# 8 비트 센서 노드 상에서 효율적인 공개키 암호를 위한 다정도 제곱 연산의 최적화

## (Optimizing Multiprecision Squaring for Efficient Public Key Cryptography on 8-bit Sensor Nodes)

김 일 희 † 박 용 수 †† 이 윤 호 †††
(Ilhee Kim) (Yongsu Park) (Younho Lee)

**요 약** Multiprecision Squaring은 공개키 알고리즘을 구성하는 연산 중에서 가장 중요한 연산 중 하나이다. 본 논문에서는 기존의 Multiprecision Squaring 알고리즘을 개선하여 연산 양을 줄임으로 성능을 향상시키는 Squaring 기법들을 제시하고 구현하였다.

Scott이[1]에서 제안한 Carry-Catcher Hybrid 곱셈 알고리즘은 Gura가 제안한 Hybrid 곱셈 알고리즘[2]을 계승 발전시킨 것으로 MIRACL 라이브러리에 구현되어 있으며, Carry-Catcher Hybrid 방법 사용한 Multiprecision Squaring 알고리즘도 MIRACL에 함께 구현되어 있다. 본 논문에서 이 Carry-Catcher Hybrid Squaring 알고리즘을 발전시켜 보다 효율적인 Squaring 알고리즘인 Lazy Doubling Squaring 알고리즘을 제안하고 구현하였으며, atmega128상에서 성능테스터를 수행하여 Carry-Catcher Hybrid Squaring 알고리즘과 비교하여 더 효율적인 알고리즘임을 보였다.

표준 Squaring 알고리즘이 $S_{ij} = x_i * x_j = S_{ji}$ 인 사실을 기반으로 곱셈의 횟수를 절반 가까이 줄인 알고리즘이라면 본 논문에서 제시한 Lazy Doubling Squaring 알고리즘은 $a_0 * 2 + a_1 * 2 + ... + a_{n-1} * 2 + a_n * 2 = (a_0 + a_1 + ... + a_{n-1} + a_n) * 2$ 라는 사실을 기반으로 하여 doubling 연산 횟수를 획기적으로 줄인 알고리즘으로, MIRACL에 구현되어 있는 Multiprecision Squaring 알고리즘 보다 atmega128상에서 약 25% 정도의 빠른 결과를 얻을 수 있었으며, 저자가 아는 바로는 현재까지 나온 어떤 방법보다 빠르다.

**키워드** : 보안, 공개키, 암호, 센서 노드, 제곱 연산

***Abstract*** Multiprecision squaring is one of the most significant algorithms in the core public key cryptography operation. The aim of this work is to present a new improved squaring algorithm compared with the MIRACL's multiprecision squaring algorithm in which the previous work [1] on multiprecision multiplication is implemented.

First, previous works on multiprecision multiplication and standard squaring are analyzed. Then, our new Lazy Doubling squaring algorithm is introduced.

In MIRACLE library [3], Scott's Carry-Catcher Hybrid multiplication technique [1] is applied to implementation of multiprecision multiplication and squaring. Experimental results of the Carry-Catcher hybrid squaring algorithm and the proposed Lazy Doubling squaring algorithm both of which are tested on Atmega128 CPU show that proposed idea has achieved significant performance improvements.

· 본 연구는 두뇌한국 21 연구 결과로 수행되었음

† 비 회 원 : 한양대학교 전자컴퓨터통신공학과
iamramj@gmail.com
†† 종신회원 : 한양대학교 정보통신대학 교수
yongsu@hanyang.ac.kr
††† 종신회원 : 영남대학교 정보통신공학과 교수
yhlee@ynu.ac.kr
(Corresponding author임)

논문접수 : 2009년 2월 6일
심사완료 : 2009년 8월 24일


정보과학회논문지: 시스템 및 이론 제36권 제5호(2009.10)

The proposed Lazy Doubling Squaring algorithm reduces addition instructions by the fact $a_0 * 2 + a_1 * 2 + ... + a_{n-1} * 2 + a_n * 2 = (a_0 + a_1 + ... + a_{n-1} + a_n) * 2$ while the standard squaring algorithm reduces multiplication instructions by the fact $S_{ij} = x_i * x_j = S_{ji}$. Experimental results show that the proposed squaring method is 25% faster than that in MIRACL.

**Key words** : security, public key, cryptography, sensor node, squaring

## 1. 서 론

공개키 암호화 알고리즘은 기존 비밀키 알고리즘의 키 전달 필요성을 없애줌으로 안전한 통신을 할 수 있도록 하는 기술이기 때문에 비밀통신을 해야 하는 수많은 환경에서 그 필요가 절실하다. 하지만 일반적으로 공개키 암호화 알고리즘들은 많은 CPU 연산을 수반하기 때문에 센서노드나 스마트카드와 같은 저성능의 CPU를 이용하는 기기에서는 실용적이지 못하다는 것이 현실이다. 따라서 기존의 공개키 알고리즘의 성능을 향상 시키거나 더 효율적인 알고리즘을 찾는 것은 암호학에서 대단히 중요한 연구 주제이다.

공개키 알고리즘에는 Diffie-Hellman의 키교환 알고리즘, RSA 알고리즘, Rabin 알고리즘, ElGamal 알고리즘, ECC 알고리즘 등 여러 가지가 있지만 이들 알고리즘 대부분에서 사용되고 있은 연산이 Multiprecision 곱셈과 Squaring연산이다. 이 두 연산은 큰 정수를 곱하거나 제곱을 하기 위한 연산으로 공개키 암호화 프로그램[4,5]에서 사용되는 알고리즘의 핵심 연산이기 때문에 이들 연산 알고리즘의 효율성은 그 영향력이 매우 크다.

