gcd 연산을 이용한 조합 소수 검사 알고리즘의 분석 및 최적화

서동우○ 조호성 박희진
한양대학교 정보통신대학 정보보호 및 알고리즘 연구실
easternseo@gmail.com○ ustog@hanmail.com hjpark@hanyang.ac.kr

Analysis and Optimization of the Combined Primality Test Using gcd Operation

Dongwoo Seo○ Hosung Jo Heejin Park
ISA Lab, The College of Information & Communications, Hanyang University

요 약

큰 소수를 빠르게 생성하기 위한 다양한 소수 검사 방법이 개발되었으며, 가장 많이 쓰이는 소수 검사 방법은 trial division과 Fermat (또는 Miller-Rabin) 검사를 조합한 방법과 gcd 연산과 Fermat (또는 Miller-Rabin) 검사를 조합한 방법이다. 이 중 trial division과 조합한 방법에 대해서는 확률적 분석을 이용하여 수행시간을 예측하고 수행시간을 최적화 하는 방법이 개발되었다하지만, gcd 연산과 조합한 방법에 대해서는 아무런 연구결과도 제시되어 있지 않다.본 논문에서는 gcd 연산을 이용한 조합 소수 검사 방법에 대해 확률적 분석을 이용하여 수행시간을 예측하고 수행시간을 최적화 하는 방법을 제안한다.

1. 서론

암호학에서는 크기가 큰 소수를 생성하는 것이 매우 중요하다. 그 이유는 사용하는 소수의 크기가 클수록 암호시스템의 보안성을 높일 수 있기 때문이다. 실제로 RSA [1]나 ElGmal [2] 같은 암호시스템이나 DSS [3] 같은 서명 구조들은 높은 보안성을 제공하기 위해 큰 소수를 이용할 것을 요구하고 있다 하지만, 큰 소수를 생성하는 것은 높은 연산비용을 요구하기 때문에 적은 연산비용으로 큰 소수를 생성하는 연구가 진행되고 있다.

소수 생성 알고리즘은 임의의 홀수 난수를 생성하는 과정과 생성된 난수가 소수인지를 판단하는 소수 검사 과정으로 구성된다. 이중 난수 생성 과정은 전체 수행 시간 중에서 아주 적은 부분만을 차지하는 반면, 소수 검사는 대부분의 시간을 차지하고 있다. 따라서 빠른 소수 생성 알고리즘의 개발을 위해서는 효율적인 소수 검사를 개발하는 것이 필요하다.

소수 검사는 크게 두 가지 종류로 구분된다. 하나는 결정적 소수 검사이고 다른 하나는 확률적 소수 검사이다. 결정적 소수 검사는 검사를 통과한 난수가 소수임을 1의 확률로 보장해 준다. 결정적 소수 검사로는 trial division [4], Pocklington's test [5], elliptic curve analogue [6], Jacobi sum test [7], Maurer's algorithm [8], Shawe-Taylor's algorithm [9]등이 있다. 결정적 소수 검사는 확실한 소수를 얻을 수 있지만, 수행 속도가 매우 느리다는 단점이 있다. 확률적 소수 검사는 검사를 통과한 난수가 소수임을 높은 확률로 보장해 주는 방법이다. 이 확률은 아주 높으며, 아주 큰 수 s에 대해 '$1-1/2^s$' 이다. 실제로 확률적 소수 검사는 거의 정확한 소수를 얻을 수 있으며, 결정적 소수 검사보다 빠르다. 대표적인 확률적 소수 검사로는 Fermat test [10], Miller-Rabin test [11, 12], Solovay-Strassen test [13], Frobenius-Grantham primality test [14]와 Lehmann primality test [15]등이 있으며, 그 중 Fermat test와 Miller-Rabin test가 널리 쓰인다.

소수 검사들은 각각 장단점이 있기 때문에 실제 구현에서는 소수 검사의 속도를 향상시키기 위해 여러 개의 소수 검사를 조합하여 사용한다. 널리 쓰이는 조합 소수 검사 방법은 trial division을 Fermat (또는 Miller-Rabin) 검사와 조합하는 방법과 gcd 연산을 Fermat (또는 Miller-Rabin) 검사와 조합하는 방법이다. Maurer [8]는 trial division을 이용한 조합 소수 검사 방법에 대해 수행시간을 예측하는 확률적 모델을 제시하였다 또한 그 조합 소수 검사가 가장 빠른 시간에 수행되도록 trial division에 사용되는 소수의 개수를 결정하는 방법을 제시하였다. 그러나, gcd 연산을 이용한 조합 소수 검사 방법에 대해서는 아직 수행시간 예측 모델이 제시되지 않았으며 수행시간 최적화 방법도 제안되지 않았다.

