JPV 소수 생성 알고리즘의 확률적 분석 및 성능 개선

(Probabilistic Analysis of JPV Prime Generation Algorithm and its Improvement)

박 희 진 † 조 호 성 ††
(Heejin Park) (Hosung Jo)

요 약 Joye와 연구자들은 기존의 조합 소수 판단 검사에서 trial division 과정을 제거한 새로운 소수 생성 알고리즘 (이하 JPV 알고리즘)을 제시하였으며, 이 알고리즘이 기존의 조합 소수 생성 알고리즘에 비해 30~40% 정도 빠르다고 주장하였다. 하지만 이 비교는 전체 수행시간이 아닌 Fermat 검사의 호출 횟수만을 비교한 것으로 정확한 비교와는 거리가 있다. 기존의 조합 소수 생성 알고리즘에 대해 이론적인 수행시간 예측 방법이 있음에도 불구하고 두 알고리즘의 전체 수행시간을 비교할 수 없었던 이유는 JPV 알고리즘에 대한 이론적인 수행 시간 예측 모델이 없었기 때문이다.

본 논문에서는 먼저 JPV 알고리즘을 확률적으로 분석하여 수행시간 예측 모델을 제시하고, 이 모델을 이용하여 JPV 알고리즘과 기존의 조합 소수 생성 알고리즘의 전체 수행시간을 비교한다. 이 모델을 이용하여 펜티엄4 시스템에서 512비트 소수의 생성 시간을 예측해 본 결과 Fermat 검사의 호출 횟수를 이용한 비교와는 달리 JPV 알고리즘이 기존의 조합 소수 생성 알고리즘보다 느리다는 결론을 얻었다. 이러한 이론적인 분석을 통한 비교는 실제 동일한 환경에서 실험을 통해서 검증되었다. 또한, 본 논문에서는 JPV 알고리즘의 성능 개선 방법을 제시한다. 이 방법을 사용하여 JPV 알고리즘을 개선하면 동일한 공간을 사용할 경우에 JPV 알고리즘이 기존의 조합 소수 생성 알고리즘과 비슷한 성능을 보인다.

키워드 : 소수 생성, 소수 판단 검사, 공개키 암호시스템, RSA

Abstract Joye et al. introduced a new prime generation algorithm (JPV algorithm hereafter), by removing the trial division from the previous combined prime generation algorithm (combined algorithm hereafter) and claimed that JPV algorithm is 30~40% faster than the combined algorithm. However, they only compared the number of Fermat-test calls, instead of comparing the total running times of two algorithms. The reason why the total running times could not be compared is that there was no probabilistic analysis on the running time of the JPV algorithm even though there was a probabilistic analysis for the combined algorithm.

In this paper, we present a probabilistic analysis on the running time of the JPV algorithm. With this analytic model, we compare the running times of the JPV algorithm and the combined algorithm. Our model predicts that JPV algorithm is slower than the combined algorithm when a 512-bit prime is generated on a Pentium 4 system. Although our prediction is contrary to the previous prediction from comparing Fermat-test calls, our prediction corresponds to the experimental results more exactly. In addition, we propose a method to improve the JPV algorithm. With this method, the JPV algorithm can be comparable to the combined algorithm with the same space requirement.

Key words : Prime generation, Primality test, Public-key cryptosystems, RSA

† 종신회원 : 한양대학교 정보통신대학 정보통신학부 컴퓨터전공 교수
hjpark@hanyang.ac.kr
†† 학생회원 : 한양대학교 정보통신학과
ustog@naver.com
논문접수 : 2006년 12월 29일
심사완료 : 2007년 12월 3일

정보과학회논문지: 시스템 및 이론 제35권 제2호(2008.2)

1. 서 론

소수는 1과 자기 자신으로만 나누어지는 정수인데 암호학에서는 크기가 큰 소수를 생성하는 것이 매우 중요하다. 왜냐하면, 사용하는 소수의 크기가 크면 클수록 보안성이 높아지기 때문이다. 실제로 RSA[1]나 ElGmal [2]같은 공개키 암호시스템[3-6]이나 DSS[7]같은 서명 구조에서 높은 보안성을 제공하기 위해 큰 소수들을 이용할 것을 요구하고 있다[8]. 하지만, 소수의 크기가 커질수록 더 높은 연산 비용이 요구되기 때문에 효율적인 소수생성은 중요한 연구과제이다.

소수 생성 알고리즘은 임의의 난수(홀수)를 생성하는 난수 생성 과정과 생성된 난수가 소수인지 판단하는 소수 판단 검사 과정으로 구성된다. 이중에서 난수 생성 과정은 전체 수행시간의 아주 적은 부분만을 차지하고 소수 판단 검사 과정이 전체 수행시간의 대부분을 차지하고 있다. 따라서 효율적인 소수 생성 알고리즘을 개발하기 위해서는 효율적인 소수 판단 검사를 개발하는 것이 중요하다.

소수 판단 검사는 두 가지로 나누어진다. 하나는 결정적 소수 판단 검사이고 다른 하나는 확률적 소수 판단 검사이다. 결정적 소수 판단 검사는 검사를 통과한 난수가 소수임을 1의 확률로 보장해 주는 방법이다. 결정적 소수 판단 검사로는 trial division[9], Pocklington's test[10], elliptic curve analogue[11], Jacobi sum test [12], Maurer's algorithm[13], Shawe-Taylor's algorithm[14]등이 있다. 결정적 소수 판단 검사는 확실한 소수를 얻을 수 있는데 비해, 수행 속도가 매우 느리다는 단점이 있다. 확률적 소수 판단 검사는 검사를 통과한 난수가 소수임을 높은 확률로 보장해 주는 방법이다. 이 확률은 아주 높으며, 아주 큰 수 s에 대해 '$1-1/2^s$'이다. 확률적 소수 판단 검사는 실제로 거의 정확한 소수를 얻을 수 있으며, 결정적 소수 판단 검사보다 빠르다. 대표적인 확률적 소수 판단 검사로는 Fermat test [15], Miller-Rabin test[16, 17], Solovay-Strassen test [18], Frobenius-Grantham primality test[19]와 Lehmann primality test[20] 등이 있으며, 그 중 Fermat 검사와 Miller-Rabin 검사가 널리 쓰인다.