이러한 이유로 공개키 알고리즘과 같은 많은 exponentiation 알고리즘 연구에서 곱셈연산의 횟수를 줄이는데 중점을 두고 있으며, 일반적인 Multiprecision 곱셈에 대한 연구는[6,7] 오래 전부터 수행되어 왔고, 최근에는 스마트카드나 센서노드용 CPU와 같은 저성능의 CPU에서의 곱셈연구도 활발하게 진행되고 있다[1,2,8,9].

하지만 Multiprecision 곱셈이 빨라지면 Squaring은 자연히 빨라지기 때문에 Squaring에 특화된 연구는 Multiprecision 곱셈만큼 활발히 행해지고 있지 않다. Multiprecision Squaring은 RSA의 핵심 연산인 modular exponentiation과 같은 연산에서 약 35%의 비중을 차지하고 있을 뿐만 아니라 ECC에서 사용되는 point addition이나 point doubling과 같은 중요한 연산에서 주요한 병목[10]이 되고 있기 때문에 이 알고리즘의 성능 향상은 그 중요도가 매우 크다고 볼 수 있다.

표준 Squaring 알고리즘이 $S_{ij} = x_i * x_j = S_{ji}$ 인 사실을 기반으로 곱셈의 횟수를 절반 가까이 줄인 알고리즘이라면 본 논문에서 제시한 Lazy Doubling Squaring 알고리즘은 $a_0 * 2 + a_1 * 2 + ... + a_{n-1} * 2 + a_n * 2 = (a_0 + a_1 + ... + a_{n-1} + a_n) * 2$ 라는 사실을 기반으로 하여 doubling 연산 횟수를 획기적으로 줄인 알고리즘이다. 우리는 MIRACL에 구현되어 있는 Multiprecision Squaring을 고쳐서 원 알고리즘 보다 atmega128상에서 약 25% 정도의 빠른 결과를 얻을 수 있었으며, 저자가 아는 바로는 현재까지 나온 어떤 방법보다 빠르다. 더 나아가, ADCC라는 새로운 CPU연산자를 제안하고, 이 연산자가 추가된다면 제안한 Squaring 알고리즘의 속도를 12.7% 더 향상시킬 수 있으며, Scott의 Carry-Catcher Hybrid 곱셈 알고리즘에 적용해도 15%정도의 속도가 향상 될 수 있음을 보였다.

본 논문의 구성은 2장에서 기존에 연구되어 활용되고 있는 Multiprecision 곱셈 및 표준 Squaring 알고리즘을 소개하고, 3장에서는 Carry-Catcher Hybrid Squaring 의 성능을 향상시킨 Lazy Doubling Squaring 기법을 소개와 향상된 성능측정 결과를 Carry-Catcher Hybrid Squaring 알고리즘과 비교하여 제시하였다. 마지막으로 4장에서는 결론과 향후 연구방향을 기술한다.

## 2. 기존 연구

### 2.1 Multiprecision 곱셈 기법

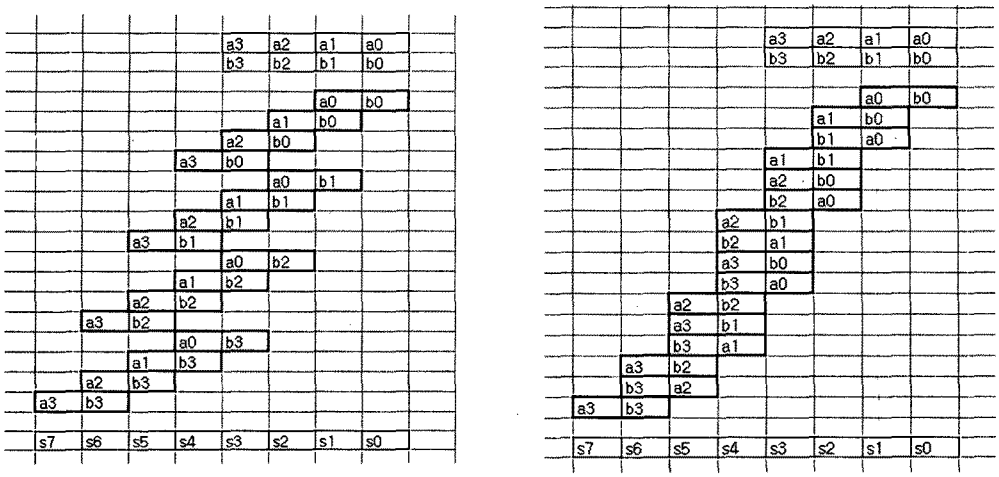
RSA 암호화 과정 중에 소요되는 시간은 대부분은 Modular Exponentiation 이 차지하고 있으며, Modular Exponentiation 과정 중에 시간을 소비하는 대표적인 연산은 Multiprecision 곱셈, squaring, modular reduction 이다. 각각의 수행시간 비율을 보면 곱셈 25%, squaring 35%, modular reduction 40% 정도를 차지한다[11]. 이 중 Multiprecision 곱셈, squaring 기법들을 소개한다.

2.1.1 School book 곱셈 알고리즘

School book 알고리즘은 어릴 적 학교에서 배웠던 곱셈방법으로 Row-Wise 곱셈이라고도 부르는 가장 기본적인 곱셈 알고리즘이다. 그림 1(a)와 같이 승수의 한 자리 수(b0)와 피승수(a0~a3)의 모든 자리 수를 곱하고 상위자리(b1)로 올라가서 이전과 같은 방식으로 반복하며 곱하여 그 결과를 더해주는 방식이다.

School Book 곱셈 방법은 각 수를 곱한 결과를 더해줄 때, 생겨나는 carry를 연속적으로 상위 자리 수에 더해줘야 하는 carry propagation이 많이 발생한다는 단점이 있지만 레지스터가 많을 경우 메모리 접근을 줄일 수 있는 장점이 있다.

