8-bit 환경에서 Lookup table 기반의 효율적인 곱셈 알고리즘

Efficient lookup Table-based Multiplication Algorithm on 8-bit Processor

서 석 충*, 정 해 일, 한 동 국, 홍 석 희*
(Seog Chung Seo, Hae-il Jung, Dong-Guk Han, Seokhie Hong)

Abstract : This paper describes some field multiplication algorithm over GF(2^m) on 8-bit processor. Through performance comparisons among algorithm, we show that our proposal is faster than existing algorithms. The proposed algorithm save 26.38% of running time compared with naive comb multiplication algorithm which is a kind of lookup-table (LUT) based algorithm. With the proposed algorithm, a scalar multiplication over GF(2^{163}) can be computed within 1.04 secs on 8-bit MICAz sensor mote.

Keywords: Elliptic Curve Cryptosystem, Field Multiplication, Sensor Mote

I. 서론

무선 센서 네트워크가 의학, 교통, 모니터링 등 많은 분야에서 사용됨에 따라서 센서 네트워크를 안전하게 만드는 것에 관한 많은 연구가 진행되고 있다. 센서 네트워크는 수백개 혹은 수천개의 메모리와 계산 능력이 제한된 센서 노드(모트)들로 구성이 되기 때문에 초기의 보안 프로토콜들은 대부분 대칭키 암호 시스템에 기반을 두었다. 하지만, 대칭키 암호 시스템은 키 공유 및 인증 측면에서 그 기능성이 떨어지기 때문에 프로토콜을 매우 복잡하게 만들며 이로 인하여 확장성이 떨어진다는 단점이 있다. 이러한 연구들은 공개키 암호 시스템은 자원이 제약된 센서모트에서 동작하기에 너무 큰 계산부하를 발생시키기 때문에 아예 논의에서 배제하였다. 하지만, 최근들어 공개키 암호시스템을 실제로 센서모트에서 효율적으로 구현하여 그 성능을 제시함으로써 공개키 암호시스템도 충분히 사용될 수 있음을 보이고 있다 [1][3][4]. 특히, 타원곡선암호 시스템 (Elliptic Curve Cryptosystem)은 기존의 RSA, DSA 암호 알고리즘에 비하여 훨씬 짧은 키 길이로도 같은 정도의 보안성을 달성할 수 있기 때문에 각광을 받고 있다.

본 논문에서는 8-bit MICAz [6] 센서 모트에서 효율적인 GF(2^m)상의 곱셈 알고리즘을 제안한다. 제안한 방법은 LUT (Lookup table)에 기반을 둔 comb 곱셈 방법의 변형으로서 8-bit 환경에서 곱셈 알고리즘이 동작할 때 발생하는 중복된 메모리 접근 연산을 줄인 것이다. 제안 알고리즘과 기존의 comb 곱셈 방법 및 다른 종류의 LUT기반의 알고리즘과 비교함으로써 제안 알고리즘의 성능 향상 폭을 제시한다. 제안 방법은 기존의 comb 곱셈 방법과 비교하여 26.38% 더 빠르다.. 제안 곱셈 알고리즘을 밑바탕으로 하여 4TNAF를 사용하는 스칼라 곱셈의 연산 시간은 1.04초로서 현재까지 MICAz 센서 모트에서 구현된 GF(2^m)상의 구현중에서 가장 빠르다.

* 책임저자(Corresponding Author)
논문접수 : 2008. 8. 12., 채택확정 : 200x. x. xx.
서석충, 정해일, 홍석희 : 고려대학교 정보보호대학원
(seosc@cist.korea.ac.kr, jung-haeil@hanmail.net, hsh@korea.ac.kr)
한동국: 전자통신 연구원
(christa@etri.re.kr)
※ 본 연구는 정보통신부 및 정보통신연구진흥원의 대학 IT연구센터 지원사업의 연구결과로 수행되었음" (IITA-2008-(C1090-0801-0025))

II. 관련 연구

1. 타원곡선 암호 개요 [2]

아래의 유한체 F 상에서 정의되는 weierstrass 식을 만족하는 해의 집합은 항등원 역할을 하는 무한원점 (Point at infinity)와 함께 아벨군을 형성한다.