본 논문에서는 gcd 연산을 이용한 조합 소수 검사 방법에 대해서 확률적 분석을 이용하여 수행시간을 예측하는 모델을 제시하고 이 조합 소수 검사가 가장 빠른 시간에 수행되도록 gcd 연산에서 사용되는 소수의 개수를 결정하는 방법을 제시한다.

본 논문은 다음과 같이 구성되어 있다. 2장에서는 소수 검사를 소개하고 3장에서는 이전 연구인 조합 소수 생성 알고리즘을 소개한다. 4장에서는 gcd 연산을 이용한 조합 소수 생성 알고리즘의 수행시간 분석 모델을 제시하고 수행시간 최적화 방법을 제안한다. 마지막으로 5장에서 결론을 내린다.

2. 소수 검사 소개

2.1. Trial division, gcd 검사, Fermat 검사

Trial division은 난수 r이 주어졌을 때, $\sqrt{r}$ 보다 작거나 같은 모든 소수들로 나누어보는 방법이다. 하지만 r이 큰 경우에는 수행시간이 매우 느리므로 단독으로 사용되기는 어려운 방법이

다. 따라서 다른 소수 검사와 조합하여 사용되는 것이 일반적이며 이 경우 trial division에 사용되는 소수의 개수 k를 제한하여 사용한다.

gcd 연산은 두 수 a, b가 주어졌을 때 두 수의 최대공약수를 구하는 연산이다. 이 gcd 연산을 이용하면 주어진 난수 r이 소수인지 검사할 수 있는데 이 방법은 난수 r과 $\sqrt{r}$ 보다 작거나 같은 모든 소수들을 곱한 값과의 gcd를 계산하여 그 값이 1인지를 확인한다. 만약 그 값이 1이면 r은 $\sqrt{r}$ 보다 작거나 같은 모든 소수들과 서로 소가 되므로 r은 소수가 된다. 하지만, gcd 연산도 매우 느리기 때문에 일반적으로 다른 소수 검사와 조합하여 사용되며 곱해지는 소수의 개수 k를 제한하여 사용한다.

Fermat 검사는 Fermat 소정리를 사용한다. Fermat 소정리는 p가 소수이고 a가 p와 서로소인 양의 정수라면 $a^{p-1} \equiv 1 \bmod p$가 성립한다는 것이다. 이를 이용한 Fermat 검사는 주어진 난수 r에 대해 다른 난수 a를 생성해서 $a^{r-1} \equiv 1 \bmod r$이 성립하면 난수 r을 소수로 판정하는 방법이다. Fermat 검사는 trial division과는 달리 단독으로 사용될 수 있다. 하지만 실제 구현에서는 속도를 향상시키기 위하여trial division이나 gcd 연산과 조합되어 사용된다.

2.2. 조합 소수 검사

Trial division과 Fermat 검사를 조합한 소수 검사는 다음과 같다.

조합 소수 생성 (n)
1. 난수 발생
2. Trial division
3. Fermat 검사

먼저 홀수 난수 r을 생성하고 trial division을 수행한다. Trial division은 난수 r을 정해진 수 g보다 작거나 같은 모든 소수 $p_1, p_2, p_3, \cdots, p_k$들로 나누어 본다. 이 소수들 중 r을 나누는 소수가 존재하면 난수 발생 단계로 되돌아가고 그렇지 않으면 r에 대한 Fermat 검사를 수행한다. 난수 r이 Fermat 검사를 통과하면 난수 r을 소수로 출력하고 그렇지 않으면 처음의 난수 발생 단계로 되돌아간다. 이 조합 소수 검사의 목적은 trial division을 Fermat 검사 이전에 수행함으로써 시간이 많이 걸리는 Fermat 검사의 횟수를 감소시키고자 하는 것이다.

gcd 연산과 Fermat 검사를 조합한 소수 검사는 다음과 같다.

조합 소수 생성 (n)
1. 난수 발생
2. gcd 연산
3. Fermat 검사

이 조합 소수 검사는 앞의 조합 소수 검사에서 trial division을 gcd 연산으로 대체한 방법이다. 즉 난수 r을 정해진 수 g보다 작거나 같은 모든 소수 $p_1, p_2, p_3, \cdots, p_k$들로 나누어 보는 trial division 과정을 수행하는 대신 난수 r과 $p_1, p_2, p_3, \cdots, p_k$를 곱한 값과의 gcd를 구하는 방법이다. gcd의 결과 값이 1이면 Fermat 검사를 수행하고 그렇지 않으면 처음의 난수 발생 단계로 되돌아간다.

3. 이전 연구

Maurer [8]는 확률적 분석을 이용하여 trial division과 Fermat 검사의 조합 소수 검사의 수행시간 예측 모델을 제시하였으며 그 내용은 다음과 같다.