2.1.2 Comba 곱셈 알고리즘




(a) School Book Multiprecision 곱셈 (b) Comba의 Multiprecision 곱셈

그림 1 School Book & Comba Multiprecision 곱셈

Column-Wise 곱셈이라고도 부르는 Comba 기법도, School book에서 처럼 $O(n^2)$ 만큼의 곱셈을 하는 것은 변함이 없다. 다만 같은 자리수의 결과를 만들어 내는 곱셈을 모아서 함으로, 반복해서 Carry를 상위 자리로 더해주는 처리를 줄 일 수 있다.

그림 1(b)와 같이 승수와 피승수의 인덱스 합이 0인 곱셈조(a0 * b0)를 모두 곱해주고 그 다음에 인덱스의 합이 1 인 곱셈조(a1 * b0, a0 * b1)를 곱해주고 계속해서 인덱스의 합이 2, 3, ..., 6 인 곱셈조를 그림 1(b)와 같이 곱해나간다. 각 곱셈조의 곱셈이 끝날 때마다 전체 곱셈의 결과가 한자리씩 확정되므로 확정된 값은 메모리에 저장할 수 있다. Comba방법의 장점은 carry 처리를 줄일 수 있다는 것과 레지스터를 적게 사용한다는 것이다.

2.1.3 Hybrid 곱셈 알고리즘

Hybrid 곱셈[2] 알고리즘은 레지스터 수와 메모리 접근 수를 최적화하는 것을 목표로 지금까지 살펴본 두 방법의 장점만을 취해 만들어진 더 빠른 Multiprecision 곱셈 알고리즘이다.

이 알고리즘은 곱할 수들을 일정한 단위로 묶어서 각각의 묶음(그림 2의 A0~A3, B0~B3)끼리는 Colum-Wise 곱셈을 수행하고, 묶음내의 각 수들간의 곱셈은 Row-Wise 곱셈을 수행한다(이하 이 묶음 단위를 Hybrid 단위 혹은 $d$라고 한다.)

즉, Hybrid 단위 내의 각 수를 곱해주는 내부 알고리즘으로는 가용한 레지스터를 가능한 많이 이용하도록 Row-Wise 곱셈 알고리즘을 사용하고, Hybrid 단위끼리 곱해주는 외부 알고리즘으로는 carry 처리의 효율을 향상 시키기 위해 Colum-Wise 곱셈을 사용한다.

이를 통해 가능한 많은 레지스터를 활용해 반복해서 읽어들이는 메모리 로드수를 최소화 하면서, 연속적으로 수행해야 하는 carry 처리를 줄일 수 있다. [2]에 의하면 레지스터가 32개인 atmega128의 경우 $d=6$이 최적이라고 밝히고 있다.

2.1.4 Carry-Catcher Hybrid 곱셈 알고리즘

Uhsadel과 Scott은 [3]과 [1]에서 각각 Hybrid 알고리즘을 개선하여 성능을 향상 시켰다. [1]의 방법이 [3]의 방법보다 더 단순하면서 큰 성능향상을 이루었다. Scott의 Carry-Catcher Hybrid 알고리즘의 주된 성능 향상 포인트는 carry 처리의 효율을 높이는 것이었다. Gura의 초기 Hybrid 곱셈에서는 덧셈을 할 때마다 Hybrid 단위간의 곱셈 중간 결과값들(s0~s7)에까지 carry를 올려주며 처리를 해야 한다.

그림 3(a)와 같이 b0와 s0를 더해줄 때, 발생할 지도 모르는 carry를 상위 자리수에 더해줘야 하고, 상위자리에서는 그 carry의 덧셈으로 인해 발생할 수 있는 또 다른 carry를 반복해서 더해주는 carry propagation 처리를 해야 한다. 그로 인해 Hybrid 곱셈처리에서 carry 처리는 가장 주된 오버헤드가 되고 있다. [3]에 의하면 이런 carry 처리로 인한 오버헤드가 적어도 66.66%는 발생한다고 분석하고 있다.

Scott은 Hybrid 곱셈 알고리즘을 사용하지만 Carry-Catcher라는 것을 이용해 carry propagation 문제를 해결하고 있다. Carry-Catcher는 레지스터의 일부를 carry 만을 누적하는 용도로 사용하는 것이다. 그런데 Carry-Catcher로 사용되는 레지스터의 손실 분 만큼 $d$를 축소해야 한다. 즉, [2]에서는 $d=6$으로 사용하였으나 [1]에서는 $d=4$로 사용하고 있다. 그러나 carry pro-

