리다. 마찬가지로 Pocklington 테스트, Jacobi sum 테스트 등도 매우 느린 단점이 있다.

확률적 검사방법은 임의의 수가 1 에 가까운 확률로 소수임을 확인하는 방법으로 Solovay-Strassen 테스트, Fermat 테스트 그리고 Miller-Rabin 테스트 등이 있다.

널리 사용되는 방법으로는 나눗셈 테스트와 함께 Fermat 테스트나 Miller-Rabin 테스트를 혼합하여 구성하는 것이다.

대표적으로 사용되는 알고리즘의 경우 나눗셈 테스트와 Miller-Rabin 테스트를 혼합하여 사용한다. 이 경우 비트 배열 알고리즘 (bit array algorithm)을 사용해서 Miller-Rabin 테스트의 횟수를 줄이는 방법이 가장 효율적인 것으로 알려져 있다.

비트 배열 알고리즘은 임의로 생성된 홀수 q 에 대해서 미리 정의된 작은 소수 집합에 대해서 모듈러 나눗셈 연산을 수행하고 그 결과를 1 과 0 으로 표시하는 비트 배열을 생성한다. 일반적인 비트 배열 알고리즘은 생성된 비트 배열 중 0 의 위치에 해당하는 홀수만을 소수 검사방법에 적용하고, 소수로 판정되면 정지하고 생성된 비트 배열을 폐기한다. 새로운 소수를 생성하기 위해서는 임의의 홀수를 다시 생성하고 그 홀수에 대해서 다시 비트 배열을 생성해야 한다.

비트 배열을 사용하는 기존의 알고리즘은 생성된 비트 배열을 단일 소수 생성에 사용 후 폐기하지만, 본 논문에서 제안하는 알고리즘은 생성된 비트 배열을 반복적으로 사용해서 비트 배열의 생성에 소요되는 연산량을 줄이고 궁극적으로 개별 소수 생성 시간을 단축시키는 장점이 있다.

III. 소수 생성 알고리즘

일반적인 소수판별 알고리즘에서 시간이 가장 오래 걸리는 부분은 primality test oracle 이다. 따라서, 본 논문에서는 최적화된 primality test oracle 의 호출 수를 줄이는 것을 가장 중요한 목표로 한다. Primality test oracle 호출을 줄이게 되면 큰 소수를 생성하는데 필요한 시간이 감소되게 된다. 호출을 줄이기 위한 방법으로는 작은 소수 집합으로 미리 나누거나 혹은 작은 소수 집합을 이용한 방법들이 제안되었다. 기존에 이미 알려진 소수 판정 알고리즘과 제안한 알고리즘을 설명하면 다음과 같다.

다음 알고리즘 1 과 2 는 가장 기본적인 소수 생성 알고리즘으로 임의의 소수를 생성 후 소수 판정 테스트를 거쳐서 소수인지 아닌지 판정한다.

> 알고리즘 1:
> 임의의 소수 p 를 생성하고 primality test oracle를 호출한다. 소수가 생성되지 않으면 다시 임의의 소수를 생성하고 이를 반복한다.

> 알고리즘 2:
> 임의의 소수 p 를 생성하고 이것을 primality test oracle를 호출한다. 소수가 생성되지 않으면 p+2에 대해서 primality test oracle를 호출하고 실패시 이를 반복한다.