Trial division과 Fermat 검사를 이용하여 1개의 소수를 생성하는 데 걸리는 시간은 소수를 생성하기까지 난수를 반복 생성해야 하는 평균횟수와 그 난수로 소수 검사를 하는데 걸리는 시간의 곱으로 나타난다. 전체 수행시간을 T_{total}이라고 하고, 소수 검사의 평균횟수를 N_{Test}, 난수가 소수인지 검사하는데 걸리는 시간을 T_{Test}라고 하면 다음과 같이 표현할 수 있다.

$$T_{Total} = N_{Test} \cdot T_{Test}$$

난수 r이 n비트 정수인 경우 N_{Test}는 다음의 식을 통해서 구할 수 있다 [8].

$$N_{Test} = \frac{\ln 2}{2} \cdot n = 0.347 \cdot n$$

T_{Test}는 난수 r을 발생하는 시간, trial division 시간, Fermat 검사를 수행하는 시간의 합이다. 이들을 각각 T_{RND}, T_{TD}, T_{FT}라고 하면 T_{Test}는 다음과 같다.

$$T_{Test} = T_{RND} + T_{TD} + T_{FT}$$

T_{RND}는 실제 측정을 통해 값을 얻을 수 있다. T_{TD}는 나눗셈을 하는 횟수 N_{div}와 나눗셈을 하는데 걸리는 시간인 T_{div}의 곱이다. T_{FT}는 Fermat 검사를 한번 수행할 때 걸리는 시간 T_{ft}와 Fermat 검사를 수행할 확률 P_{ft}의 곱으로 구해진다. T_{div}와 T_{ft}는 측정을 통해서 얻을 수 있고 N_{div}와 P_{ft}는 g에 대해서 다음 수식을 통해 구할 수 있다 [8].

$$N_{div}(g) = 1 + \sum_{3 \le q \le g} \prod_{3 \le p \le q} (1 - \frac{1}{p})$$

$$P_{ft}(g) = \prod_{3 \le p \le g} \left(1 - \frac{1}{p}\right)$$

따라서 g보다 작은 소수를 사용한 trial division과 Fermat 검사를 이용한 조합 소수 검사의 수행 시간은 다음과 같이 정리된다.

$$\begin{aligned} T_{total}(n,g) = 0.347n \cdot & (T_{rnd} \\ & + T_{div}(1 + \sum_{3 \le q \le g} \prod_{3 \le p \le q} (1 - \frac{1}{p})) \\ & + T_{ft} \prod_{3 \le p \le g} (1 - \frac{1}{p})) \end{aligned}$$

전체 수행 시간은 입력인자 g 값에 따라 달라지는데, 그 이유는 trial division 의 수행횟수와 Fermat 검사를 수행할 확률이 g 값에 따라 변하기 때문이다. 따라서 전체 수행 시간을 최적화하는 g 값인 g_{opt} 를 구할 필요가 있다. g_{opt} 는 나눗셈을 수행하는 데 걸리는 시간 T_{div} 와 모듈러 멱승을 수행하는 데 걸리는 시간 T_{exp} 를 측정하여 다음의 식에 대입하면 구할 수 있다.

$$g_{opt} = \frac{T_{exp}}{T_{div}}$$

4. 연구 내용

본 논문에서는 gcd 연산과 Fermat 검사를 조합한 소수 검사에 대해서 확률적 분석을 이용하여 수행시간을 예측하는 모델을 제시하고 이 조합 소수 검사가 가장 빠른 시간에 수행되도록 gcd 연산에서 사용되는 소수의 개수를 결정하는 방법을 제시한다.

먼저 gcd 연산과 Fermat 검사를 이용하여 1개의 소수를 생성하는 데 걸리는 시간은 생성되는 난수의 평균 개수 N_{Test} 와 난수를 하나 생성하여 소수 검사를 하는데 걸리는 시간 T_{Test} 의 곱으로 나타낼 수 있다. T_{Test} 는 난수 발생 시간 T_{RND}, gcd 연산 시간 T_{GCD}, Fermat 검사 시간 T_{FT} 의 합이므로 전체 수행시간은 다음과 같이 표현할 수 있다.

$$T_{Total} = N_{Test} \cdot (T_{RND} + T_{GCD} + T_{FT})$$

N_{Test}, T_{RND}, T_{FT} 는 3장에서 구한 값들과 같으므로 난수의 길이가 n 비트이고 gcd 연산에서 k 개의 소수를 사용했을 때의 수행시간 $T_{total}(n, k)$ 는 다음과 같다.

$$T_{total}(n,k) = 0.347n\left(T_{rnd} + T_{\gcd(k)} + T_{ft}\prod_{i=1}^{k}\left(1-\frac{1}{p_i}\right)\right) \quad \textbf{수식 (1)}$$

따라서 위의 식에서 gcd 연산에 걸리는 시간만을 분석하면, gcd 연산과 Fermat 검사를 이용한 조합 소수 검사의 수행 시간을 예측할 수 있다.