실제 구현에서는, 소수 생성의 속도를 향상시키기 위해 소수 판단 검사를 조합하여 사용한다. 널리 쓰이는 조합으로는 trial division과 확률적 소수 판단 검사의 조합으로 확률적 소수 판단 검사에는 Fermat 검사나 Miller-Rabin 검사가 쓰인다. Trial division은 난수 r을 $\sqrt{r}$보다 작거나 같은 모든 소수들로 나누어보는 방법이다. 하지만 $\sqrt{r}$보다 작은 모든 소수로 나누어 볼 경우, 수행시간이 매우 느리므로, 조합 소수 판단 검사에서는 trial division에 사용되는 소수들의 개수를 제한하여 주어진 어떤 정수 g보다 작은 소수들만으로 나누어 본다. Fermat 검사는 주어진 난수 r보다 작고 1보다 큰 하나의 임의의 수 a를 생성하고 $a^{r-1}=1 \pmod r$이 성립하는지 검사하여 성립하게 되면 r을 소수로 판단하는 검사이다. Fermat 검사는 소수가 아닌 수를 항상 통과시키는 문제가 있으며, 이런 수를 Carmichael 수라고 부른다[21]. 하지만 Carmicheal 수의 갯수가 아주 적기 때문에 Fermat 검사는 실제로 OpenSSL과 같은 곳에서 널리 쓰인다[22]. 예를 들어 r이 512비트의 수이고 Fermat 검사를 통과했을 때, r이 소수가 아닐 확률은 10^{-20}보다 작다[23]. Miller-Rabin 검사는 Fermat 검사를 개선한 방법으로 $r-1=2^j q$ $(0 \leq j \leq k)$이라고 했을 때, j가 0인 경우, $a^q \bmod r = 1$이 성립하거나 혹은 어떤 j가 $a^{2^j q} \bmod r = n-1$을 만족하는 경우, r을 소수로 판단하는 검사 방법으로 Carmichael 수를 통과시키지 않는다. 다음은 조합 소수 판단 검사를 의사코드로 나타낸 것이다.

표 1 조합 소수 판단 검사

조합 소수 판단 검사 (r)
1. r에 대해 g보다 작은 소수들만으로 trial division을 수행한다.
2. Trial division을 통과한 r에 대해 확률적 소수 판단 검사를 수행한다.

Maurer[14]는 trial division과 확률적 소수 판단 검사의 조합 소수 판단 검사에 대해 확률적 분석을 이용한 수행시간 예측 모델을 제시하였다. 또한 조합 소수 판단 검사가 가장 빠른 시간에 수행되도록 trial division에 사용되는 g값을 효율적으로 선택하는 방법을 제시하였다.

2000년 Joye와 연구자들은 기존의 조합 소수 생성 알고리즘에서 trial division 과정을 제거한 새로운 소수 생성 알고리즘(이하 JPV 알고리즘)을 제시하였다[24]. 이 알고리즘은 두 가지 특징을 가지고 있는데 첫 번째 특징은 난수를 생성할 때 임의의 난수가 아닌 작은 소수들의 곱과 서로 소인 난수를 생성하는 것이며 두 번째 특징은 여러 개의 난수를 매 번 생성하지 않고 처음 생성한 난수에 간단한 연산을 수행하여 새로운 난수를 얻는 것이다. 이 두 가지 특징 때문에 trial division이 없더라도 효율적인 소수 생성을 할 수 있다.

Joye와 연구자들은 JPV 알고리즘과 기존의 조합 소수 생성 알고리즘의 비교 결과를 제시하였고 이 비교에 따르면 JPV 알고리즘이 30~40%정도 빠른 것으로 나타났다. 하지만 이 비교는 전체 수행시간을 비교하지 않고, Fermat 검사의 호출횟수만을 비교한 것으로 정확한

비교라고 하기 어렵다. 두 알고리즘의 전체 수행시간을 비교할 수 없었던 이유는 조합 소수 생성 알고리즘에 대해서는 확률적 분석에 의한 전체 수행시간 예측 모델이 이미 존재하지만, JPV 알고리즘에 대해서는 확률적 분석에 의한 전체 수행시간 예측 모델이 없었기 때문이다. 따라서 정확한 비교를 위해서는 먼저 JPV 알고리즘에 대해 전체 수행시간 예측 모델을 개발하고 이를 이용하여 비교하여야 한다.

본 논문에서는 먼저 JPV 알고리즘을 확률적으로 분석하여 수행시간 예측 모델을 제시하고, 이 모델을 이용하여 JPV 알고리즘과 기존의 조합 소수 생성 알고리즘의 전체 수행시간을 비교한다. 이 모델을 이용하여 펜티엄4 시스템에서 512비트 소수의 생성 시간을 예측해 본 결과 Fermat 검사의 호출 횟수를 이용한 비교와는 달리 JPV 알고리즘이 기존의 조합 소수 생성 알고리즘보다 느리다는 결론을 얻었다. 이러한 이론적인 분석을 통한 비교는 실제 동일한 환경에서 실험을 통해서 검증되었다. 또한, 본 논문에서는 JPV 알고리즘의 성능 개선 방법을 제시한다. 이 방법을 사용하여 JPV 알고리즘을 개선하면 동일한 공간을 사용할 경우에 JPV 알고리즘이 기존의 조합 소수 생성 알고리즘과 비슷한 성능을 보인다.

본 논문의 구성은 다음과 같다. 2장에서는 먼저 이전 연구인 조합 소수 생성 알고리즘과 JPV 알고리즘을 소개한다. 3장에서는 JPV 알고리즘의 확률적 분석 모델을 제시하고, 이 모델을 이용하여 조합 소수 생성 알고리즘과 비교한다. 또한 JPV 알고리즘의 성능향상을 위한 개선방안을 제시한다. 4장에서는 결론을 내린다.

2. 이전 연구

2.1 조합 소수 생성 알고리즘

조합 소수 생성 알고리즘은 난수 생성 과정, trial division 과정, 확률적 소수 판단 검사 과정으로 나누어진다. 확률적 소수 판단 검사 과정은 Fermat 검사나 Miller-Rabin 검사가 주로 사용된다. 다음은 n비트의 소수를 생성하는 조합 소수 생성 알고리즘이다.

표 2 조합 소수 생성 알고리즘

조합 소수 생성 (*n*)
1. 난수 발생
   n비트 난수(홀수) r을 발생시킨다.
2. Trial division
   r을 g보다 작은 모든 소수들로 나누어 본다.
   어떤 소수로도 나누어지지 않으면, 3번 과정으로 진행한다.
   하나의 소수에 의해서라도 나누어지면, 1번 과정을 되돌아간다.
3. 확률적 소수 판단 검사
   Trial division을 통과한 r에 대해 Fermat 검사나 Miller-Rabin 검사를 수행한다.
   r이 이 검사를 통과하면, 소수로 판단하고, 아니면 1번 과정으로 되돌아간다.