$$E/F: y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6, a_i \in F \quad (1)$$

F 의 지표가 2인 경우, 위의 식은 아래와 같이 단순화 될 수 있다.

$$E/GF(2^m): y^2 + xy = x^3 + ax^2 + b,\ a,b \in F \quad (2)$$

아벨군의 규칙에 의거하여 타원곡선 상에 존재하는 두 점 P_1과 P_2 의 합의 결과 P_3 은 여전히 타원곡선 위에 존재한다. 서로 다른 두 점을 더하는 연산은 타원곡선 점 덧셈 연산 (Elliptic curve point addition: ECADD)라 불리고 같은 점을 더하는 연산은 타원곡선 점 두배 연산 (Elliptic curve point doubling: ECDBL)으로 불린다. 타원 곡선 상의 임의의 두 점을 각각 $P_1 = (x_1, y_1)$, $P_2 = (x_2, y_2)$ 라 할 때 ($P_1 \neq P_2$), P_1과 P_2 의 합의 결과인 P_3의 아핀 좌표는 아래와 같이 계산될 수 있다.

$$x_3 = \lambda^2 + \lambda + x_1 + x_2 + a, y_3 = (x_1 + x_3)\lambda + x_3 + y_1,$$
$$\lambda = \frac{(y_1 + y_2)}{(x_1 + x_2)} \text{ if } P_1 \neq P_2, \text{ and } \lambda = \frac{y_1}{x_1} + x \text{ if } P_1 = P_2 \quad (3)$$

타원곡선 암호시스템에서 가장 핵심이 되는 연산은 스칼라 곱셈 연산이다 (kP : k 는 정수, P 는 타원곡선 상의 한 점). kP 는 점 P 를 k 번 반복하여 더하는 계산이며 이는 ECADD와 ECDBL 연산을 통하여 계산된다. 이 스칼라 곱셈 연산을 효율적으로 구현하기 위하여 여러가지 선택사항이 있다. 즉, 유한체 선택 (GF(p) 또는 GF(2^m)) 좌표계 선택 (아

핀 또는 사영 좌표계), 유한체 연산 알고리즘 (유한체 곱셈, 유한체 역원, 유한체 제곱 등), 스칼라 곱셈 알고리즘 (Binary method, wNAF 기반의 스칼라 곱셈, wTNAF 기반의 스칼라 곱셈)의 선택이 그것들이다. 본 논문에서는 유한체 $GF(2^m)$에 서의 유한체 곱셈 연산을 8-bit 프로세서에서 최적화시키는 것에 대하여 논의한다. 또한 본 논문의 구현은 $GF(2^m)$ 상의 Koblitz 커브에 기반을 두었으며, 스칼라 곱셈을 계산하기 위하여 wTNAF 방법을 사용하였다 [2][7].

III. 제안 알고리즘

본 절에서는 $GF(2^m)$ 상의 곱셈 연산을 효율적으로 계산하는 알고리즘을 제시한다. 제안 알고리즘은 윈도우를 사용하는 comb 곱셈 알고리즘의 변형으로서, 일반적인 comb 곱셈 알고리즘의 동작 시에 발생하는 중복된 메모리 접근연산을 줄인 것이다.

1. 표기법 정리

다음 표기법들은 본 논문에서 알고리즘 기술 시에 사용된다. $a(z)$ 와 $b(z)$ 는 $GF(2^m)$ 의 원소라고 가정한다.

$a(z)\ll i$: 상위 비트들을 0으로 패딩하면서 $a(z)$ 를 오른쪽으로 i 비트만큼 시프트한다.

$a(z)\gg i$:하위 비트들을 0으로 패딩하면서 $a(z)$ 를 왼쪽으로 i 비트만큼 시프트한다.

W : 하나의 8비트 워드, w : comb 곱셈에서 사용되는 윈도우 크기

&: 비트와이즈 AND.

$U(W)$: ($W\gg 4$)를 반환한다.

$L(W)$: ($W \& 0x0F$)를 반환한다.