4.1. gcd 연산

gcd 를 구하는 알고리즘은 Euclid의 gcd 알고리즘과 J.Stein 에 의해 발견된 Binary GCD 알고리즘이 있다. 먼저 Euclid 알고리즘은 gcd(a, b) = gcd(b, a mod b) 라는 성질을 이용한 것으로서 다음과 같이 재귀호출을 이용하여 gcd 를 구한다.

EUCLID (a, b)
1. b 가 0이면 a 를 반환.
2. b 가 0이 아니면, EUCLID (b, a mod b)를 호출.

Binary GCD 알고리즘은 나눗셈 연산을 사용하지 않고 뺄셈 연산과 시프트 연산을 사용하여 gcd 를 구하는 알고리즘이며 다음과 같다.

BinaryGCD (a, b)
1. $a = b$ 라면 return a
2. $a < b$ 라면 swap(a, b)
3. a, b 모두 홀수이고 $a > b$ 이면,
   BinaryGCD (a, b) = BinaryGCD (($a - b$)/2, b)
4. a 가 홀수이고 b 가 짝수이면,
   BinaryGCD (a, b) = BinaryGCD (a, b/2)
5. a 가 짝수이고 b 가 홀수이면,
   BinaryGCD (a, b) = BinaryGCD (a/2, b)
6. a, b 모두 짝수이면,
   BinaryGCD (a, b) = BinaryGCD (a/2, b/2)

Euclid 알고리즘과 Binary GCD 알고리즘은 각각의 장단점이 있기 때문에 실제 구현에서는 두 알고리즘을 조합하여 사용한다. Euclid 알고리즘은 두 수가 거의 같은 크기의 비트수를 가질 때까지 사용되며, 비트수의 크기가 거의 같게 되면 Binary GCD 를 이용한다. 이 조합 GCD 알고리즘의 의사코드는 다음과 같다.

GCD (a, b)
1. b 가 0이면 a 를 반환 한다.
2. a 와 b 의 비트수 차이가 2 이상이면, GCD (b, a mod b)
3. a 와 b 의 비트수 차이가 2 이하이면, BinaryGCD (a, b)

4.2. 조합 GCD 검사의 수행 시간 분석

조합 GCD 검사의 수행시간은 Euclid 함수를 수행하는 시간과 BinaryGCD 를 수행하는 시간의 합으로 표현되며, 소수의 개수인 k 의 값에 따라 변화한다.

$$T_{\gcd(k)} = T_{Euclid} + T_{BGCD}$$

일반적인 Euclid 함수는 모듈러 연산을 반복적으로 수행하여 두 수의 최대공약수를 구하지만, 조합 GCD의 경우에는 모듈러 연산은 입력받은 두 수의 비트수를 비슷하게 맞추는 역할만을 수행한다. Euclid(a, b) 함수는 한 번의 나눗셈 연산을 수행하므로 나눗셈에 대한 수행시간으로 예측할 수 있으며, 나눗셈에 대한 시간 복잡도는 a 를 b 로 나눌 때의 몫을 q 라 하면 $O((1 + \log q)\log b)$로 나타난다. BinaryGCD(a, b) 함수는 뺄기연산과 쉬프트 연산을 반복수행하며 a, b 중 큰 수의 비트수에 비례한다. 즉 $a > b$ 일 경우 $O(\log a)$ 이다 [4].

따라서 $T_{\gcd}$ 는 입력되는 두 수의 비트 크기에 좌우된다. 입력되는 두 수는 난수 r 과 $p_1p_2...p_k$ (이하 Π_k) 이며, 난수 r 은 n 비트로 고정되고 Π_k 는 k 가 변함에 따라 커지거나 작아지므로 k 가 변함에 따라 두 수의 대소 관계가 달라진다. 따라서 $T_{\gcd}$는 r 이 Π_k 보다 큰 경우와 r 이 Π_k 보다 작은 경우로 나누어 분석을 해야 한다.

r 이 Π_k 보다 더 큰 경우, 조합 GCD의 수행시간은 $T_{Euclid}(r, \Pi_k)$를 수행하는 시간과 $T_{BGCD}(r \bmod \Pi_k, \Pi_k)$를 수행하는 시간의 합이 된다.

$$T_{\gcd(k)} = T_{Euclid}(r, \Pi_k) + T_{BGCD}(r \bmod \Pi_k, \Pi_k)$$

$T_{Euclid}(r, \Pi_k)$ 은 나눗셈을 수행하므로 나눗셈의 시간복잡도가 $O((1+\log q)\log b)$ 이고 $q = \lfloor a/b \rfloor$ 라는 사실로부터 $T_{Euclid}(r, \Pi_k)$ 의 시간복잡도는 다음과 같다.