Maurer는 조합 소수 생성 알고리즘의 수행시간을 확률적으로 분석하였으며, 그 내용은 다음과 같다[13]. 소수 하나를 생성하는 걸리는 전체 시간을 T_{total}이라고 하면, T_{total}은 조합 소수 생성 알고리즘을 한번 수행하는 시간 T_{comb}과 발생되는 난수의 평균 개수 N_{comb}의 곱이 되므로, 다음과 같이 표현할 수 있다.

$$T_{total} = T_{comb} N_{comb} \tag{1}$$

홀수 난수 r이 n비트 정수일 때, N_{comb}는 다음의 식을 통해서 구할 수 있다.

$$N_{comb}(n) = \frac{(\ln 2^n)}{2} \approx 0.347n \tag{2}$$

T_{comb}는 각 과정의 소요 시간들의 합으로 나타낼 수 있다. 먼저 난수 발생 과정에서 난수 한 개를 발생시키는 시간을 T_{rnd}로 정의하고 trial division 과정에서 나눗셈을 한번 하는데 걸리는 시간을 T_{div}, 총 나눗셈 횟수를 N_{div}로 정의한다. 그리고 확률적 소수 판단 검사 과정에서 검사를 한번 수행하는 데 걸리는 시간과 확률적 소수 판단 검사를 수행할 확률을 각각 T_{ppt}와 P_{ppt}로 정의하면, 다음과 같이 T_{comb}를 표현할 수 있다.

$$T_{comb} = T_{rnd} + T_{div} N_{div} + T_{ppt} P_{ppt} \tag{3}$$

위 식 (3)에서 T_{rnd}, T_{div}, T_{ppt}는 실험을 통해 실측값을 구할 수 있으며, N_{div}와 P_{ppt}는 소수의 특성과 알고리즘의 확률적인 분석방법을 통해 이론적인 예측값을 다음과 같이 구할 수 있다[13].

$$P_{ppt}(g) = \prod_{\substack{3 \le p \le g \\ p\text{는 소수}}} \left(1 - \frac{1}{p}\right) \tag{4}$$

$$N_{div}(g) = 1 + \sum_{\substack{3 \le q \le g \\ q\text{는 소수}}} \prod_{\substack{3 \le p \le q \\ p\text{는 소수}}} (1 - \frac{1}{p}) \tag{5}$$

식 (3)에 식 (4)와 식 (5)를 대입하여 T_{comb}를 구하고, 이 T_{comb}와 식 (2)의 N_{comb}값을 식 (1)에 대입하면 T_{total}은 다음과 같다.

$$\begin{aligned} T_{total}(n,g) &= 0.347n \cdot T_{comb} \\ &= 0.347n \cdot (T_{rnd} + T_{div}(1 + \sum_{\substack{3 \le q \le g \\ q\text{는 소수}}} \prod_{\substack{3 \le p \le q \\ p\text{는 소수}}} (1 - \frac{1}{p})) \\ &\quad + T_{ppt} \prod_{\substack{3 \le p \le g \\ p\text{는 소수}}} (1 - \frac{1}{p})) \end{aligned} \tag{6}$$

$T_{total}(n,g)$는 g값이 증가함에 따라 점점 감소하다가 다시 증가하는 아래로 볼록한 그래프의 모양을 가지는데 이는 trial division의 수행횟수와 확률적 소수 판단 검사를 수행할 확률 사이의 상관관계 때문이다. 다음 그림 1은 256bit의 소수를 생성할 때, 사용하는 소수의 개

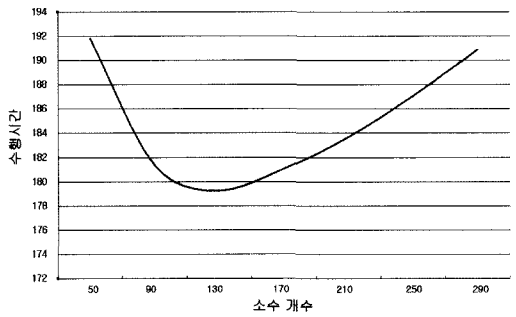

그림 1 소수의 수가 증가함에 따른 전체수행시간의 변화

수가 증가함에 따른 전체수행시간의 변화를 나타낸 것이다.

나눗셈의 횟수가 1번씩 증가할 때마다 확률적 소수 판단 검사를 수행할 확률이 낮아지므로 전체 수행시간은 줄어들게 된다. 하지만, 나눗셈의 횟수가 너무 많아질 경우에는 나눗셈을 수행하는 시간이 더 걸리기 때문에 전체수행시간은 증가하게 된다. 이 때 전체 수행시간을 최소가 되게 하는 g값을 g_{opt}라고 한다.

Maurer는 전체수행시간이 최소가 되는 경우는 나눗셈 연산 시간의 증가와 멱승 연산 시간의 감소가 균형을 이룰 때로 나눗셈을 한번 수행하는 시간과 멱승연산을 한번 수행하는 데 예상되는 시간이 같아지는 지점을 뜻한다. 여기서 멱승연산을 한번 수행하는데 예상되는 시간은 g보다 큰 소수 중 가장 작은 수를 q라고 하면, $(1/q)T_{exp}$이 된다. 여기서 g와 q의 크기는 큰 차이가 나지 않으므로 $1/q \approx 1/g$가 된다. 식으로 표현하면, $T_{div} = (1/g)T_{exp}$이 되고 이를 정리하면 g_{opt}를 구하는 식 (7)을 얻을 수 있다[13].

$$g_{opt} = \frac{T_{exp}}{T_{div}} \tag{7}$$

2.2 JPV 알고리즘

Joye와 연구자들은 기존의 조합 소수 생성 알고리즘에서 trial division 과정을 제거한 새로운 소수 생성 알고리즘을 제시하였다. JPV 알고리즘[24]은 전처리 계산, 바탕값 생성, 후보 생성, 소수 판단 검사, 바탕값 재생성 과정으로 나누어지며, 각 과정은 다음과 같다.