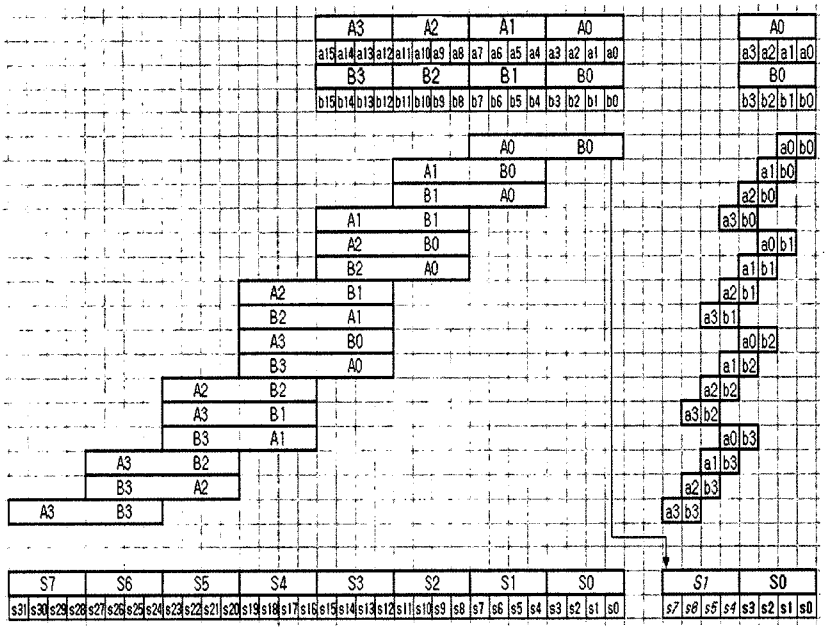


그림 2 Gura의 Original Hybrid 곱셈

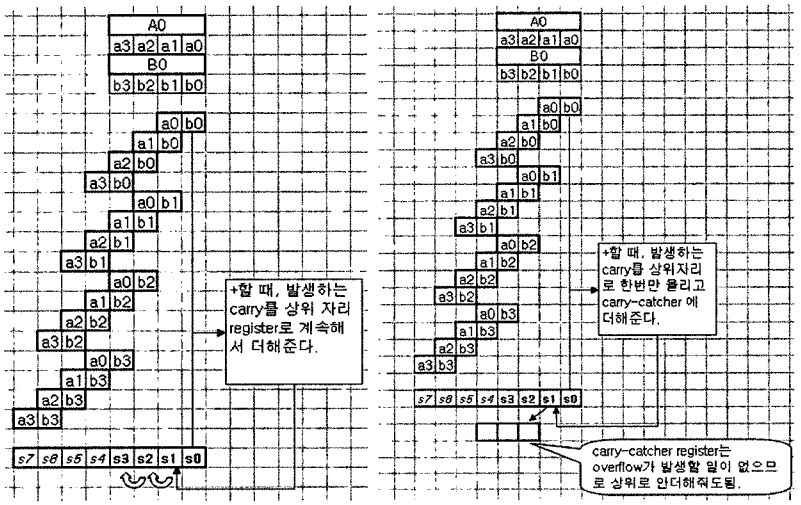



그림 3 (a) Gura의 일반 Hybrid 곱셈에서의 carry 처리와 (b) Scott의 Carry-Catcher Hybrid 곱셈에서의 carry 처리

pagation의 오버헤드가 워낙 크기 때문에 $d$가 작아짐으로 인한 손실을 상쇄하고도 성능 향상을 가져올 수 있다.

한번의 덧셈으로 인해 반복적으로 수반되는 carry처리를 Carry-Catcher Hybrid 알고리즘에서는 Overflow가 발생하지 않는 Carry-Catcher 레지스터에 누적해 두고 각 수들의 곱셈계산의 중간결과를 메모리에 저장하기 시점에서 Carry-Catcher에 저장된 carry들의 누적 값을 중간 결과값(s0~s7)과 더해주게 된다. 실제 필요한 레지스터 수는 중간 결과값을 저장하기 위해 2d개가 필요하고 Carry-Catcher를 위해 $2d-1$개가 필요하다. 이렇게 함으로써 중간 결과값을 구하기 위해 수행하는 덧셈으로 인해 생기는 연속적인 carry 처리를 대폭 줄일 수 있게 된다. 중간 결과값은 Hybrid단위를 한 번씩만 곱해서 구해지는 것이 아니라, N개의 Hybrid단위끼리 곱할 경우 최대 2N - 1번의 Hybrid단위 곱셈을 수행하여 구하게 된다. 예를 들어, 그림 2에서와 같이 하이브리드 단위 A0~A3 와 B0~B3 를 곱할 경우, S3값을 구하기 위해 7번(= 2*4-1)의 하이브리드 단위 곱셈이 필요하다.

위에서 제시한 곱셈 방법 이외에도 Karatsuba Ofman 곱셈법[6]은 오래된 알고리즘이지만 일반적인 곱셈 알고리즘으로는 처음으로 $O(n^2)$ 연산 복잡도를 깬 다항식 기반의 알고리즘으로 $O(n^{\log(3)})$ 연산 복잡도를 갖는다. 하지만 다항식 기반의 문제를 해결하기 위해 재귀호출 등 오버헤드가 많아 일정 수 이상의 큰 수가 아니면 일반 곱셈법보다 효율성이 떨어지며, Montgomery reduction에서는 사용할 수 없다[2]는 단점이 있다.

### 2.2 Multiprecision squaring 기법

Multiprecision squaring의 경우, Mutiprecision 곱셈의 특수한 경우라고 생각할 수 있다. 그러나 승수와 피승수가 같은 점을 이용하여 기존의 곱셈 방법들보다 효율적인 방법들이 제안되었다.

2.2.1 표준 Multiprecision Squaring 알고리즘

어떤 큰 정수 A가 있을 때 A를 k-bit 워드의 승수로 나누어 $a_{n-1}$, ..., $a_0$ 와 같이 표기 한다면, A를 Squaring 할 경우, $ai * aj$ $(i \neq j)$의 결과가 같은 값이 항상 2번씩 등장한다는 사실을 기초로 일반 Multiplication에 비해 연산 수를 줄이는 Squaring방법을 표준 Squaring 알고리즘이라 한다.

즉 Squaring의 경우 승수와 피승수가 같기 때문에 그림 4(a)와 같이 $ai * aj$ $(i \neq j)$의 경우는 곱의 결과가 같은 곱셈 쌍이 반드시 존재한다. 따라서 이러한 곱셈의 경우 두 번 곱셈을 수행하지 않고 한 번 곱한 결과를 shift연산이나 덧셈 연산으로 그림 4(b)와 같이 doubling하여 계산할 수 있다. (이하 이와 같이 두 번씩 등장하는 곱셈 곱셈을 doubling 곱셈이라 한다.) 이와 같은 특성 때문에 Multiprecision Squaring의 경우 Multiprecision 곱셈보다 절반 가까운 곱셈 수를 줄일 수 있다. 정확히 절반이 되지 않는 이유는 한 번씩만 등장하는 $ai * aj$ $(i = j)$와 같은 곱셈이 있기 때문이다. (이하 이런 곱셈을 'self 곱셈'이라 한다.)

