오퍼랜드 스캐닝 방법을 이용한 다진법 몽고메리 알고리즘에 대한 연구

문상국

목원대학교 전자공학과

Study on High-Radix Montgomery's Algorithm Using Operand Scanning Method

Sangook Moon

Mokwon University, Department of Electronic Engineering

E-mail : smoon@mokwon.ac.kr

요 약

RSA 암호 알고리즘의 고속 연산에 핵심이 되는 법 곱셈 (modular multiplication)을 고속으로 처리하기 위해서 몽고메리 알고리즘이 연구되고 발전되어 왔다. 이 몽고메리 알고리즘에서는 법 곱셈에 나눗셈이 들어가지 않기 때문에 빠른 법 곱셈 연산을 수행할 수 있다. 하지만, 일반 잉여 형태의 숫자를 몽고메리 표현 형태로 변환하고 이후에 결과를 다시 일반 잉여 형태로 변환하는 과정에서 별도로 연산이 필요하게 된다. 1024 비트 이상의 고비도의 RSA 연산을 수행하기 위해서는 키 비트를 워드 단위로 쪼개어 다진법 개념을 도입하여 연산할 수가 있다. 본 논문에서는 몽고메리 알고리즘을 개선시키기 위하여 오퍼랜드 스캐닝 개념을 도입한 방법을 연구하여 비교하였다. 각각의 방법은 최적화에 대한 이슈, 메모리 공간에 대한 이슈, 연산 시간에 대한 이슈를 고려 대상으로 한다.

ABSTRACT

In order for fast calculation for the modular multiplication which plays an essential role in RSA cryptography algorithm, the Montgomery algorithm has been studed and developed in varous ways. Since there is no division operation in the algorithm, it is able to perform a fast modular multiplication. However, the Montgomery algorithm requires a few extra operations in the progress of which transformation from/to ordinary modular form to/from Montgomery form should be made. Concept of high radix operation can be considered by splitting the key size into word-defined units in the RSA cryptosystems which use longer than 1024 key bits. In this paper, We adopted the concept of operand scanning methods to enhance the traditional Montgomery algorithm. The methods consider issues of optimization, memory usage, and calculation time.

키워드

몽고메리알고리즘, RSA, 오퍼랜드 스캐닝

Ⅰ. 서 론

모듈라 곱셈 (modular multiplication)은 기본적으로 이론적인 수학에서 주로 사용되는 연산이었고, 공학에는 응용되는 사례가 많지 않았으나, 현대 암호학이 발달하면서, RSA 암호시스템과 Diffie-Hellman 키 교환 시스템이 모듈라 곱셈을 암호 연산에 도입하면서부터 그 연구에 대한 필요성이 증대되어 고속 모듈라 곱셈에 대한 연구가 발전되어 왔다. 고속 모듈라 곱셈에 대한 연구에서 가장 괄목할만하면서 유용한 것은 단연 몽고메리 알고리즘이다. 몽고메리 알고리즘은 모듈라 곱셈과 모듈라 제곱 연산을 다음과 같은 몽고메리 곱을 사용함으로써 가속화를 가능하게 한다.

$$\text{MonPro}(a, b) = abr^{-1} \bmod n \qquad (1)$$

위 식 (1)에서 a와 b는 n보다 작아야 하고 n과 r은 서로 소여야 한다. 이 알고리즘은 r이 2의 제곱수일때 더욱 강력한 기능을 발휘하여 나눗셈 연산을 쉬프트 연산으로 대치시킬 수 있는 특성을 가지고 있다.

몽고메리 알고리즘을 구현하는 방법이 복잡하기 때문에 그 알고리즘 자체를 구현하기 위한 알고리즘이 다양하게 연구되어 왔다. 몽고메리 알고리즘을 효율적으로 구현하는 데 있어서 핵심적으로 추구해야 할 성능은 연산 속도와 자원 소비의 효율성, 즉 하드웨어 면적이다. 본 논문에서는 몽고메리 알고리즘을 구현하는 데 있어서 적용된 몇 가지의 오퍼랜드 스캐닝을 사용한 방법을 비교해 보고 응용 분야에 필요한 알고리즘을 선택할 수 있도록 장단점을 비교하여 연구하고자 한다.