1. **전처리 계산:** 알고리즘에 필요한 정수값 η, Π, ρ, $\lambda(\Pi)$를 미리 계산한다. 생성될 소수 q가 n비트라고 하면, q의 범위는 $2^{n-1}+1 \leq q \leq 2^n-1$이 된다. 여기서 $2^n-1 = W_{max}$, $2^{n-1}+1 = W_{min}$로 정의한다. 이 때, η값은 k개의 서로 다른 홀수 소수들의 곱이며, Π와 ρ는 η의 배수이면서, $\Pi \leq W_{max} - W_{min}$과 $\rho \geq W_{min}$을 만족하는 값이다. $\lambda(\Pi)$는 Π에 대한 Carmichael 함수[25]로서, $\Pi = \prod_{i=1}^{k} p_i^{\delta_i}$ 라고 하면, $\lambda(\Pi)$는 각 $\lambda(p_i^{\delta_i})$의 최소공배수가 된다. p_i가 홀수인 경우, $\lambda(p_i^{\delta_i}) = p_i^{\delta_i-1}(p_i-1)$로 계산하고, 짝수이고, δ_i가 3이상일 경우, $\lambda(2^{\delta_i})=2^{\delta_i-2}$이고 짝수이고 3보다 작을 경우, $\lambda(2)=1$, $\lambda(4)=2$로 계산한다.
2. **바탕값 계산:** Π와 서로 소인 c를 생성한다. 먼저 Π보다 작은 n비트의 난수 c를 생성하고, c와 Π가 서로 소인지를 검사한다. 서로 소이면, c를 반환하고, 그렇지 않으면, c에 1을 더하여 새로운 c를 생성하고 다시 검사한다. Π와 서로 소인 c를 찾을 때까지 이 과정을 반복한다. 이 알고리즘에서는 다음 식 $c^{\lambda(\Pi)} \bmod \Pi = 1$ 이 성립하면, c와 Π가 서로 소로 판단한다.
3. **후보 생성:** 생성된 바탕값을 이용하여 소수 판단 검사에 사용될 후보 q를 생성한다. 후보 q는 바탕값 c와 전처리 계산값 ρ의 합으로 구한다. 여기서 c와 ρ가 모두 홀수가 되어 q가 짝수가 될 경우, q에 η값을 더하여 홀수로 만들어준다.
4. **소수 판단 검사:** 생성된 후보 q에 Fermat 검사를 수행한다. q가 소수로 판단되면, q를 반환하고 종료한다. 그렇지 않으면, 다음 과정으로 진행한다.
5. **바탕값 재생성:** 새로운 바탕값 c'을 생성한다. 새로운 바탕값은 다음 식 $c' = 2c \bmod \Pi$를 계산하여 구한다. 바탕값 c'이 재생성되면 '3. 후보 생성' 과정으로 되돌아간다. '3. 후보 생성' 과정에서는 c'으로 c를 대체하여 후보 q를 생성하는데 사용한다.

Joye와 연구자들은 JPV 알고리즘을 기존의 조합 소수 생성 알고리즘과 비교하였다[24]. 비교에 사용한 조합 소수 생성 알고리즘은 trial division에 10개의 작은 소수를 사용했으며, 성능 비교의 기준으로는 소수 판단 검사 호출 횟수를 이용하였다. 이 비교에 따르면, JPV 알고리즘이 10개의 소수를 이용한 기존의 조합 소수 생성 알고리즘에 비해 30%~40% 정도 빠르며, 생성하는 소수의 비트수가 커질수록 더 좋은 성능을 보이는 것으로 나타났다.

하지만, 이 비교 방법은 두 가지 문제점을 가지고 있다. 첫 번째 문제점은 JPV 알고리즘과 조합 소수 생성 알고리즘을 비교할 때, 조합 소수 생성 알고리즘의 trial division에서 사용하는 작은 소수들의 개수를 10개로 제한한 것이다. 일반적으로 소수의 개수를 늘리면, 소수 판단 검사의 호출 횟수는 줄어들기 때문에 비교대상인 조합 소수 생성 알고리즘에서 사용하는 작은 소수들의 개수를 10개로 제한한 것은 적절하지 않다. 두 번째 문제점은 소수 판단 검사의 호출횟수만을 가지고 성능을 비교할 수 없다는 사실이다. 왜냐하면, trial division에서 많은 소수를 이용하면 할수록, 소수 판단 검사의 호출횟수는 줄어들지만, trial division에서 소요되는 시간이 많아져서 전체 수행속도가 증가할 수도 있기 때문이다.

3. 연구 내용

먼저 3.1절에서는 JPV 알고리즘의 확률적 분석 모델을 제시하고 3.2절에서는 이 모델을 이용하여 JPV 알고리즘과 조합 소수 생성 알고리즘을 비교하며 3.3절에서는 JPV 알고리즘의 성능 개선 방안을 제시한다.

3.1 JPV 알고리즘의 확률적 분석

JPV 알고리즘이 하나의 소수를 생성하는 걸리는 시간은 1. 전처리 계산 과정은 제외하고, 2. 바탕값 생성, 3. 후보 생성, 4. 소수 판단 검사, 5. 바탕값 재생성 과정의 합으로 예측할 수 있다. 각 항에서 걸리는 시간을 T_i로 나타내고, 각 항의 반복 횟수를 N_i로 나타내면, 전체 수행시간 T_{JPV}는 다음과 같다.

$$T_{JPV} = T_2N_2 + T_3N_3 + T_4N_4 + T_5N_5 \quad (8)$$

각 항의 반복 수행 횟수에 대해 알아보면, 과정 2는 단 한번만 수행되므로 N_2=1이고, 과정 3과 4는 같은 횟수만큼 반복되므로 N_3=N_4이고, 과정 5는 확률적 소수 판단 검사가 실패했을 때만 수행되므로, N_5=N_4 -1이 된다. 따라서 식 (8)은 다음과 같이 나타낼 수 있다.

$$T_{JPV} = T_2 + (T_3 + T_4)N_4 + T_5(N_4 - 1) \quad (9)$$

각 과정에 걸리는 시간을 알아보면 다음과 같다. 먼저, T_2는 과정 2에서 c를 생성하고 c와 Π가 서로 소인지 확인하고 서로 소가 아닌 경우 c와 1을 더해 새로운 c를 만드는 시간의 합으로 표현된다. n비트 난수 c를 생성하는데 걸리는 시간을 T_{rnd}, ($c^{\lambda(\Pi)}$ mod Π)를 계산하는데 걸리는 시간 T_{ramda}, 덧셈을 수행하는 데 걸리는 시간을 T_{add}, ($c^{\lambda(\Pi)}$ mod Π)를 계산하는 횟수를 N_{ramda}라고 하면, 과정 2를 수행하는데 걸리는 시간은 다음과 같다.

$$\begin{aligned} T_2 &= T_{rnd} + T_{ramda}N_{ramda} + T_{add}\,(N_{ramda} - 1) \quad (10)\\ &= T_{rnd} + (T_{ramda} + T_{add})N_{ramda} - T_{add} \end{aligned}$$