임의의 소수 p 를 생성할 때 걸리는 시간은 0.151ms 이고 p+2 의 연산에 걸리는 시간은 0.115ms 이다. 실패 시 임의의 수를 생성하거나 기존의 임의의 수에 +2 를 하는 것은 소수 생성에 안정성 측면에서 큰 영향을 주지 않는다고 알려져 있다. 시뮬레이션을 통해서 512 bit 의 소수를 생성할 때 primality test oracle 부분을 통과하는데 걸리는 시간은 성공 시 290.5ms, 실패 시 48.5ms 가 걸린다. 알고리즘 1 과 2 에서 512bit 소수를 생성을 위해서는 177.8 회를 반복하였을 때 1 개의 소수가 발생되었다. 이 값은 어떤 수 x보다 작은 소수의 개수가 계산하고자 할 때 $\ln x/2$ 개의 후보에 대해서 소수 판정을 해야 하는데 $512 \times \ln 2/2 =$ 177.4 회와 비슷한 값을 가진다. 알고리즘 2 에서 소수 생성에 걸리는 시간은 시뮬레이션으로는 8864.45ms 이다. 실패가 177 회이고 성공이 1 회라고 하면 8895.5ms 이고 실패가 176 회 성공이 1 회이면 8846.89ms 이다.

알고리즘 3 은 임의로 생성된 수를 바로 primality test oracle 호출하지 않고 prime sieve test 를 거쳐서 최대한 primality test oracle 호출을 줄이는 알고리즘이다. primality test oracle 호출이 소수 판정 알고리즘에서 가장 시간이 많이 걸리는 부분으로 소수가 아닐 가능성이 있는 것은 primality test oracle 호출을 하지 않는 것이 효율적이다.

> 알고리즘 3:
> 임의의 소수 p를 생성한다. 작은 소수의 집합을 A라 하고 이 소수들로 A를 나누어 집합내의 모든 원소로 나누어지지 않는 p에 대해서만 primality test oracle를 호출한다. 실패 시 반복적으로 수행한다.

작은 소수 집합 크기가 작으면 임의로 생성된 p를 나누는데 걸리는 시간은 적게 걸리지만 primality test oracle 를 호출하는 횟수가 집합 크기가 큰 것에 비해 많기 때문에 항상 최적이지는 않다. 1 회의 성공과 26 회의 실패와 222 개의 작은 소수 집합으로 임의로 생성된 수를 나누는데 걸리는 시간을 더하면 1650.33ms 이다. 즉 알고리즘 2 에서 178 회 정도 밀러라빈 테스트를 수행하는데 알고리즘 3 은 밀러라빈 테스트를 29 회 정도 수행하게 된다. 소수 생성에 많은 부분을 차지하는 밀러라빈 테스트의 호출을 약 15.2%로 줄여 소수 생성 시간을 단축하였다. 이 15.2%은 $1-\left(1-\frac{1}{3}\right)\left(1-\frac{1}{5}\right)\left(1-\frac{1}{7}\right)\cdots(1-\frac{1}{p_{222}})$의 15.4 와 거의 유사하다.

알고리즘 4 는 알고리즘 3 의 prime sieve test 를 더 효율적으로 개선한 것으로 작은 소수들로 나누는 과정인 즉 작은 소수의 배수를 찾는 과정에 걸리는 시간을 줄인다.

> 알고리즘 4:
> 임의의 소수 p를 생성한다. 작은 소수의 집합을 A와 크기가 s인 bit array B를 설정한다. Bit array B는 작은 소수 집합 A의 연산으로 각 값을 채운다. Bit array 값이 0인 것에 대해서만 primality test oracle를 호출한다.

이 후의 알고리즘은 소수를 생성하는데 있어서 하나의 소수만을 생성하고 소수 생성 프로그램을 종료하는 것이 아니라 기존에 생성되었던 비트 배열의 재사용을 통해서 소수 생성 시간을 줄이고자 제안한 알고리즘이다.

> 알고리즘 5:
> 임의의 소수 p를 생성한다. 작은 소수의 집합을 A와 크기가 B인 bit array가 설정한다. Bit array B는 작은 소수 집합 A의 연산으로 각 값을 채운다. Bit array 값이 0인 것에 대해서만 primality test oracle를 호출한다. 알고리즘 4 에서는 성공 시에도 bit array의 마지막까지 0 인 bit array에 대해서 primality test oracle 를 호출한다. 생성된 소수는 인터리빙을 이용하여 임의성을 준다.