II. RSA 암호 알고리즘

RSA란 암호화와 인증을 할 수 있는 공개키 암호 시스템이다. 이것은 1977년 Ron Rivest와 Adi Shamir, Leonard Adleman에 의해서 개발되었다. RSA는 대단히 큰 정수의 인수분해가 곤란함을 기반으로 보안성과 전자서명을 제공한다. 이것은 다음과 같은 동작 원리를 가진다.

2.1. RSA의 키 생성 알고리즘

- 각각의 주체 (서로 암호문을 주고받을 대상)는 RSA 공개키 (public key)와 그에 상응하는 비밀키 (private key)를 생성해야 한다.

- 각각의 주체 A는 다음과 같은 과정을 수행해야 한다.

. 개략적으로 비슷한 크기의 큰 임의의 소수 p, q를 생성한다.

. $n = pq$ 와 $\phi = (p-1)(q-1)$를 계산한다.

. 임의의 정수 e 선택

($\gcd(e, \phi) = 1$을 만족하는 $1 < e < \phi$)

. 확장 유클리드 알고리즘을 사용하여 특별한 정수 d를 계산한다.

($ed \equiv 1 (\bmod \phi)$을 만족하는 $1 < d < \phi$)

. 공개키 (n, e), 비밀키 (d)

2.2. 확장 유클리드 알고리즘 (extended Euclidean algorithm)

- INPUT : $a \geq b$ 인 두개의 음이 아닌 정수

- OUTPUT : $d = \gcd(a, b)$와 $ax + by = d$를 만족하는 정수 x, y

- RSA 키 생성 알고리즘에서 생성된 정수 e와 d를 암호화 지수 (exponent) 또는 복호화 지수라 부르고, n을 모듈러스 (modulus)라 한다.

2.3. RSA 공개키 암호화와 복호화

B가 A에게 보낼 메시지 m을 암호화하고, A는 복호화 한다.

* 암호화 (encryption) : B는 다음 과정을 수행한다.

- A의 신뢰할 수 있는 공개키 (n, e)를 가져온다.

- $[0, n-1]$ 의 간격으로 메시지를 정수 m으로 나타낸다.

- $c = m^e \bmod n$ 을 계산한다.

- 암호문 c를 A에게 보낸다.

* 복호화 (decryption) : A는 다음 과정을 수행한다.

- 암호문 c에서 평문 m을 얻기 위해 비밀키 (private key) d를 사용한다.

- $m = c^d \bmod n$을 계산한다.

2.4. RSA 복호화 과정

$ed \equiv 1 (\bmod \phi)$ 이기 때문에, $ed = 1 + k\phi$를 만족하는 정수 k가 존재한다.

페르마의 정리 (Fermat's theorem)에 의해 만약 $\gcd(m, p) = 1$ 이면 $m^{p-1} \equiv 1 (\bmod p)$ 이다.

이식의 양변에 $k(q-1)$ 제곱을 하고 , m을 곱하면 식은 다음 식 (2)과 같다.

$$m^{1 + k(p-1)(q-1)} \equiv m (\bmod p) \qquad (2)$$

만약 $\gcd(m, p) = p$이면 $\bmod p$ 의 0 양변이 합동이기 때문에 위의 합동은 유효하다. 그

러므로 모든 경우에 대해 $m^{ed} \equiv m(\bmod p)$ 이다.

같은 방식으로, $m^{ed} \equiv m(\bmod q)$일 때, p, q가 다른 소수이기 때문에 $m^{ed} \equiv m(\bmod n)$ 이고 $c^d \equiv (m^e)^d \equiv m(\bmod n)$ 이다.