T_3은 바탕값을 이용하여 홀수 후보를 생성하는데 걸리는 시간이다. 과정 3에서는 덧셈을 반드시 한 번 수행하고, q가 짝수일 경우 추가로 덧셈을 한 번 더 수행한다. q가 짝수일 확률은 1/2이므로, 평균 1.5번의 덧셈을 수행한다.

$$T_3 = 1.5T_{add} \quad (11)$$

T_4는 한 후보에 대해 확률적 소수 판단 검사를 수행하는 시간이며 T_{ppt}로 나타낼 수 있다.

$$T_4 = T_{ppt} \quad (12)$$

T_5는 후보가 소수가 아닐 경우 새로운 바탕값 c를 생성하기 위해 수행되는 시간이다. 과정 5의 ($c \leftarrow 2c$ mod Π)의 연산은 ($c + c$) mod Π로 대체되며, ($c + c$)의 결과가 Π보다 작은 경우에는 1번의 덧셈 연산만을 수행하게 되고, ($c + c$)의 결과가 Π보다 크면 Π를 빼는 연산을 추가로 수행하게 된다. $2c$가 Π보다 클 확률은 1/2이고, 일반적으로 뺄셈 연산과 덧셈 연산의 수행시간은 비슷하므로, 과정 5는 평균 1.5번의 덧셈을 수행한다고 할 수 있다.

$$T_5 = 1.5T_{add} \quad (13)$$

식 (10)~(13)를 식 (8)에 대입하여 정리하면 다음과 같다.

$$\begin{aligned} T_{JPV} &= T_2 + (T_3 + T_4)N_4 + T_5(N_4 - 1)\\ &= (T_{rnd} + (T_{ramda} + T_{add})N_{ramda} - T_{add})\\ &\quad + (1.5\,T_{add} + T_{ppt})N_4 + 1.5\,T_{add}(N_4 - 1)\\ &= T_{rnd} + (N_{ramda} + 3N_4 - 2.5)T_{add} + N_{ramda}T_{ramda}\\ &\quad + N_4T_{ppt} \end{aligned} \quad (14)$$

위의 식 (14)에서 T_{rnd}, T_{ramda}, T_{add}, T_{ppt}값은 측정을 통해 구할 수 있으며, N_{ramda}과 N_4는 확률적 분석을 통해 다음과 같이 구할 수 있다.

N_{ramda}는 $c^{\lambda(\Pi)}$ mod Π=1을 계산하는 횟수이므로, 이 식을 만족하는 c가 평균적으로 몇 번의 시도를 거쳐 발견되는 지를 분석하면 된다. $c^{\lambda(\Pi)}$ mod Π=1이 성립하기 위해서는 c와 Π가 서로 소가 되어야 하고, c와 Π가 서로 소가 되기 위해서는 Π의 모든 인자와 c가 서로소여야 한다. c가 k개의 작은 홀수 소수들인 p_1, p_2, ..., p_k들의 곱으로 이루어진 Π와 서로 소일 확률은 $\prod_{i=1}^{k}\left(1-\frac{1}{p_i}\right)$이며, Π와 서로 소인 c를 찾기 위해 시도되어야 할 횟수는 c와 Π가 서로 소일 확률의 역수이므로 N_{ramda}는 다음과 같다.

$$N_{ramda} = \prod_{1 \le i \le k} \left(\frac{1}{1-\frac{1}{p_i}}\right) = \prod_{1 \le i \le k}\left(\frac{p_i}{p_i - 1}\right) \quad (15)$$

N_4는 후보 q가 생성되는 횟수이므로, 후보 q가 소수일 확률의 역수가 된다. 후보 q가 소수일 확률은 Π와 서로 소인 임의의 정수가 소수일 확률이므로 다음과 같이 조건부 확률로 구할 수 있다. 사건 A를 q가 Π와 서로 소가 되는 사건이라 하고, 사건 B를 q가 소수 판단 검사를 통과할 사건이라고 하면, q가 소수가 될 확률은 사건 A가 일어났을 때 사건 B가 일어날 조건부 확률이므로 $\mathbf{P}(B|A)$=$\mathbf{P}(B \cap A)/\mathbf{P}(A)$이고 $\mathbf{P}(B \cap A)$=$\mathbf{P}(B)$이므로 $\mathbf{P}(B|A)$=$\mathbf{P}(B)/\mathbf{P}(A)$가 된다. Π가 k개의 소인수를 가지고 있으므로 $\mathbf{P}(A)$는 $\prod_{i=1}^{k}\left(1-\frac{1}{p_i}\right)$이고, $\mathbf{P}(B)$는 소수를 찾을 때까지 시도해야 하는 평균 횟수의 역수이므로 $\mathbf{P}(B)=\frac{1}{0.347n}$이다. 따라서 $\mathbf{P}(B|A)=\frac{1}{0.347n}/\prod_{i=1}^{k}\left(1-\frac{1}{p_i}\right)$이 되며, N_4는 $\mathbf{P}(B|A)$의 역수이므로 다음과 같이 표현된다.

$$N_4 = 0.347n \cdot \prod_{1 \le i \le k} (1 - \frac{1}{p_i}) \quad (16)$$

최종적으로 식 (15)과 (16)의 N_{ramda}와 N_4를 식 (14)에 대입하면, 다음의 정리를 얻는다.

정리 1. JPV 알고리즘을 이용하여 n비트 소수 하나를 생성하는 시간은 다음과 같다.

$$\begin{aligned} T_{JPV}(n) = & \; T_{rnd} + \prod_{1 \le i \le k} (\frac{p_i}{p_i - 1}) T_{ramda} \\ & + 0.347n \prod_{1 \le i \le k} (1 - \frac{1}{p_i}) T_{ppt} \\ & + (1.041n \prod_{1 \le i \le k} (1 - \frac{1}{p_i}) \\ & + \prod_{1 \le i \le k} (\frac{p_i}{p_i - 1}) - 2.5) T_{add} \end{aligned}$$

이 때, p_1, p_2, ..., p_k는 Π의 소인수이며, T_{rnd}는 난수 하나를 생성하는 시간, T_{add}는 덧셈을 한 번 수행하는 시간이고, T_{ramda}는 $c^{\lambda(\Pi)} \bmod \Pi$를 계산하는 시간, T_{ppt}는 확률적 소수 판단 검사를 한 번 수행하는 시간이다

3.2 확률적 분석값과 실제 수행시간의 비교

본 논문에서 제시한 확률적 분석이 얼마나 정확하게 수행 시간을 예측하는지 확인하기 위해 펜티엄4에서 512비트 소수의 생성시간을 확률적 분석을 통해 예측하고, 그 예측값을 실제 수행시간과 비교하였다. 실험환경은 펜티엄 4 3.0Ghz CPU와 메모리 1GB를 장착한 시스템이며, 프로그래밍 언어로는 자바 JDK 5.0을 사용하였으며, OpenSSL[22]과 GNUCrypto[26]를 참조하였다.