$$\begin{aligned} &O((1+\log q)\log \Pi_k) \\ &\quad = O\left(\left(1+\log\left(\frac{r}{\Pi_k}\right)\right)\log \Pi_k\right) \\ &\quad = O\left(\log \Pi_k + \log \Pi_k \log r - (\log \Pi_k)^2\right) \end{aligned}$$

즉, 시간복잡도는 $\log \Pi_k$ 에 대해서 $\log r/2$ 부근에서 위로 볼록한 2차식의 꼴로 나타난다. 다음은 r 이 512비트일 때 Π_k를 16에서 512비트까지 변화시키면서 r 을 Π_k 로 나누는 시간의 변화를 보이고 있다. 그 결과 256비트 부근에서 위로 볼록한 2차 함수 모양을 보이고 있음을 확인할 수 있다.

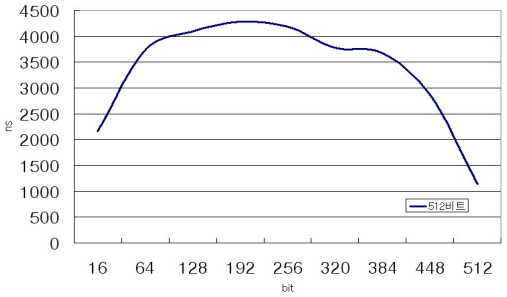

그림 1 Π_k 크기에 따른 나눗셈 수행시간의 변화량

따라서 $T_{Euclid}(r, \Pi_k)$ 의 수행시간은 다음과 같이 나타낼 수 있다.

$$T_{Euclid} = x(\log \Pi_k)^2 + y(\log \Pi_k) + z, \quad x < 0$$

$T_{BGCD}(r \bmod \Pi_k, \Pi_k)$ 의 수행시간은 $r \bmod \Pi_k < \Pi_k$이므로 $O(\log \Pi_k)$이고, 따라서 다음과 같이 $\log \Pi_k$의 1차식으로 표현할 수 있다.

$$T_{BGCD} = u(\log \Pi_k) + v$$

그러므로 r 이 Π_k 보다 큰 경우에 조합 GCD 검사의 수행 시간은 다음과 같이 정리된다.

정리 1.
n 비트의 난수 r 과 k 개의 소수들의 곱으로 이루어진 Π_k 에 대해 $r > \Pi_k$ 일 때, 조합 GCD 연산을 수행하는데 걸리는 시간은 다음과 같다.

$$T_{\gcd(k)} = T_{Euclid}(r, \Pi_k) + T_{BGCD}(r \bmod \Pi_k, \Pi_k)$$

$$T_{Euclid}(r, \Pi_k) = x(\log \Pi_k)^2 + y(\log \Pi_k) + z$$
$$T_{BGCD}(r \bmod \Pi_k, \Pi_k) = u(\log \Pi_k) + v$$

이 때, $\Pi_k = p_1 p_2 \ldots p_k$, p_i 는 소수이고, $x < 0$, y, z, u, v는 계수이다.

r 이 Π_k 보다 작을 경우에는 조합 GCD의 수행시간은 Eculid (Π_k, r) 를 수행하는 시간과 BinaryGCD$(r, \Pi_k \bmod r)$ 을 수행하는 시간의 합이 된다.

$$T_{\gcd(k)} = T_{Euclid}(\Pi_k, r) + T_{BGCD}(r, \Pi_k \bmod r)$$

r 이 Π_k 보다 작을 경우, Euclid(Π_k, r)의 시간복잡도는 다음과 같다.

$$\begin{aligned} &O((1+\log q)\log r) \\ &\quad = O\left(\left(1+\log\left(\frac{\Pi_k}{r}\right)\right)\log r\right) \\ &\quad = O\left(\log r + \log r \log \Pi_k - (\log r)^2\right) \end{aligned}$$

$\log r$ 은 n 으로 일정하므로, 상수로 볼 수 있다. 따라서 Euclid(Π_k, r) 는 $\log \Pi_k$ 에 대한 1차식으로 표현된다.

$$T_{Euclid} = u'(\log \Pi_k) + v'$$

BinaryGCD$(r, \Pi_k \bmod r)$ 은 두 수중 큰 수인 r 의 비트수에 비례하는데 r 이 n 비트로 일정하므로 상수시간 c가 된다.

$$T_{BGCD} = c$$

따라서 r 이 Π_k 보다 작은 경우에 조합 GCD 검사의 수행 시간은 다음과 같이 정리된다.

정리 2.
n 비트의 난수 r 과 k 개의 소수들의 곱으로 이루어진 Π_k 에 대해 $r < \Pi_k$ 일 때, 조합 GCD 연산을 수행하는데 걸리는 시간은 다음과 같다.

$$T_{\gcd(k)} = T_{Euclid}(\Pi_k, r) + T_{BGCD}(r, \Pi_k \bmod r)$$

$$T_{Euclid}(r, \Pi_k) = u'(\log \Pi_k) + v'$$
$$T_{BGCD}(r \bmod \Pi_k, \Pi_k) = c$$

이 때, $\Pi_k = p_1 p_2 \ldots p_k$, p_i 는 소수, c는 상수이고 u', v' 계수이다.