$t=\left\lceil \frac{m}{W} \right\rceil$: 다항식 $a(z)$ 를 메모리에 저장하기 위해 필요한 워드의 수이다.

$a(z)$, $b(z)$ 는 각각 $\sum_{i=0}^{m-1} a_i z^i, \sum_{i=0}^{m-1} b_i z^i$ 로 표현되는 다항식이다.

윈도우를 사용하는 왼쪽에서 오른쪽으로 동작하는 (left-to-right, ltr) comb 곱셈의 계산은 다음과 같이 표현된다.

$$a(z)b(z)=(\cdots((\tilde{a}_{s-1}b(z)z^w+\tilde{a}_{s-2}b(z))z^w+\cdots\tilde{a}_1 b(z))z^w+\tilde{a}_0 b(z),\ \tilde{a}_i=(a_{wi+w-1}\cdots a_{wi+1}a_{wi}),$$

$$0\le i\le s-1, s=\left\lfloor \frac{m}{w} \right\rfloor+1.$$

2. 윈도우를 사용하는 comb 곱셈 방법

$GF(2^m)$ 상에서의 효율적인 곱셈 알고리즘으로서 윈도우를 사용하는 comb 곱셈 방법이 많이 사용된다. $a(z)b(z)$ 의 곱셈을 연산하기 위하여, 윈도우를 사용하는 comb 곱셈 방법은 두 단계로 구성된다. 첫 번째 단계는 $wb(z)$ 를 사전계산하는 것이다 (Algorithm 1의 과정 3). 예를 들어, $w=4$ 일 경우에, $b(z)$ 에서 $15b(z)$ 까지 사전에 계산한다. 실제 계산은 사전에 계산한 값을 이용하여 처리한다. 실제로 곱셈을 수행하는 단계에서는 승수 $a(z)$ 를 W 비트의 컬럼 단위로 나열한 후, 왼쪽에서부터 w씩 스캔한 후 (상위 블럭을 처리한 후에 하위 블럭을 처리)에 사전에 미리 계산한 값을 가지고와 중간 결과값에 더한다 ($GF(2^m)$ 상의 덧셈 연산은 XOR 연산이다, Algorithm 1의 과정 5와 7, 이 과정을 부분곱셈이라 부른다). 즉, [그림 1]에서 상위 w 비트 블록을 위에서 아래로 계산한 후에, 하위 w 비트 블록을 처리하는 것이다. 스캔한 w 비트 값에 해당하는 값을 사전계산 테이블로부터 로드하여 중간 결과값에 더하는 것이다.

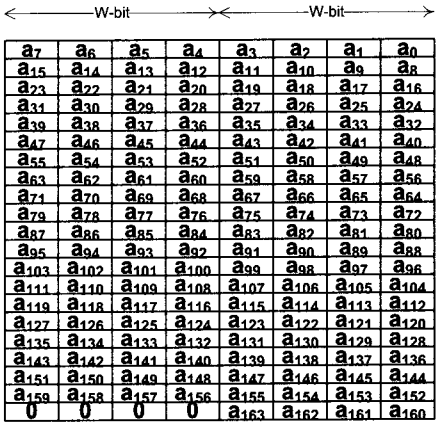

그림 1. $a(z)b(z)$ 연산에서 승수 $a(z)$ 표현, 상위 w 비트 블럭과 하위 w 비트 블럭으로 구성된다.

Algorithm 1. Left-to-right comb 곱셈 ($w=4$)

1. *Input* : $a(z), b(z)$ *over* $GF(2^m)$
2. *Output* : $c(z)=a(z)b(z)$
3. *Compute* $T_u=u(z)b(z)$ *for* $u(z)$ *of* degree $w-1$
4. $C\leftarrow 0$
5. *for* $j\leftarrow 0$ *to* $t-1$ *do*
   $u\leftarrow U(a[j])$
   for $i\leftarrow 0$ *to* t *do*
   $C[i+j]=C[i+j]\oplus T_u[i]$
6. $C\leftarrow C\ll w$
7. *for* $j\leftarrow 0$ *to* $t-1$ *do*
   $u\leftarrow L(a[j])$
   for $i\leftarrow 0$ *to* t *do*
   $C[i+j]=C[i+j]\oplus T_u[i]$
8. Return (C)