정리 1의 식을 이용하여 512비트 소수를 생성하는 시간을 예측하기 위해서는 T_{rnd}, T_{ramda}, T_{ppt}, T_{add}값이 필요하며, 이들은 측정을 통해 구할 수 있다. 512비트 소수를 생성할 때, 사용되는 η, Π, ρ, $\lambda(\Pi)$값은 다음 표 3과 같으며, 이 값은 Joye와 연구자들에 의해 제안된 값이다[24].

표 3 512비트 소수 생성 시 사용한 전처리 계산값

인자	전처리 계산값
η	b16b d1e0 84af 628f e508 9e6d abd1 6b5b 80f6 0681 d6a0 92fc b1e8 6d82 876e d719 2100 0bcf dd06 3fb9 081d fd07 a021 af23 c735 d52e 63bd 1cb5 9c93 cbb3 98af d
Π	$1729 \cdot \eta$
ρ	$4120 \cdot \eta$
$\lambda(\Pi)$	1dc6 c203 d4cc 7800 33f9 c5d8 d97a a246 8a54 e370 0

Π는 3부터 367사이에 있는 73개의 작은 소수들의 곱으로 이루어져 있으므로 임의의 난수가 Π와 서로 소일 확률은 다음과 같다.

$$\prod_{1 \le i \le 73} \left(1 - \frac{1}{p_i}\right) = 0.188093$$

다음 표 4는 JPV 알고리즘의 수행시간 예측을 위해 필요한 T_{rnd}, T_{ramda}, T_{ppt}, T_{add}를 측정한 것이다. 측정은 512비트 난수에 대해 각 연산을 수행하였으며, 연산 당 1,000,000번을 수행하고 평균을 구하였다. 확률적 소수 판단 검사를 한번 수행하는 데 걸리는 시간인 T_{ppt}는 Miller-Rabin 검사를 한 번 수행하는 데 걸리는 시간으로 측정하였다.

표 4 각 인자의 측정값

입력인자	측정시간(ns)
T_{add}	430
T_{rnd}	15,544
T_{ramda}	1,500,133
T_{ppt}	4,734,181

앞에서 구한 값들을 정리 1의 식에 대입하여 512비트 소수 생성 시간을 예측해 보면 다음과 같다.

$$\begin{aligned} T_{JPV}(512) = & \; 15{,}544 + \frac{1}{0.188093} \cdot 1{,}500{,}133 + 0.347 \cdot 512 \\ & \cdot 0.188093 \cdot 4{,}734{,}181 \\ & + (1.041 \cdot 512 \cdot 0.188093 + \frac{1}{0.188093} - 2.5) \cdot 430 \\ = & \; 15{,}544 + 7{,}975{,}457 + 158{,}203{,}805 + 44{,}319 \\ = & \; 166{,}239{,}125(ns) \end{aligned}$$

이 예측값이 어느 정도 정확한지 확인하기 위해 JPV 알고리즘을 구현하고 512비트 소수를 생성하는 데 걸리는 시간을 측정하였다. 실험 방법은 512비트 소수를 100,000번 생성하여 그 평균을 계산하였다. 다음 표 5는 512비트 소수 생성 시 확률적 분석에 의한 예측값과 측정값을 비교한 것이다.

표 5 512비트 소수 생성 시 예측값과 측정값 비교

	예측값(ns)	실측값(ns)	오차율(%)
전체 수행 시간(ns)	166,239,125	168,336,171	1.2

비교 결과, JPV 알고리즘의 확률적 분석에 의한 예측값과 실제 측정값은 차이가 크지 않으며, 이는 JPV 알고리즘의 확률적 분석이 상당히 정확하게 실제수행시간을 예측함을 의미한다.

3.3 소수 생성 알고리즘의 비교

3.1절에서 제시한 확률적 분석을 이용하여 JPV 알고리즘과 조합 소수 생성 알고리즘을 비교하였다. 먼저 g_{opt}값을 사용하는 최적 조합 소수 생성 알고리즘과 JPV 알고리즘을 비교하였고, JPV 알고리즘이 사용하는 공간과 동일한 크기의 공간을 사용하는 조합 소수 생성 알고리즘과 JPV 알고리즘과 비교하였다.

조합 소수 생성 알고리즘의 최적 수행시간은 Maurer

가 제안한 확률적 소수 생성 알고리즘 분석을 이용하여 구할 수 있다[13]. 최적 수행시간을 예측하기 위해 먼저 T_{exp}와 T_{div}을 측정하여 g_{opt}를 구한다.

표 6 T_{exp}와 T_{div}의 측정값

입력인자	측정시간(ns)
T_{div}	1,894
T_{exp}	4,734,181

$$g_{opt}=\frac{T_{exp}}{T_{div}}=\frac{4,734,181}{1,894}\approx 2,499.56$$

g_{opt} 예측값은 약 2,499이며, 이보다 작은 소수의 개수는 366개로 범위는 3~2,477이다. 소수 366개를 이용하는 조합 소수 생성 알고리즘이 확률적 소수 판단 검사를 수행할 확률 $P_{ppt}(g)$와 trial division에서 수행하는 나눗셈의 횟수 $N_{div}(g)$를 식 (4)와 식 (5)를 이용하여 계산하면 다음과 같다.

$$P_{ppt}(2,499)=\prod_{\substack{3\le p\le 2,499\\ p\text{는 소수}}}^{\infty}\left(1-\frac{1}{p}\right)=0.14318$$

$$N_{div}(2,499)=1+\sum_{\substack{3\le q\le 2,499\\ q\text{는 소수}}}\prod_{\substack{3\le p\le 2,499\\ p\text{는 소수}}}^{\infty}(1-\frac{1}{p})=65.18$$

위에서 계산한 g_{opt}, P_{ppt}, N_{div}와 표 4와 6의 측정값들을 식 (6)에 대입하면, 수행시간의 예측값을 다음과 같이 구할 수 있다.

$$\begin{aligned}T_{total}(512,366)&=(15,544+1,894\cdot 65.18+4,734,181\cdot 0.14318)\\&\quad\cdot 0.347\cdot 512\\&=(15,544+123,450+677,840)\cdot 177.664\\&=145,121,995(ns)\end{aligned}$$

위에서 구한 최적 조합 소수 생성 알고리즘의 예측값과 실제 측정값을 JPV 알고리즘의 예측값과 실제 측정값과 비교해 보면 다음 표 7과 같다.