4.3. 확률적 분석값과 실제 수행시간의 비교

이 예측값이 어느 정도 정확한지 확인하기 위해 gcd를 이용한 조합 소수 검사 알고리즘을 구현하고 512비트 소수를 생성하는 데 걸리는 시간을 측정하였다.

실험 환경은 펜티엄 4 3Ghz CPU와 2GB 메모리를 탑재한 시스템이고 운영체제는 윈도우 XP 기반으로 프로그래밍 언어는 J2SE 5.0을 사용하였다. 실험 방법은 512비트 소수를 100,000번 생성하여 그 평균 시간을 계산하였다.

gcd 연산과 Fermat 검사를 이용한 조합 소수 검사 알고리즘의 수행시간을 예측하려면 난수 r 이 Π_k 보다 큰 경우와 작은 경우에 대해 각각 정리 1과 정리 2의 (x, y, z, u, v)와 (u', v', c)의 값을 구해야 한다.

먼저 r 이 Π_k 보다 더 큰 경우, 수행시간은 정리 1에 따르면 $T_{Euclid}(r, \Pi_k)$와 $T_{BGCD}(r \bmod \Pi_k, \Pi_k)$을 모두 구해야 한다. 하지만,

$T_{Euclid}(r, \Pi_k)$은 $T_{BGCD}(r \bmod \Pi_k, \Pi_k)$에 비해 무시할 수 있을 정도로 매우 작다. 다음 표는 n이 256, 512, 1024비트일 때 $T_{Euclid}(r, \Pi_k)$와 $T_{BGCD}(r \bmod \Pi_k, \Pi_k)$를 측정한 것이다.

표 1 T_{Euclid}와 T_{BGCD}의 수행시간 비교

	256비트	512비트	1,024비트
T_{Euclid} (ns)	961~1,462	1,146~4,284	1,581~10,804
T_{BGCD} (ns)	36,773	102,568	314,373

따라서 이 경우 조합 GCD의 수행시간은 $T_{BGCD}(r \bmod \Pi_k, \Pi_k)$만 고려하며 다음의 $\log\Pi_k$에 대한 1차식의 u, v를 구하여 예측한다.

$$T_{gcd(k)} = u(\log\Pi_k) + v$$

r이 Π_k보다 작은 경우, 상수 c값은 실제 측정을 통해서 구할 수 있으므로, 이 경우에도 T_{gcd}의 수행시간은 $\log\Pi_k$에 대한 1차 방정식으로 표현이 가능하다.

$$T_{gcd(k)} = u'(\log\Pi_k) + v'$$

그러므로 두 식의 계수인 (u, v)와 (u', v')의 값을 구하면, 정리 1과 정리 2로부터 조합 GCD 검사의 수행시간을 예측할 수 있다. (u, v)와 (u', v') 값은 직접 측정을 구할 수 있지만, 모든 비트크기에 대해서 실험을 수행하는 것은 무리가 있기 때문에 두 가지 경우에 대해 각 4개의 표본지점을 취하여 회귀직선을 구하였다. 그 결과 실제 측정을 통해 구한 (u, v)와 (u', v') 값과 매우 유사한 결과를 얻었다. 다음은 최종적으로 구해진 조합 GCD의 수행 시간 예측식이다.

$$T_{gcd(k)} = \begin{cases} 201.3916\left(\log\prod_{i=1}^{k} p_i\right) - 3993.05 & (r > \Pi_k) \\ 17.17009\left(\log\prod_{i=1}^{k} p_i\right) + 96861.38 & (r < \Pi_k) \end{cases} \quad \text{수식 (2)}$$

위의 식을 수식 (1)에 대입하면 전체 수행 시간에 대한 예측 모델을 구할 수 있으며, 수식에 사용되는 T_{RND}와 T_{ft}를 측정하여 대입하면 예측 수행시간을 구할 수 있다. 다음 표는 512비트 소수 생성 시 전체 수행시간의 예측값과 실제 측정값을 비교한 결과이다.

표 2 전체 수행시간의 예측값과 실제 측정값의 비교

r	Π_k	예측값 (ns)	실측값 (ns)	오차(%)
512	37	270,978,926	271,522,585	0.3
	64	240,388,771	240,388,973	0.9
	128	213,770,583	215,331,396	0.8
	256	192,186,375	192,909,076	0.4
	509	182,804,861	185,047,987	1.3
	1028	169,294,994	171,142,247	1.1
	2046	195,741,444	161,713,761	1.3
	4096	157,520,997	159,420,791	1.2
	8101	158,036,091	159,975,743	1.3

다음은 전체 수행시간의 예측값과 측정값의 비교를 그래프로 나타낸 것이다.