3. 제안 방법

기존의 윈도우를 사용하는 ltr comb 곱셈 방법이 8-bit 프로세서에서 구현될 때 중복된 메모리 접근 연산이 발생한다. 즉, Algorithm 1에서 5와 7은 부분 곱셈 값을 계산하는 부분이다. 과정 5와 7에서 C[i+j]가 매번 로드되고 연산의 결과가 저장되고 있음을 알 수 있

Algorithm 2. 제안 곱셈 방법 ($w=4$)

1. *Input* : $a(z), b(z)$ *over* $GF(2^m)$
2. *Output* : $c(z) = a(z)\square b(z)$
3. *Compute* $T_u = u(z)\square b(z)$ *for* $u(z)$ *of* degree $w-1$
4. $C \leftarrow 0$
5. *for* $j \leftarrow 0$ *to* $t-2$ by 4 *do*
   $u_1 \leftarrow U(a[j]), u_2 \leftarrow U(a[j+1]), u_3 \leftarrow U(a[j+2]), u_4 \leftarrow U(a[j+$
   $C[j] = C[j] \oplus T_{u_1}[0];$
   $C[j+1] = C[j+1] \oplus T_{u_1}[1] \oplus T_{u_2}[0];$
   $C[j+2] = C[j+2] \oplus T_{u_1}[2] \oplus T_{u_2}[1] \oplus T_{u_3}[0];$
   for $i \leftarrow 3$ *to* t by 1 *do*
     $C[i+j] = C[i+j] \oplus T_{u_1}[i] \oplus T_{u_2}[i-1] \oplus T_{u_3}[i-2] \oplus T_{u_4}[i-3];$
   $C[j+t] = C[j+t] \oplus T_{u_2}[t-1] \oplus T_{u_3}[t-2] \oplus T_{u_4}[t-3];$
   $C[j+t+1] = C[j+t+1] \oplus T_{u_3}[t-1] \oplus T_{u_4}[t-2];$
   $C[j+t+2] = C[j+t+2] \oplus T_{u_4}[t-1];$
6. $C \leftarrow C\square z^w$
7. *for* $j \leftarrow 0$ *to* $t-2$ by 4 *do*
   $u_1 \leftarrow L(a[j]), u_2 \leftarrow L(a[j+1]), u_3 \leftarrow L(a[j+2]), u_4 \leftarrow L(a[j+3$
   $C[j] = C[j] \oplus T_{u_1}[0];$
   $C[j+1] = C[j+1] \oplus T_{u_1}[1] \oplus T_{u_2}[0];$
   $C[j+2] = C[j+2] \oplus T_{u_1}[2] \oplus T_{u_2}[1] \oplus T_{u_3}[0];$
   for $i \leftarrow 3$ *to* t by 1 *do*
     $C[i+j] = C[i+j] \oplus T_{u_1}[i] \oplus T_{u_2}[i-1] \oplus T_{u_3}[i-2] \oplus T_{u_4}[i-3];$
   $C[j+t] = C[j+t] \oplus T_{u_2}[t-1] \oplus T_{u_3}[t-2] \oplus T_{u_4}[t-3];$
   $C[j+t+1] = C[j+t+1] \oplus T_{u_3}[t-1] \oplus T_{u_4}[t-2];$
   $C[j+t+2] = C[j+t+2] \oplus T_{u_4}[t-1];$
8. $u_1 \leftarrow L(a[t-1]);$
   for $i \leftarrow 0$ *to* t by 1 *do*
     $C[i+t] = C[i+t] \oplus T_{u_1}[i];$
9. *Return* (C)

다. 즉, 연속한 과정 5와 7의 수행에서 C[i+j]가 중복되어 로드되며, 마찬가지로 결과값 역시 같은 장소에 매번 저장되고 있다. 따라서, 연속한 과정 5 또는 7을 통합함으로써 중복된 메모리 접근을 줄일 수있다. 본 논문에서는 4개의 연속한 과정 5와 7을 하나로 통합하여 중복된 메모리 접근을 줄였으며, 또한 $a(z)$ 의 최상위 워드의 상위 w 비트는 항상 0이기 때문에, 이에 대하여 과정 5를 수행할 필요가 없다. 제안 방법에서는 과정 5와 7을 통합하여 중복된 메모리 접근을 줄였으며, 또한 최상위 워드의 상위 w 비트에 대한 과정 5의 연산을 생략함으로써 연산 시간을 줄였다.