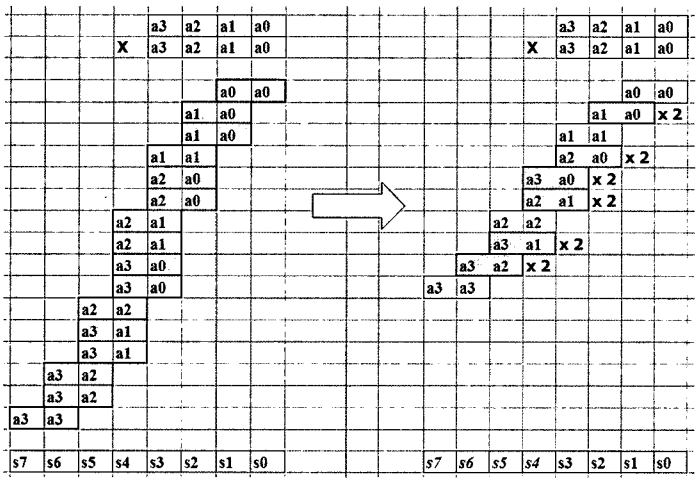
2.2.2 Carry-Catcher Hybrid Squaring

그림 5는 본 연구의 비교 대상이 되는 Carry-Catcher Hybrid Squaring 방법을 보여주고 있다. 본방법도 Hybird 기법을 이용한다. Hybrid 단위 내부의 곱셈을 도식한 그림 5의 좌측 그림을 보면 a0 * a4는 a4 * a0와 같으므로 각각을 두 번씩 곱하는 것이 아니라 한 번만 곱하고 그 곱한 값을 doubling하는 것을 통해 곱셈 횟수를 줄이고 있는 것을 확인할 수 있다. 곱셈의 횟수가 정확히 반이 되지 않는 이유는 우측 그림의 self 곱셈 Hybrid단위(A0 * A0) 내에 존재하는 self 곱셈(a0 * a0, a1 * a1,..)은 doubling하지 않기 때문이다. 본 방법도 Carry-Catcher Hybrid Squaring 방법과 같이 atmega128 환경에서 $d$=4일 경우 최적의 성능을 보이며 그러한 방식으로 Miracl 라이브러리에 구현되어 있다[3].

## 3. 새로운 Squaring 알고리즘 제안

### 3.1 Lazy Doubling Squaring

우선 제안 방법의 개념적인 내용을 살펴본다. 표준 Squaring 에서는 그림 6(a) (그림 5의 우측 Hybrid 단위 self-곱셈 부분과 동일)에서처럼 doubling 곱셈을 할 때마 곱셈 결과를 doubling하여 최종 결과에 더해주고 있다. 하지만 그림 6(b)처럼 doubling해야 할 곱셈결과를 doubling하기 전에 모두 더해주고, 더해진 결과(m1~m6)를 doubling 해주면 doubling을 하기 위해 수행하



(a) 일반 Comba 곱셈 (b) Comba곱셈을 이용한 표준 Squaring 기법

그림 4 일반 Comba곱셈 기법과 Comba곱셈을 이용한 표준 Squaring 기법

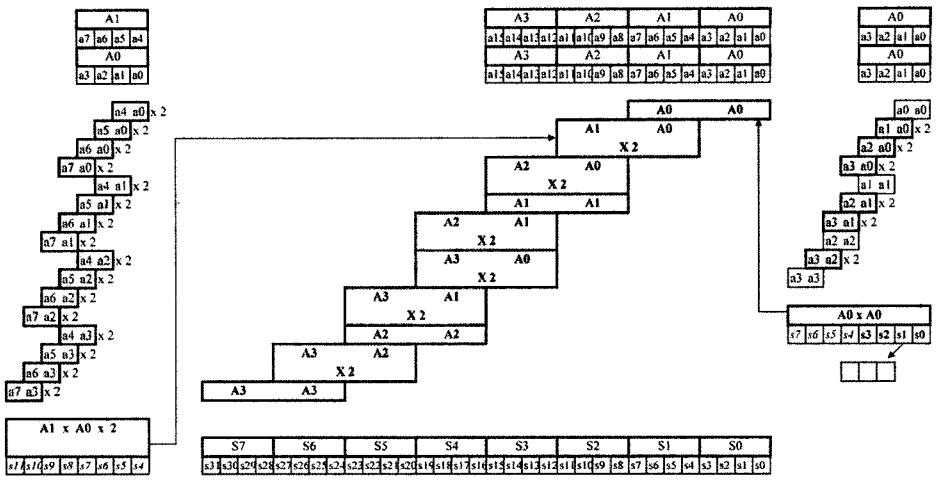
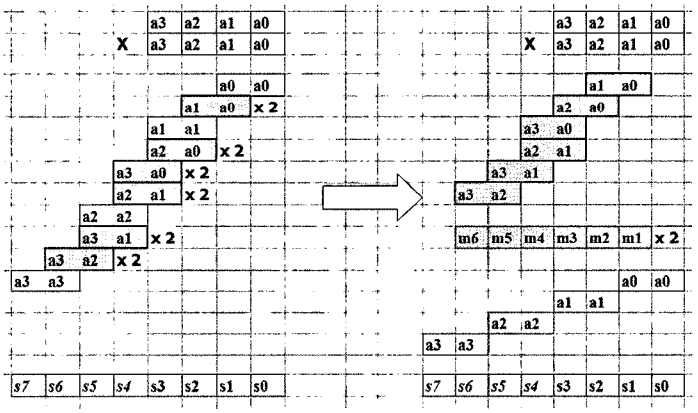


그림 5 MIRACL에 구현되어 있는 Carry-Catcher Hybrid Squaring



(a) Comba곱셈을 이용한 표준 Squaring 기법 (b) Lazy Doubling Squaring 기법

그림 6 표준 Squaring 기법과 본 고에서 제안하는 Lazy Doubling Squaring기법 (개념도)

는 연산수가 대폭 줄어 들게 된다.