표 7 JPV 알고리즘과 최적 조합 알고리즘의 비교

	예측값(ns)	실측값(ns)	오차(%)
JPV 알고리즘	166,239,125	168,336,171	1.2
최적 조합 알고리즘	145,121,995	144,968,267	0.1

확률적 모델을 이용하여 예측한 결과 최적 조합 소수 생성 알고리즘의 성능이 13%정도 더 좋은 것으로 예측되었으며, 실제 측정한 결과도 비슷한 성능차이를 보였다.

최적 조합 소수 생성 알고리즘의 수행속도는 JPV 알고리즘의 수행속도보다 빠르지만, 최적 조합 소수 생성 알고리즘이 trial division에 사용하는 작은 소수들을 저장하기 위해 JPV 알고리즘에 보다 공간을 더 사용하는 단점이 있다. 따라서 조합 소수 생성 알고리즘이 JPV 알고리즘과 동일한 공간을 사용한다는 조건으로 다시 비교하였다. 다음 표 8은 최적 조합 소수 생성 알고리즘과 JPV 알고리즘이 사용하는 저장 공간을 비교한 것이다.

표 8 JPV 알고리즘과 최적 조합 알고리즘의 저장공간 비교

	저장 공간(비트)	저장 내용
JPV 알고리즘	1,687	전처리 계산값 (η, Π, ρ, $\lambda(\Pi)$)
최적 조합 알고리즘	5,856	366개의 작은 소수

다음 표 9는 JPV 알고리즘이 사용하는 저장 공간에 대한 세부내용으로 이는 전처리 계산값을 저장하기 위한 공간이다.

표 9 JPV 알고리즘의 저장 공간 요구량

	저장 공간 (비트)
η	500
Π	511
ρ	512
$\lambda(\Pi)$	164
전체 요구 공간	1,687

조합 소수 생성 알고리즘의 trial division에 사용되는 작은 소수들을 동일한 공간에 저장되는 것만큼만 사용하면, 두 알고리즘이 동일한 공간을 사용한다는 조건으로 비교를 할 수 있다. 하나의 소수를 저장하는 공간으로 16비트를 사용하면, 1,687비트 공간에 저장할 수 있는 소수의 수는 105개이며, 범위는 3~577 사이의 소수이다.

조합 소수 생성 알고리즘이 105개의 소수를 사용하는 경우, 확률적 소수 판단 검사를 수행할 확률 $P_{ppt}(g)$와 trial division의 나눗셈 횟수 $N_{div}(g)$를 식 (4)와 (5)를 이용하여 계산하면 다음과 같다.

표 10 105개의 소수를 사용할 경우 N_{div}와 P_{ft}값

	예측값
$N_{div}(g)$	24.74
$P_{ppt}(g)$	0.175619

위의 계산값들을 이용하면, JPV 알고리즘과 동일한 공간을 사용하는 조합 알고리즘의 수행시간을 예측하면 다음과 같다.

$$\begin{aligned}T_{total}(512,105)&=(15,544+1,894\cdot 24.74+4,734,181\\&\quad\cdot 0.175619)\cdot 0.347\cdot 512\\&=(15,544+46,857+831,412)\cdot 177.664\\&=158,798,392(ns)\end{aligned}$$

이전과 동일한 방법으로 실제 수행시간을 측정하여, 같은 크기의 공간을 사용하는 조합 소수 생성 알고리즘

표 11 JPV 알고리즘과 동일 공간을 사용하는 조합 알고리즘의 비교

	예측값(ns)	실측값(ns)	오차율(%)	공간(bit)
JPV 알고리즘	166,239,125	168,336,171	1.2	1,687
최적 조합 알고리즘	145,121,995	144,968,267	0.1	5,856
조합 알고리즘 (동일공간)	158,798,392	158,450,260	0.2	1,687

의 예측값과 실측값을 JPV 알고리즘과 비교하면 다음과 같다.

JPV 알고리즘과 동일한 공간을 사용하는 조합 소수 생성 알고리즘을 비교한 결과 JPV 알고리즘이 조합 소수 생성 알고리즘에 비해 여전히 좋지 않은 것으로 나타났다. 하지만, 수행 속도의 차이는 최적 조합 소수 생성 알고리즘과 비교한 경우에 비해 많이 줄어들었다.

3.4 JPV 알고리즘의 성능 개선

끝으로 JPV 알고리즘의 성능을 개선하기 위한 방안을 제시한다. JPV 알고리즘은 과정 2에서 c와 Π가 서로 소인지를 검사하기 위해 '$c^{\lambda(\Pi)} \bmod \Pi$' 연산을 수행한다. 하지만 이 modular 멱승연산은 시간이 많이 걸리는 연산이기 때문에 이 연산 대신에 시간이 적게 걸리는 Euclid 함수[9]를 사용하면 성능을 개선할 수 있다. Euclid 함수는 두 수의 최대공약수를 구하는 함수인데 $c^{\lambda(\Pi)} \bmod \Pi$ 연산을 Euclid 함수로 대체할 수 있는 이유는 c와 Π의 최대공약수가 1이면 두 수는 서로 소이기 때문이다. 두 수의 최대공약수를 구하는 Euclid 함수의 의사코드는 다음과 같다.

표 12 Euclid 함수

```
EUCLID(a, b)
1. b가 0이면 a를 반환 한다.
2. b가 0이 아니면, EUCLID(b, a mod b)를 재귀 호출한다.
```

개선된 JPV 알고리즘의 수행시간은 원래 JPV 알고리즘의 수행시간에서 Euclid 함수로 대체하는 부분의 수행시간만 달라진다. 따라서 3.2절에서 원래 JPV 알고리즘의 수행시간 $T_{JPV}(512)$를 구하는 과정에서 T_{ramda} 대신 Euclid 함수의 수행시간인 T_{EUC}를 대입하면 개선된 JPV 알고리즘의 수행시간을 구할 수 있다.

표 13을 보면 T_{EUC}는 T_{ramda}에 비해 매우 빠르다는 것을 알 수 있으며 이 T_{EUC}를 이용하여 개선된 JPV 알고리즘의 수행시간을 구하면 다음과 같다.