알고리즘 4 와의 차이는 만약 확률적으로 소수일 가능성이 높은 수가 생성되었다면 소수 판정을 종료하는 것이 아니라 기존의 비트 배열을 재사용하여서 이미 생성된 소수 근처에서 소수를 다시 찾아가는 과정이다. 이는 비트 비열을 생성하는 시간을 줄일 수 있지만 공격자가 이 알고리즘을 파악하고 하나의 소수를 찾았을 때 다른 소수도 찾는 것이 용이한 단점이 있다. 소수 집합 크기가 클수록 소수 생성 시간이 줄어든다.

알고리즘 6 은 알고리즘 5 를 변형한 것으로 한 번 생성된 bit array 에서 소수를 원하는 개수만큼만 찾는 알고리즘이다.

> 알고리즘 6:
> 임의의 소수 p를 생성한다. 작은 소수의 집합을 A와 크기가 B인 bit array가 설정한다. Bit array B는 작은 소수 집합 A의 연산으로 각 값을 채운다. Bit array 값이 0인 것에 대해서만 primality test oracle를 호출한다. 한 bit array내에서 k개의 소수를 생성할 때까지 0인 bit array에 대해서 primality test oracle를 호출한다.

본 논문에서 제안하는 알고리즘으로 알고리즘 7 은 비트 배열을 재사용함으로써 비트 배열을 생성하는 시간을 줄이는 알고리즘이다. 비트배열 생성할 때의 나눗셈을 최소한으로 줄여서 곱셈으로 바꾸어서 비트 배열을 재사용하는 것이 핵심이다.

> 알고리즘 7:
> 임의의 소수 p를 생성한다. 작은 소수의 집합을 A와 A를 포함하는 소수 집합을 A', 크기가 B인 bit array가 설정한다. Bit array B는 작은 소수 집합 A의 연산으로 각 값을 채운다. 이렇게 A에 대해서 만들어진 bit array는 재사용이 가능할 것이다. 그리고 A' 에 대해서도 연속적으로 bit array를 만든다. 임의의 수와 A의 각 소수들의 곱을 처음 생성한 p에 더하면 특정한 bit array부분은 재사용이 가능하고 이에 대해서 primality test oracle를 호출한다.

이를 알고리즘으로 표현하면 다음과 같다.

> 입력: $\mathrm{A} = \mathrm{SPS}(r)$, $\mathrm{A}' = \mathrm{SPS}(r')$, $r < r'$ 여기서 생성하고자 하는 소수 개수 M
> 출력: l bit 잠재적인 소수
>
> 1. $cnt = 0$
> 2. $0 \le i \le s-1$ 에 대하여 $a_i := 0$ 으로 놓고 $cnt_1 = 0$로 놓는다.
> 3 임의의 l bit 홀수 q 를 발생시키고 $q^{(0)} := q,\ i = 0$로 놓는다.
> 4. $p_j \in \mathrm{SPS}(r)$에 대하여
>   a. $w_j^{(i)} := q^{(i)} \bmod p_j$;
>   b. $g(p_j)$를 구한다.
> $g(p_j) = 0,\ w_j^{(i)} = 0$
> $g(p_j) = (p_j - w_j^{(i)})/2,\ \ w_j^{(i)}$ is odd
> $g(p_j) = (2p_j - w_j^{(i)})/2,\ \ w_j^{(i)}$ is even
>   c. $0 \le m \le \lfloor (\mathrm{s} - \mathrm{g}(\mathrm{p_j}))/\mathrm{p_j} \rfloor$ 에 대하여 $a_{g(p_j)+mp_j} := 1$로 놓는다.
> 5. $p_j \in \mathrm{SPS}(r') \backslash \mathrm{SPS}(r)$에 대해서 $g(p_j)$를 구한다.
> 6. for $i := 0$ to $s-1$에 대하여
>   a. 만일 $(a_i = 0)$ AND $(MRTest(q^{(i)}) = \mathrm{TRUE})$이면 $q^{(i)}$를 저장한다.
>   b. $cnt_1 = cnt_1 + 1,\ cnt = cnt + 1$
>   c. $cnt_1 < k$ 이면 $q^{(i+1)} := q^{(i)} + 2$
>   d. $cnt = M$이면 출력하고 종료한다.
>   e. $cnt_1 \ge k$ 이면 5로 이동한다.
> 7. $cnt < M$ 이면 5로 이동한다. $cnt = M$ 이면 출력하고 종료한다.