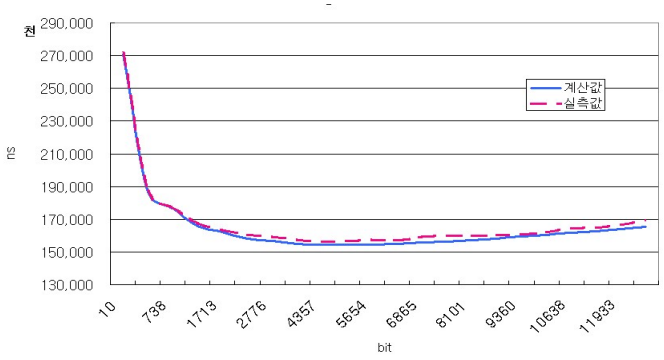

그림 2 전체 수행시간의 예측값과 측정값의 비교

전체 수행 시간 예측값과 실측값의 오차는 0.3%~1.3%로 매우 낮으며 따라서 gcd 연산과 Fermat 검사를 이용한 조합 소수 생성 알고리즘의 전체 수행 시간에 대한 분석과 예측이 잘 수행되었음을 알 수 있다.

4.4. 최단 수행시간을 구하기 위한 k_{opt}의 계산

gcd 연산과 Fermat 검사를 이용한 조합 소수 생성 알고리즘의 수행 시간은 감소하다가 다시 증가하는 형태를 가지고 있기 때문에 최적 수행시간이 존재함을 알 수 있다. 따라서 512비트 소수를 생성할 때, 수행 시간 예측 모델로부터 gcd 연산을 이용한 조합 소수 생성 알고리즘이 최단 시간에 수행되게 하는 k값을 k_{opt}라고 할 때, k_{opt}는 다음 식을 만족하는 정수이다.

$$T_{total}(k+1) - T_{total}(k) = 0$$

위의 식을 전개하여 T_{gcd}와 T_{ft}에 대한 식으로 정리하면 다음과 같이 나타낼 수 있다.

$$0.347n(T_{rnd} + T_{gcd(k+1)} + T_{ft}\prod_{i=1}^{k+1}(1-\frac{1}{p_i})) = 0.347n(T_{rnd} + T_{gcd(k)} + T_{ft}\prod_{i=1}^{k}(1-\frac{1}{p_i}))$$

좌변과 우변을 다시 정리하면, 다음과 같이 나타낼 수 있다.

$$T_{gcd(k+1)} - T_{gcd(k)} = T_{ft}(\prod_{i=1}^{k+1}\left(1-\frac{1}{p_i}\right) - \prod_{i=1}^{k}\left(1-\frac{1}{p_i}\right))$$

$$a(\log\prod_{i=1}^{k+1} p_i) + b - a(\log\prod_{i=1}^{k} p_i) - b = T_{ft}\frac{1}{p_{k+1}}\left(\prod_{i=1}^{k}\left(1-\frac{1}{p_i}\right)\right)$$

$$\frac{a}{T_{ft}} = \frac{1}{p_{k+1}\log(p_{k+1})}\left(\prod_{i=1}^{k}\left(1-\frac{1}{p_i}\right)\right)$$

정리 3.
조합 GCD 연산과 Fermat 검사를 이용한 조합 소수 생성 알고리즘은 k가 다음과 같은 식을 만족할 때, 최적 시간에 수행된다.

$$\frac{a}{T_{ft}} \simeq \frac{1}{p_{k+1}\log(p_{k+1})}\left(\prod_{i=1}^{k}\left(1-\frac{1}{p_i}\right)\right)$$

이 때, a는 조합 GCD 검사의 수행시간이 $a(\log \Pi_k)+b$ 일 때, 1차항의 계수를 나타내고, T_{ft}는 Fermat 검사를 수행하는 시간이다. p_i는 소수를 나타낸다.

k가 위와 같은 조건을 만족하면, 최적 수행시간이 된다. 결국 최적 수행시간을 구할 수 있는 k_{opt} 값은 a 와 T_{ft}에 의해 결정됨을 알 수 있다. T_{ft}는 측정을 통해 구할 수 있으며, a 값은 r이 Π_k보다 작은지 큰지에 따라 변하므로 k_{opt} 또한 두 경우로 나누어 생각해보아야 한다.