그러나 그림 6의 비교의 경우 추가적으로 사용되는 m1-m6 레지스터 사용으로 인한 오버헤드가 나타나는 것이 단점으로 발생하게 된다.

이제 제안 방법을 본격적으로 살펴보기로 한다. 위의 개념적인 내용을 바탕으로 제안 방법을 설계하였다. 제안 방법의 첫번째 핵심은 아래와 같이 기존의 Carry-Catcher Hybrid Squaring 방법에 위의 Lazy Doubling Squaring 방법을 적용시킬 경우 추가적으로 요구되는 레지스터가 발생하지 않게 되는 것이다. 제안 방법과 Carry-Catcher Hybrid Squaring 방법과 비교해 보면, 그림 7의 오른쪽 그림과 같이 레지스터 m0-m7이 그림 5의 오른쪽 그림의 s0-s7과 동일한 역할을 수행하게 된다. 따라서 추가적인 레지스터 요구는 발생하지 않는다.

이제 제안 방법을 그림 7을 통하여 자세히 살펴보도록 한다. 그림 7에서와 같이 doubling 곱셈 결과(색칠된 부분)을 먼저 계산하고 메모리에 저장하기 직전에 누적된 중간 값(M0~M7)을 lazy doubling해서 메모리에 저장해 놓는다. 그리고 self 곱셈(a0*a0, a1*a1..)을 계산할 때, 저장해 두었던 중간 결과값과 곱셈을 위한 승수들을 메모리로부터 로드하여, 곱셈을 계산하고 중간 결과값에 더해준다. 서로 다른 Hybrid 단위를 곱할 때는 그림 7의 왼쪽에 있는 곱셈과정처럼 모든 요소의 곱을 계산해 주지만, A0*A0 나 A1*A1과 같은 self 곱셈 Hybrid 단위를 곱할 때는 그림 7의 오른쪽 곱셈과정처럼 self 곱셈은 수행하지 않는다.

Hybrid 단위간의 곱셈 과정을 하나의 세션으로 볼 때, 중간 결과 값(M0~M7)이 확정되는 시점은 각 중간

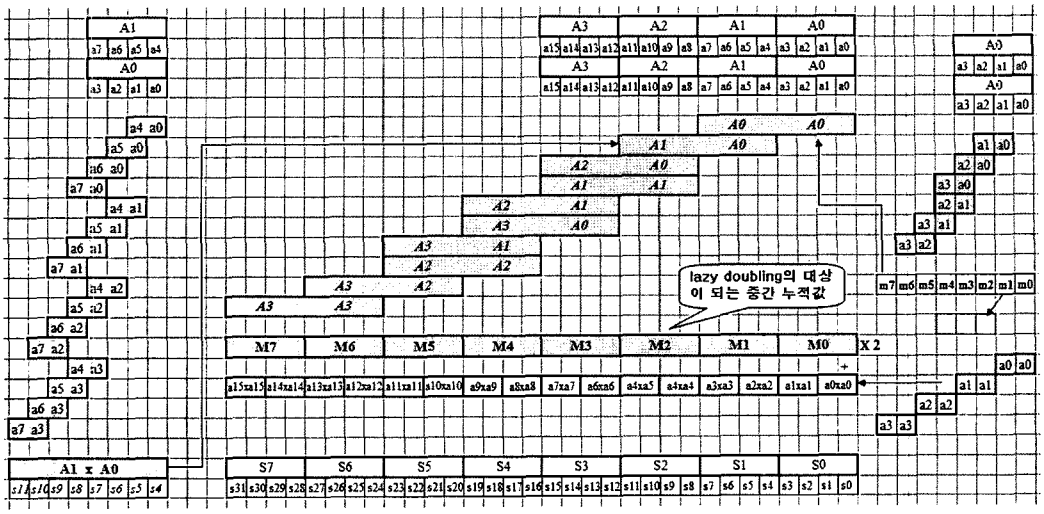



그림 7 본 고에서 제안된 새로운 Lazy Doubling Squaring

결과값들에 따라 다르다. M0와 같은 경우는 A0*A0 만 끝나면 중간 결과 값이 확정되지만, M2의 경우에는 A1*A0, A2*A0, A1*A1이 모두 끝나야 비로소 M2값을 확정할 수 있다. 이와 같이 하나의 중간 결과 값을 확정하기 위해 여러 번(최대 2N-1번, N=Hybrid단위의 개수)의 세션이 필요하기 때문에 그에 따른 carry 처리에 특별한 주의가 필요하다. 그림 7의 A0*A0가 끝났을 때 생성되는 8 byte의 곱셈 결과 중 절반인 하위 4 byte만 M0로 확정된다. 그러므로 확정된 M0는 doubling 하여 메모리에 저장할 수 있지만, M1의 일부가 되는 상위 4 byte는 그 다음 세션에서 계산되는 A1*A0가 끝나는 시점에서 doubling할 수 있다. 이러한 이유로 M0를 doubling할 때 발생하는 carry는 상위 자리에 바로 더할 수가 없다. 바로 더해주면 이 때 더한 carry도 M1을 doubling 할 때 함께 doubling 되기 때문에 계산에 오류가 생긴다. 따라서 상위 자리수로 올라가야 하는 carry는 상위 세션에서 doubling을 한 후에 더해줘야 정확한 계산을 할 수가 있다.