비트 배열을 재사용함에 있어서 재사용되는 비트 배열을 위해서 이 때 사용되는 소수의 곱이 필요하다. 이 소수의 곱셈 시간이 bit array 를 다시 만들 때 사용되는 나눗셈보다 시간이 적게 걸린다. 소수의 곱셈은 곱셈에 필요한 개수만큼 더 시간이 걸린다.

소수의 곱을 활용하기 위해서는 임의의 짝수 값이 필요하다. 이 임의의 짝수 값이 얼마나 크냐에 따라서 안정성이 달라질 것이다. 작은 소수의 곱은 생성하고자 하는 소수 비트보다 작아야 한다. 생성하고자 하는 소수 비트에서 작은 소수 곱의 비트를 뺀만큼 비트를 임의로 생성하면 된다. 그리고 소수 곱을 특정 값보다 작은 모든 소수를 곱하는 것이 아니라 그 집합 내에서 임의성을 두어서 곱한다면 안정성은 더욱 증가할 것이다. 표 1 은 512 비트에서의 곱셈 개수를 나타낸다.

표 1. 작은 소수 곱셈에 걸리는 시간(ms)

소수 곱셈 개수(R 크기)	1536bit	1920bit	2560bit	3200bit
67(64bit)	1.92	1.96	2.03	2.09
60(128bit)	1.71	1.75	1.81	1.88

R: 512 비트에서 작은 소수 곱의 비트를 뺀 비트

시뮬레이션 환경은 ARM922-T 프로세서 (200 MHz)이고 MIRACL 라이브러리를 이용하여 bit array 크기가 3200 일 때 512bit 소수를 생성한 결과는 (그림 1) 과 같다.

고속 소수 생성기를 ARM 프로세서에서 구현할 때 ARM 프로세서는 데스크톱과는 달리 매우 제한적인 환경에서 동작하는 것이므로 메모리를 체크하여 메모리 사용의 효율성을 검증하는 것이 필요하다. 알고리즘 1 ~ 7 까지 ROM 과 RAM, stack 과 heap 사용량을 조사하여 다음의 표에 나타내었다. 기본 알고리즘은 소수 판별알고리즘을 포함하지 않는 프로그램의 골격의 메모리 사용량을 나타낸다. 작은 소수 집합이 1836 개이고 bit array 크기는 3200 일 때 메모리 사용량은 다음과 같다. 알고리즘 4 와 7 을 비교하면 ROM 부분은 0.67% 증가로 표 2 과 같다. Heap 과 stack 합은 3.8% 증가한다. 이것은 stack 부분에서 많은 메모리를 사용하기 때문이다.

IV. 결론

소수 생성 알고리즘에서 가장 중요한 부분은 primality test oracle 의 호출 수를 줄이는 것이다. Primality test oracle 호출을 줄이게 되면 큰 소수를 생성하는데 필요한 시간이 감소되게 된다. 호출을 줄이기 위한 방법으로는 작은 소수 집합으로 미리 나누거나 혹은 작은 소수 집합을 이용한 방법들이 제안되었다. 비트 배열을 사용하는 기존의 알고리즘은 생성된 비트 배열을 단일 소수 생성에 사용 후 폐기하지만, 생성된 비트 배열을 반복적으로 사용하는 제안하는 알고리즘은 비트 배열의 생성에 소요되는