Algorithm 2의 과정 5와 7은 Algorithm 1의 과정 5와 7의 4번의 수행을 하나로 통합한 것이다. 따라서, for-loop의 카운터가 4씩 증가한다. 4개의 부분 곱셈을 하나로 통합한 이유는 본 논문의 구현은 m=163인 $GF(2^{163})$에 기반을 두었기 때문이다. 즉, $GF(2^{163})$의 하나의 원소를 8-bit 프로세서의 메모리에 저장할 때, 21 (= $\lceil(163/8)\rceil$)개의 워드가 사용된다. 따라서, 과정 5와 7에 대한 4번의 수행을 하나로 통합하였을 때, 사후 처리 과정이 간단해진다. 뿐만 아니라, 이를 통하여 최상위 워드의 상위 w 비트에 대해 수행되는 과정 5의 수행을 자동적으로 생략할 수 있다. 반면, 하위 w 비트에 대해서는 과정 7의 연산이 수행되어야 하므로, 과정 8을 통하여 사후 처리를 한 것이다.

표 1. $GF(2^{163})$에서 LUT 에 기반을 둔 곱셈 알고리즘들의 성능 비교

	유한체 곱셈 연산 시간
Algorithm 1	0.00370277
Algorithm 2	0.00272593
유한체 곱셈 에뮬레이트	0.00610253

표 2. 기존의 구현들과의 성능 비교 (RAM 과 ROM 의 단위는 바이트이다)

	스칼라 곱셈 연산 시간	RAM 사용량	ROM 사용량
NanoECC (GF(p))	1.27 초	1,843	47,206
NanoECC ($GF(2^m)$)	2.16 초	1,740	33,177
This Work	1.04 초	618	5,892

4. 유한체 곱셈 에뮬레이팅

작은 워드 크기의 프로세서에서 $GF(2^m)$의 유한체 곱셈 연산이 GF(p)에 비하여 상대적으로 느린 이유는 워드 단위의 정수 곱셈 연산이 명령어 레벨에서 제공되는 것에 반하여, 캐리를 발생하지 않는 다항식 곱셈 연산이 명령어 레벨에서 제공되지 않기 때문이다. 몇몇 프로세서에서는 $GF(2^m)$에서의 곱셈 연산을 명령어 레벨에서 제공하지만 MICAz 센서 모트가 사용하는 ATmega128 프로세서에서는 지원되지 않는다. 따라서, $GF(2^m)$의 곱셈을 LUT을 이용하여 에뮬레이팅하여 실험해보았다. 이 경우에는, 앞 절의 알고리즘과는 달리 연산되는 다항식 값에 상관없이 LUT이 고정적이다. 8-bit의 두 워드의 다항식 곱셈과 정확히 일치하도록 에뮬레이트하기 위해서는 $2*2^8*2^8$-byte가 필요하다. 하지만, 이것은 4-Kbyte의 RAM을 가진 MICAz 모트에서는 불가능하다. 따라서, 8-bit의 두 워드의 곱셈을 4-bit 단위로 나누어서 4개의 4-bit 곱셈으로 처리하였다. 따라서, LUT의 크기는 2^4*2^4-byte가 된다. 또한, 앞절에서 제안한 방법과 마찬가지로 연속한 곱셈 연산을 통합하여 중복된 메모리 접근을 줄였다.