### 3.2 기존 연구와의 연산 수 비교

3.2.1 감소되는 덧셈 수의 비교

위의 그림 5, 7과 같이 N개의 하이브리드 단위가 존재하고, 각 하이브리드 단위 $d$가 4일 경우에 doubling을 하기 위해 필요한 덧셈 연산 수를 비교해 보자.

그림 5의 Carry-Catcher Hybrid Squaring의 경우, 각 Hybrid 단위 별 곱셈 횟수를 비교하면, self 곱셈이 아닌 단위 Hybrid 곱셈의 경우 총 곱셈 횟수는 $d^2$이며 이들을 모두 doubling 하므로 $d^2$의 덧셈이 발생한다. N개의 Hybrid 단위의 제곱을 경우 위와 같은 경우가 N(N-1)/2 번 발생한다. self-곱셈인 단위 Hybrid 곱셈의 경우, 총 곱셈 횟수는 $(d^2-d)/2+d$ 이며 이 중, 내부의 byte 단위의 self 곱셈이 $d$번 존재하므로 doubling으로 인한 덧셈 횟수는 $(d^2-d)/2$ 이다. 전체 N개의 Hybrid 단위 제곱에서 위와 같은 self 곱셈의 경우는 N번 발생한다. 결과적으로 doubling 으로 인해 야기되는 총 덧셈 횟수는 $d^2 * N(N-1)/2 + (d^2-d)/2 * N = Nd(Nd-1)/2$ 번이 발생한다. 그렇지만 각 곱셈의 결과가 2바이트 단위이므로 최종적으로 $Nd(Nd-1)$ 번의 덧셈이 발생한다.

그림 7의 Lazy Doubling Squaring의 경우에는 각 Hybrid 단위 내부에서는 doubling을 위해 추가 되는 덧셈이 존재하지 않는다. 대신 byte 단위의 self-곱셈 부분을 제외한 모든 Hybrid 단위 덧셈을 더한 결과인 d*2N byte 부분에 대해서 doubling 덧셈이 행해진다. (그림 7에서 M0~M7 부분) 결과적으로 $2Nd$ 번의 덧셈이 발생하게 된다. 따라서 감소되는 덧셈 연산의 양은 $Nd(Nd-1) - 2Nd = (Nd)^2-3Nd$ 이며 atmega128 환경에서 최적의 $d$는 4이며 N은 1 이상이므로 항상 성능 향상이 있게 된다.

3.2.2 전체 연산량의 비교

표 1은 atmega128 상에서 160bit * 160bit의 Squaring을 할 때 각 알고리즘 별 연산수를 비교한 표이다. 제안 방법과 Carry-Catcher Hybrid Squaring 알고리즘과의 비교 결과를 나타낸다. 사용되는 cycle의 수에서 제안 방법이 기존 방법에 비해 약 20.19% 감소된 것을 알 수 있다.

Carry-Catcher Hybrid Squaring의 연산 수와 비교해 볼 때 가장 큰 차이가 나는 것은 덧셈 연산(add/adc)이다. 이것은 Lazy doubling 으로 인해 덧셈 연산

표 1 MIRACL의 Carry-Catcher Hybrid Squaring과 본 고의 Lazy Doubling Squaring의 연산 수 비교

| | | This Work (Lazy Doubling Squaring) | | Carry-Catcher Hybrid Squaring | |
|---|---|---|---|---|---|
| Instruction | CPI | Instructions | Cycles | Instructions | Cycles |
| add/adc | 1 | 804 | 804 | 1265 | 1265 |
| mul | 2 | 210 | 420 | 210 | 420 |
| ldd | 2 | 120 | 240 | 100 | 200 |
| st | 2 | 40 | 80 | 40 | 80 |
| mov/movw | 1 | 84 | 84 | 70 | 70 |
| other | | | 20 | | 30 |
| Totals | | | **1648** | | **2065** |

이 절약된 것임을 알 수 있다. 그러나 ldd instruction 횟수에서는 제안 방법이 기존의 Carry-Catcher Hybrid 방법보다 횟수가 던 많은 것을 알 수 있는데, 이는 그림 7의 오른쪽 그림에서 byte 단위의 self 곱셈 결과를 추후에 더해주기 때문에 발생한다. 그러나 이러한 오버헤드는 제안 방법이 감소시킨 덧셈 연산의 양에 비해 미미하므로 전체적으로 성능 향상이 있게 된다.

표 2는 다양의 비트 길이의 Squaring 연산을 수행할 경우의 제안 방법과 기존 방법의 비교이다. 전체적으로 계산하는 비트 길이가 길어질수록 제안 방법의 이득이 증가함을 알 수 있다.

표 2 MIRACL의 Carry-Catcher Hybrid Squaring과 본 고의 Lazy Doubling Squaring의 성능비교

| | 64bit | 128bit | 160bit | 256bit | 320bit |
|---|---|---|---|---|---|
| Carry-Catcher Hybrid Squaring | 397 | 1365 | 2065 | 5029 | 7725 |
| **This work** | **370** | **1126** | **1648** | **3790** | **5698** |
| saving cycle | 27 | 239 | 417 | 1239 | 2027 |
| saving percent | 6.80% | 17.51% | 20.19% | 24.64% | 26.24% |

3.2.3 사용되는 레지스터 수의 비교

아래의 표 3은 기존 방법과 제안 방법을 atmega128 상에서 구현시 사용되는 레지스터의 용도 및 전체 개수에 대해 나타내고 있다. 양 방법 모두 사용 가능한 레지스터의 한계 (32개) 때문에 $d$=4값이 사용 가능한 최대 값이 된다.