본 논문의 실행 환경에서 r이 512비트인 경우, k_{opt}값은 다음과 같이 구할 수 있다. r이 Π_k보다 큰 경우에는 수식 (2)에서 r이 Π_k보다 큰 경우의 a 값을 T_{ft}로 나누고, 그 몫이 정리 3의 수식을 만족하게 하는 k 값을 구하면 되며, 이 수식을 만족하는 k_{opt} 값은 92가 된다. 하지만, 소수 92개의 곱의 비트크기는 512비트보다 크다. 이는 r이 Π_k보다 크다는 조건에 위배되므로 유효하지 않은 결과이다. r이 Π_k보다 작은 경우, 동일한 방식으로 구해보면, k_{opt} 값은 463이고 소수 463개의 곱은 512비트보다 크므로 유효한 범위 내에 있는 결과이다 따라서 512비트 소수를 생성하는 경우 k_{opt}의 예측값은 463이며, 이때 최적 수행시간으로 수행될 것으로 예측된다.

다음 그래프와 표는 이 예측값을 측정값과 비교한 결과를 보여주고 있다. 측정한 k_{opt} 위치는 예측한 k_{opt}와 일치하며 수행시간의 오차는 2% 정도이다.

표 3 k_{opt}에서의 예측값과 실측값의 비교

	예측값(ns)	실측값(ns)	오차(%)
전체 수행 시간	154,445,787	157,533,956	2.0

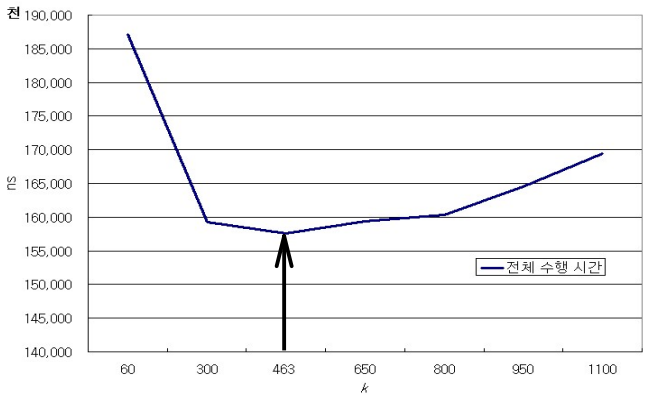

그림 3 k_{opt}에서의 측정값

5. 결론

본 논문에서는 gcd 연산을 이용한 조합 소수 생성 알고리즘의 수행시간을 분석하여 수행 시간 예측 모델을 제시하였다 또한 이 수행 시간 예측 모델을 이용하여 최적 수행시간을 구하기 위한 k_{opt} 값을 정하는 방법을 제안하였다. 마지막으로 수행시간의 예측값과 k_{opt} 값을 실제 측정값과 비교하여 본 논문의 예측 모델이 상당히 정확함을 보였다.

6. 참고문헌

[1] R.L. Rivest, A. Shamir and L. Adleman, A method for obtaining digital signatures an public-key cryptosystems, *Communications of the ACM* 21(2) pp.120-126 (1978)

[2] T. ElGmal, A public key cryptosystem and a signature scheme based on discrete logarithms, *IEEE Transactions on Information Theory* 31(4), pp.469-472 (1985)

[3] National Institute for Standards and Technology, Digital Signature Standard(DSS), *Fedral Register* 56 169 (1991)

[4] T.H. Cormen, C.E. Leiserson, R.L. Rivest and C. Stein, Introduction to Algorithms, 2nd ed, MIT press (1991)

[5] H.C. Pocklington, The determination of the prime or composite nature of large numbers by Fermat's theorem, *Proc. of the Cambridge Philosophical Society* 18, pp.29-30 (1914)

[6] A.O.L. Atkin and F. Morain, Elliptic curves and primality proving, *Mathematics of Computation* 61, pp.29-63 (1993)

[7] W. Bosma and M.P. van der Hulst, Faster primality testing, *CRYPTO'89, LNCS* 435, pp.652-656 (1990)

[8] U.M. Maurer, Fast Generation of Prime Numbers and Secure Public-Key Cryptographic Parameters, *Journal of Cryptology* 8(3), pp.123-155 (1995)

[9] J. Shawe-Taylor, Generating strong primes, *Electronics Letters* 22(16), pp.875-877 (1986)

[10] A.J. Menezes, P.C. van Oorschot, and S.A. Vanstone, Handbook of Applied Cryptography, CRC Press, (1997)

[11] G.L. Miller, Riemann's Hypothesis and Tests for Primality, *Journal of Computer Systems Science* 13(3), pp.300-317 (1976)

[12] M.O. Rabin, Probabilistic Algorithm for Primality Testing, *Journal of Number Theory* 12, pp.128-138 (1980)

[13] R. Solovay and V. Strassen, A fast Monte-Carlo test for primality, *SIAM Journal on Computing* 6, pp.84-85 (1977)

[14] J. Grantham, A probable prime test with high confidence, *Journal of Number Theory* 72, pp.32-47 (1998)

[15] D.J. Lehmann, On primality tests, *SIAM Journal of Computing* 11(2), pp.374-375 (1982)