IV. 성능 분석

[표 1]은 Algorithm 1과 제안 알고리즘 그리고 유한체 곱셈을 LUT를 이용하여 에뮬레이트하는 알고리즘의 성능을 비교한 것이다. 곱셈을 에뮬레이트한 경우의 성능이 가장 떨어지는데 이는 8-bit의 두 워드의 곱셈을 에뮬레이트하기 위하여, 4번의 4-bit 곱셈을 에뮬레이트 하기 때문이다. Algorithm 2는 세가지 알고리즘중 가장 빠르며, Algorithm 1의 연산시간을 26.38% 단축시키고 있다 (알고리즘들을 TinyOS [5]상에 nesC 언어 [8]를 이용하여 구현하여 MICAz 모트에서 구동하였다).

[표 2]는 본 논문의 연구 결과를 기존의 연구결과와 비교하고 있다. 본 논문의 연구 결과는 현재까지 가장 효율적으로 알려진 NanoECC에 비하여 연산 시간 및 메모리 사용량

측면에서 더욱 뛰어나다. NanoECC의 경우에는 MIRACL 암호 라이브러리에 기반을 둔 것이기 때문에 코드 크기가 상대적으로 크다.

참고문헌

[1] D. J. Malan, M. Welsh, and M. D. Smith, "A Public-Key Infrastructure for Key Distribution in TinyOS Based on Elliptic Curve Cryptography," In the first IEEE international conference on Sensor and Adhoc communications and Networks (SECON04), 2004.

[2] D. Hankerson, A. J. Menezes, and S. Vanstone, "Guide to Elliptic Curve Cryptography," Springer-Verlag, 2004.

[3] P. Szczechowiak, L. B. Oliveira, M. Scott, M. Collier, and R. Dahab, "NanoECC: Testing the Limits of Elliptic Curve Cryptography in Sensor Networks," European Wireless Sensor Networks (EWSN 2008), LNCS 4913, pp. 305-320, 2008.

[4] N. Gura, A. Patel, A. Wander, H. Eberle, and S. Chang-Shantz, "Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs," CHES 2004, LNCS 3156, pp. 119-132, 2004.

[5] TinyOS forum, Available at "http://www.tinyos.net/".

[6] MICAz Hardware Description Available at "http://www.xbow.com/Products".

[7] J. Solinas, "Efficient Arithmetic on Koblitz curves," Design, Codes and Cryptography, 19:195-249, 2000.

[8] nesC language reference manual, Available at "www.tinyos.net/api/nesc/doc/ref.pdf"

서 석 충 (Seog Chung Seo)

2005년 아주대학교 정컴학부 학사. 2007년 광주과기원 정보통신 석사. 2007년 ~현재 고려대학교 정보경영공학전문대학원 박사과정. 관심분야는 부채널 공격, 공개키 암호시스템 안전성 분석 및 고속구현, 타원곡선 등임.

정 해 일 (Hae-il Jung)

2008년 동의대학교 수학과 학사.
2008년 ~ 현재 고려대학교 정보경영공학전문대학원 석사과정.
관심분야 암호시스템 고속구현 및 분석, Side-channel Attack.

한 동 국 (Dong-Guk Han)

1999년 고려대학교 수학과 졸업. 2002년 고려대학교 수학과 석사(이학석사). 2005년 고려대학교 정보보호대학원 박사(공학박사). 2004년~2005년 일본 Kyushu Univ., 방문연구원. 2005년~2006년 일본 Future Univ.-Hakodate, Post.Doc. 2006년 6월~현재 한국전자통신연구원 정보보호연구단 선임연구원. 관심분야는 암호시스템 안전성 분석 및 고속 구현, 부채널 분석, RFID/USN 정보보호 기술등임
관심분야: 공개키 암호시스템 안전성 분석 및 고속 구현, 부채널 분석, RFID/USN 정보보호 기술

홍 석 희 (Seokhie Hong)

1995년 고려대학교 수학과 학사. 1997년 고려대학교 수학과 석사. 2001년 고려대학교 수학과 박사. 1999년~2004년 (주)시큐리티 테크놀로지스 선임연구원. 2003년~2004년 고려대학교 시간강사. 2004년~2005년 K.U. Leuven 박사후연구원. 2005년~현재 고려대학교 정보경영전문대학원 조교수. 관심분야는 대칭키 암호 알고리즘, 공개키 암호 알고리즘, 포렌식등