## 4. 결론 및 향후 연구 방향

본 논문에서는 센서노드나 스마트카드와 같은 저전력 CPU에서 효율적인 Carry-Catcher Hybrid 알고리즘을 개선하여 보다 효율적인 Squaring 알고리즘을 제안하였다. 제안하는 알고리즘은 Atmel AVR 플랫폼상에서 MIRACL에 구현되어 있는 Carry-Catcher Hybrid Squaring 알고리즘 보다 월등한 연산 효율성을 보인다. Squaring 연산은 RSA나 ECC와 같은 공개키 알고리즘의 핵심적인 연산이기 때문에 이 논문의 결과를 이용하여 이러한 공개키 알고리즘의 성능을 직접적으로 향상시킬 것으로 기대한다. 또한 제안된 알고리즘은 비교적 간단하기 때문에 여러 다른 플랫폼에도 어렵지 않게 적용 가능할 것으로 예상한다. 향후, 제안된 알고리즘을 ARM이나 x86 등 다양한 플랫폼에 적용하는 연구를 통해 본 알고리즘을 일반화하는 작업을 하고자 한다.

## 참 고 문 헌

[1] M. Scott and P. Szczechowiak, "Optimizing Multiprecision Multiplication for Public Key Cryptography," Cryptology ePrint Archive, Report 2007/299, 2007. http://eprint.iacr.org/.

[2] N. Gura, A. Patel, A. Wander, H. Eberle, and S.C. Shantz, "Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs. Proceedings of Workshop on Cryptographic Hardware and Embedded

표 3 atmega128 상에서 사용되는 레지스터의 비교 ($d$ = 4)

| | Lazy doubling squaring (제안 방법) | | Carry-Catcher hybrid squaring | |
|---|---|---|---|---|
| 용도 | 개수 | 할당된 레지스터 | 개수 | 할당 레지스터 |
| 곱셈결과를 저장할 레지스터 | 2 | r0, r1 | 2 | r0, r1 |
| 승수 레지스터 | 4 | r2~r5 | 4 | r2~r5 |
| 피승수 레지스터 | 1 | r6 | 1 | r6 |
| 상위 session에 넘겨줄 carry 저장용 | 1 | r7 | 0 | |
| 곱셈 중간 결과를 저장할 레지스터 | 8 | r8~r15 | 8 | r8~r15 |
| Carry-Catcher 레지스터 | 7 | r16~r22 | 7 | r16~r22 |
| doubling시 추가적으로 필요한 Carry-Catcher | 1 | r23 | 0 | |
| movw 를 이용한 레지스터 clear용 레지스터 | 1 | r24 | 0 | |
| Carry를 더해줄 때 사용하는 0값을 담은 레지스터 | 1 | r25 | 1 | r7 |
| 승수의 메모리 포인터용 레지스터 | 2 | r30, r31 | 2 | r28, r29 |
| 곱셈 결과를 저장할 메모리 포인터용 레지스터 | 2 | r26, r27 | 2 | r26, r27 |
| 피승수의 메모리 포인터용 레지스터 | 0 | | 2 | r30, r31 |
| Totals | 30 | | 29 | |

Systems," CHES 2004, 6th International Workshop, pp. 119-132, 2004.
[3] MIRACL, Multiprecision Integer and Rational Arithmetic C/C++ Library. http://www.shamus.ie/.
[4] Koç, Ç. K., "High-Speed RSA Implementation," Technical Report TR-201, version 2.0, RSA Laboratories, November 1994.
[5] D. Zuras, "More on Squaring and Multiplying Large Integers," *IEEE Transactions on Computers*, vol.43, no.8, pp.899-908, August 1994.
[6] A. Karatsuba and Y. Ofman, "Multiplication of Multidigit Numbers on Automata," *Soviet Physics-Doklady (English translation)*, vol.7, no.7, pp.595-596, 1963.
[7] P. Comba, "Exponentiation cryptosystems on the IBM PC," *IBM Systems Journal*, 29(4):526{538, 1990.
[8] Leif Uhsadel, Axel Poschmann, and Christof Paar, "Enabling Full-Size Public-Key Algorithms on 8-bit Sensor Nodes," In *Proceedings of ESAS 2007, volume 4572 of LNCS. Springer*, 2007. http://www.ist-ubisecsens.org/publications/ecc_esas2007.pdf.
[9] P. Szczechowiak, L. Oliveira, M. Scott, M. Collier and R. Dahab, NanoECC: "Testing the limits of elliptic curve cryptography in sensor networks, Wireless Sensor Networks," (EWSN 2008), LNCS 4913 (2008), 305-320.
[10] H. Thapliyal and M.B. Srinivas, "An Efficient Method of Elliptic Curve Encryption Using Ancient Indian Vedic Mathematics," 48th IEEE MIDWEST Symposium on Circuits and Systems (MWSCAS 2005).
[11] Donald Knuth, The Art of Computer Programming, Third Edition, Volume Two, Seminumerical Algorithms, Addison-Wesley, 1998.

김 일 희

1995년 Tokyo Technical College 정보처리학(준학사). 2005년 독학사 컴퓨터과학과(학사). 2009년 한양대학교 전자컴퓨터통신공학(석사). 2009년~현재 주식회사 UZEN u-Biz 사업부 차장

박 용 수

1996년 KAIST 전산학과(학사). 1998년 서울대학교 컴퓨터공학과(석사). 2003년 서울대학교 전기컴퓨터공학부(박사). 2003년~2004년 서울대학교 자동제어특화연구센터 박사후연수연구원. 2005년~현재 한양대학교 정보통신대학 컴퓨터전공 조교수

이 윤 호

정보과학회논문지 : 시스템 및 이론 제 36 권 제 2 호 참조