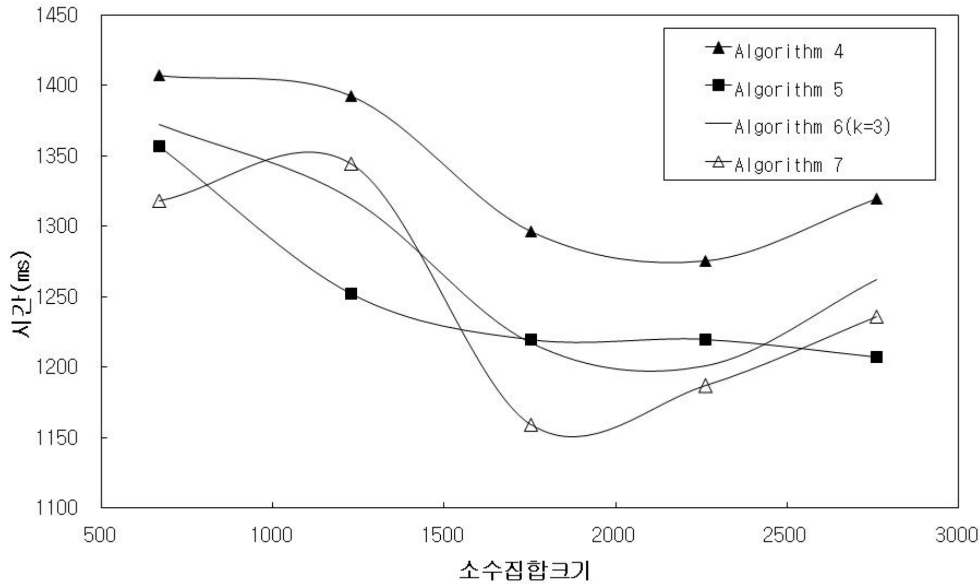

그림 1 비트 배열 크기는 3200에서의 알고리즘 4,5,6,7

표 2. 소수 판별 알고리즘에 따른 메모리 사용량(Byte)
(소수 집합 크기 1836, bit array 크기 3200)

Algorithm	ROM	RAM	Heap	Stack	총합(Heap + Stack)
기본	23556	1000	24517	1152	25669
1	76052	1004	24933	1664	26597
2	76124	1004	24933	1664	26597
3	76624	1004	39977	1728	41705
4	76888	1004	39977	3488	43465
5	76884	1004	39977	3488	43465
6	76964	1004	39977	3520	43497
7	77404	1004	39977	5152	45129

연산량을 줄임으로써 다수의 소수 생성 시간을 단축시킬 수 있었다. 이 때 더 많은 메모리를 사용한다는 단점이 있다.

V. 감사의 글

본 논문은 SK Telecom 의 출연에 의한 재정지원으로 이루어졌습니다.

VI. 참고문헌

[1] 류대걸 역, "C와 C++로 구현하는 암호화 알고리즘," 인포북, 2003.
[2] A. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*, NewYork: CRC press.
[3] William Stallings, *Cryptography and Network Security: Principles and Practice, 2nd*, Prentice Hall.
[4] J. Brandt and I.B. Damgard, "On generation of probable primes by incremental search," in *Proc. CRYPTO' 92*, pp. 358-369.
[5] W. Diffie and M.E. Hellman, *New directions in cryptography*, IEEE Transactions on Information Theory 22 (1976) 644- 654.
[6] E. Bach, J. Shallit, *Algorithmic Number theory, Foundations of Computing Volume I: Efficient Algorithms*, the MIT Press, Cambridge, MA, 1996.
[7] C. Lu, A. L. M. Dos Santos, *A note on Efficient Implementation of Prime Generation Algorithms in Small Portable Devices*, Computer Networks, vol. 49, no. 4, 15 Nov. 2005, pp. 476-491.
[8] C. K. Koc and C. Paar, Eds., "*Cryptographic hardware and embedded systems*," in *Proc LNCS*, vol. 1965, pp. 340-354.
[9] N. Nedjah, L. de M. Mourelle, "*Parallel computation of modular exponentiation for fast cryptography*," Int. J. High Performance Syst. Arch., vol. 1, no.1, pp. 44-49, 2007.