III. 몽고메리 곱셈

r이 2의 제곱수인 경우, n과 서로 소가 되기 위한 조건은 n이 홀수이면 된다. 몽고메리 곱셈을 설명하기 위해 다음 집합을 정의하면,

$$\{\bar{a} = ar \bmod n \mid 0 \le a \le n-1\} \quad (3)$$

식 (3)과 같이 나타낼 수 있고 이는 완전한 잉여 집합니다. 다시말해 이 집합은 0에서 n-1까지의 모든 정수를 포함하고 있으며, 따라서 식 (3)의 집합의 모든 원소는 0에서 n-1까지의 정수와 일대일 대응이 된다는는 의미를 가진다. 몽고메리 알고리즘은 이런 특성을 이용하여 고속 곱셈을 수행하는데, 두 수의 곱의 n에 대한 나머지를 계산함으로써 이를 가능하게 한다. 두 n에 대한 나머지인 $\bar{a}, \bar{b}$가 주어졌을 때 다음과 같이 몽고메리 곱을 정의할 수 있다 [3].

$$\bar{c} = \bar{a}\bar{b}r^{-1} \bmod n \quad (4)$$

여기서 r^{-1}은 $r^{-1}r = 1 \bmod n$ 인 관계가 성립하는 모듈라 역수를 의미한다. (4)의 결과인 c는 $c = ab(\bmod n)$인 관계가 성립하는데, 그 이유는 아래와 같다.

$$\begin{aligned} \bar{c} &= \bar{a}\bar{b}r^{-1} \ (\bmod n) \\ &= arbrr^{-1} (\bmod n) \\ &= cr \ (\bmod n) \end{aligned} \quad (5)$$

몽고메리 알고리즘을 기술하기 위해서는 추가적인 n'의 정의가 필요한데, $rr^{-1} - nn' = 1$의 특성을 가지며, r^{-1}과 n'은 확장 유클리드 알고리즘으로 계산이 가능하다. 몽고메리 곱셈 알고리즘은 다음 식 (6)과 같이 표현된다.

function MonPro($\bar{a}, \bar{b}$)
Step 1. $t := \bar{a}\bar{b}$
Step 2. $u := [t + (tn' \bmod r)n]/r$
Step 3. if $u \ge n$ then return $u - n$, else return u (6)

여기서 발생하는 문제점은, 일반적인 나머지를 n에 대한 나머지로 변환하고, 또 계산된 n에 대한 나머지를 시초의 일반적인 나머지로 변환하는 것이 오버헤드로 작용한다는 점이다. 따라서, 몽고메리 감축 연산은 단일 모듈라 연산에 사용되는 것보다는 아래와 같은 모듈라 제곱 연산에 사용되는 것이 매우 효율적이다.

function ModExp(a,e,n)
Step 1. $\bar{a} := ar \bmod n$
Step 2. $\bar{x} := 1r \bmod n$
Step 3. for $i = j-1$ downto 0
$\bar{x} := MonPro(\bar{x}, \bar{x})$
if $e_1 = 1$ then $\bar{x} :=$ Monpro($\bar{x}, \bar{a}$)
Step 4. return $x :=$ MonPro($\bar{x}$,1) (7)

IV. 오퍼랜드 스캐닝

큰 수에 대한 연산은 그 수를 워드 단위로 쪼개어 수행하는 것이 컴퓨터 수학에서는 일반적이다. 이 컴퓨터의 워드 크기를 w라고 한다면 그 수는 $W = 2^w$로 표현되는 수들의 연속으로 생각할 수 있다. Koc은 이와 같은 방식으로 몽고메리 알고리즘을 수행하는 데 필요한 시간과 공간을 최적화 하는 몇가지 방식을 제안하였다.