표 13 T_{EUC}와 T_{ramda}의 측정값

입력인자	측정시간
T_{EUC}	101,198(ns)
T_{ramda}	1,500,133(ns)

$$\begin{aligned} T_{JPV}(512) &= 15{,}544 + 5.3165 \cdot 101{,}198 \\ &\quad + 0.347 \cdot 512 \cdot 0.188093 \cdot 4{,}734{,}181 \\ &\quad + (1.041 \cdot 512 \cdot 0.188093 + 5.3165 - 2.5) \cdot 430 \\ &= 15{,}544 + 538{,}019 + 158{,}203{,}805 + 44{,}242 \\ &= 158{,}801{,}610(ns) \end{aligned}$$

표 14에서는 개선된 JPV 알고리즘의 수행시간을 다른 알고리즘들의 수행시간과 비교하고 있다.

Euclid 함수를 이용하여 JPV 알고리즘을 개선해 본 결과 기존의 JPV 알고리즘에 비해 성능 향상이 있었다. 조합 소수 생성 알고리즘과 비교해 보면 동일한 공간을 사용하는 경우와는 비슷하거나 좀 더 나은 것으로 나타났지만, 최적 조합 소수 생성 알고리즘에 비해서는 여전히 좋지 않은 것으로 나타났다.

4. 결 론

본 논문에서는 JPV 알고리즘을 확률적으로 분석하여 수행시간 예측 모델을 제시하였고, 이 모델을 이용하여 JPV 알고리즘과 조합 소수 생성 알고리즘의 전체 수행시간을 비교하였다. 그 결과 펜티엄4 시스템에서 512비트의 소수를 생성할 때 조합 소수 생성 알고리즘이 JPV 알고리즘보다 더 좋다는 결론을 얻을 수 있었다. 또한 JPV 알고리즘의 성능 개선 방법을 제시하였다. 이 방법을 사용하여 JPV 알고리즘을 개선하면 동일한 공간을 사용할 경우에 JPV 알고리즘이 기존의 조합 소수 생성 알고리즘과 비슷한 성능을 보였다.

참 고 문 헌

[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)

표 14 각 알고리즘의 예측값과 실측값의 비교

	예측값(ns)	실측값(ns)	오차율(%)	공간(bit)
JPV 알고리즘	166,239,125	168,336,171	1.2	1,687
최적 조합 알고리즘	145,121,995	144,968,267	0.1	5,856
조합 알고리즘 (동일공간)	158,798,392	158,450,260	0.2	1,687
개선 JPV 알고리즘	158,801,610	160,167,512	0.9	1,687

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] W. Diffie and M.E. Hellman, New directions in cryptography, IEEE transactions on Information Theory, 22(6), pp. 644-654 (1976).
[4] IEEE P1363: Standard for Public-Key Cryptography (2000).
[5] N. Koblitz, *A Course in Number Theory and Cryptography*, Berlin: Springer (1987).
[6] Public-Key Cryptography Standards, PKCS #1 RSA Cryptography Standard.
[7] National Institute for Standards and Technology, Digital Signature Standard(DSS), *Fedral Register* 56 169 (1991).
[8] International Organization for Standard, ISO/IEC 18032: Prime Number Generation (2005).
[9] T.H. Cormen, C.E. Leiserson, R.L. Rivest and C. Stein, Introduction to Algorithms, 2nd ed, MIT press (1991).
[10] 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).
[11] A.O.L. Atkin and F. Morain, Elliptic curves and primality proving, *Mathematics of Computation* 61, pp. 29-63 (1993).
[12] W. Bosma and M.P. van der Hulst, Faster primality testing, *CRYPTO'89, LNCS* 435, pp. 652-656 (1990).
[13] U.M. Maurer, Fast Generation of Prime Numbers and Secure Public-Key Cryptographic Parameters, *Journal of Cryptology* 8(3), pp. 123-155 (1995).
[14] J. Shawe-Taylor, Generating strong primes, *Electronics Letters* 22(16), pp. 875-877 (1986).
[15] A.J. Menezes, P.C. van Oorschot, and S.A. Vanstone, Handbook of Applied Cryptography, CRC Press, (1997).
[16] G.L. Miller, Riemann's Hypothesis and Tests for Primality, *Journal of Computer Systems Science* 13(3), pp. 300-317 (1976).
[17] M.O. Rabin, Probabilistic Algorithm for Primality Testing, *Journal of Number Theory* 12, pp. 128-138 (1980).
[18] R. Solovay and V. Strassen, A fast Monte-Carlo test for primality, *SIAM Journal on Computing* 6, pp. 84-85 (1977).
[19] J. Grantham, A probable prime test with high confidence, *Journal of Number Theory* 72, pp. 32-47 (1998).
[20] D.J. Lehmann, On primality tests, *SIAM Journal of Computing* 11(2), pp. 374-375 (1982).
[21] R.D. Carmichael, On composite numbers P which satisfy the Fermat congruence $a^{P-1} \equiv 1 \pmod{P}$, *Amer. Math. Monthly* 19, pp. 22-27, 1912.
[22] OpenSSL, http://openssl.org/
[23] C. Pomerance, On the Distribution of Pseudoprimes, *Mathematics of Computation*, 37(156), pp. 128-138, 1981.
[24] M. Joye, P. Paillier, and S. Vaudenay, Efficient Generation of Prime Numbers, *CHES 2000, LNCS* 1965, pp. 340-354 (2000).
[25] H. Riesel, Prime numbers and computer methods for factorization, Boston, Basel, Stuttgart: Birkhäuser, (1985).
[26] The GNU Crypto project, http://www.gnu.org/software/gnu-crypto/

박 희 진

1994년 2월 서울대학교 컴퓨터공학과 졸업(학사). 1996년 2월 서울대학교 컴퓨터공학과 졸업(석사). 2001년 2월 서울대학교 컴퓨터공학과 졸업(박사). 2001년 3월~2003년 2월 서울대학교 컴퓨터연구소 전문연구원. 2003년 3월~2003년 8월 이화여자대학교 컴퓨터학과 BK 조교수. 2004년 9월~현재 한양대학교 정보통신대학 컴퓨터전공 조교수. 관심분야는 컴퓨터 알고리즘, 생물정보학, 암호학, 컴퓨터 보안

조 호 성

2005년 2월 단국대학교 전자컴퓨터공학부 졸업(학사). 2007년 2월 한양대학교 일반대학원 정보통신학과 졸업(석사). 2007년 3월~현재 한양대학교 일반대학원 전자컴퓨터통신공학과 박사과정. 관심분야는 암호칩 설계, 컴퓨터보안, 생물정보학