1. 분리형 오퍼랜드 스캐닝

```
ⓐ Product
for i=0 to s-1
  C := 0
  for j=0 to s-1
    (C, S) := t[i+j] + a[j] * b[i] + C
  t[i+j] := S
t[i+s] := C

ⓑ Reduction
for i=0 to s-1
  C := 0
  m := t[i] * n'[0] mod W
  for j=0 to s-1
```

```
    (C, S) := t[i+j] + m * n[i] + C
    t[i+j] := S
  t[s+1] := t[s+1] + C

ⓒ Division
for j=0 to s-1
  u[j] := t[j+s]

ⓓ Compensation
B := 0
for i=0 to s-1
  (B, D) := u[i] - n[i] - B
  t[i] := D
(B, D) := u[s] - B
t[s] := D
if B=0 then return t[0], ..., t[s-1]
       else return u[0], ..., u[s-1]
```

2. 대단위 집적 오퍼랜드 스캐닝

```
ⓐ Product & Reduction & Division
for i=0 to s-1
  C := 0
  for j=0 to s-1
    (C, S) := t[j] + a[j] * b[i] + C
    t[j] := S
  (C, S) := t[s] + C
  t[s] := S
  t[s+1] := C
  m := t[0] * n'[0] mod W
  (C, S) := t[0] + m * n[0]
  for j=1 to s-1
    (C, S) := t[j] + m * n[j] + C
    t[j-1] := S
  (C, S) := t[s] + C
  t[s-1] := S
  t[s] := t[s+1] + C

ⓑ Compensation
B := 0
for i=0 to s-1
  (B, D) := t[i] - n[i] - B
  u[i] := D
(B, D) := t[s] - B
t[s] := D
if B=0 then return u[0], ..., u[s-1]
       else return t[0], ..., t[s-1]
```

3. 미세 오퍼랜드 스캐닝

```
ⓐ Product & Reduction & Division
for i=0 to s-1
  (C, S) := t[0] + a[0] * b[i]
  t[1] := t[1] + S
  m := S * n'[0] mod W
  (C, S) := S + m * n[0]
  for j=1 to s-1
    (C, S) := t[j] + a[j] * b[i] + C
    t[j+1] := t[j+1] + C
    (C, S) := S + m*n[j]
    t[j-1] := S
  (C, S) := t[s] + C
  t[s-1] := S
  t[s] := t[s+1] + C
  t[s+1] := 0

ⓑ Compensation
B := 0
for i=0 to s-1
  (B, D) := u[i] - n[i] - B
  t[i] := D
(B, D) := u[s] - B
t[s] := D
if B=0 then return t[0], ..., t[s-1]
       else return u[0], ..., u[s-1]
```

V. 결 론

RSA 곱셈 연산을 수행하기 위한 몽고메리 연산의 성능을 높이기 위해서 오퍼랜드 스캐닝 방법에 대한 알고리즘을 조사하였다. 알고리즘을 분석해 보면, 분리형 오퍼랜드 스캐닝 방법은 덧셈 부분에서 이득을 취할 수 있으나 메모리를 사용해야 하는 단점이 있어 공간 면에서 비효율적이다. 집적 오퍼랜드 스캐닝 방식들이 공간 면에서 효율성을 보이는데, 대단위 집적 방식과 미세 집적 방식이 덧셈 연산의 수, 읽기 회수, 쓰기 회수 등에서 비슷한 효율성을 보였으나, 미세 집적 방식이 하드웨어로 구현이 적절하며, 알고리즘을 다시 한번 최적화 시킬 수 있는 여지가 있다. 이는 실제 하드웨어로 구현 시 연산 자체를 축약시킬 수 있다는 것을 의미하며, 차후 미세 접 오퍼랜드 스캐닝 방식을 수정하여 구현한다면 연산 횟수와 메모리 제어에서 상당한 이득을 볼 수 있을 것이라 기대된다.

참고문헌

[1] 김철, 암호학의 이해, 영풍문고, 1996.

[2] R. L. Rivest, A. Shamir, and L. M. Adleman, "A Method for Obtaining Digital Signatures and Public-key Cryptosystems," Communications of the ACM, Vol. 21, pp. 120-126, Feb. 1978.

[3] Cetin Kaya Koc, et. al., "Analyzing and Comparing Montgomery Multiplication Algorithms," IEEE Micro, June 1996.