情報保護學會論文誌
第18卷 第3號, 2008. 6

TinyECCK : 8 비트 Micaz 모트에서 $GF(2^m)$상의 효율적인 타원곡선 암호 시스템 구현*

서 석 충[1*†], 한 동 국[2**‡], 홍 석 희[1*]
[1]고려대학교 정보경영공학전문대학원, [2]한국전자통신연구원

TinyECCK : Efficient Implementation of Elliptic Curve Cryptosystem over $GF(2^m)$ on 8-bit Micaz Mote

Seog Chung Seo[1†], Dong-Guk Han[2‡], Seokhie Hong[1]
[1]Graduate School of Information Management and Security, Korea University,
[2]Electronics and Telecommunications Research Institute

요 약

본 논문에서는 "작은 워드 크기를 사용하는 센서모트에서는 $GF(2^m)$상의 partial XOR 곱셈연산이 저전력 마이크로프로세서에 의하여 효율적으로 지원되지 않기 때문에 $GF(2^m)$에 기반을 둔 타원곡선 암호시스템의 소프트웨어 구현은 비효율적이다"라는 일반적으로 인정된 의견을 검증한다. 비록 센서모트에서 $GF(2^m)$에 기반을 둔 몇 가지의 소프트웨어 구현은 있지만, 이것들의 성능은 센서네트워크에서 사용할 만큼 충분하지 못하다. 기존 구현들의 성능 저하는 유한체 곱셈과 감산 연산에서 발생하는 중복된 메모리 접근에서 기인한다. 따라서 본 논문에서는 유한체 곱셈과 감산과정에서 발생하는 불필요한 메모리 접근을 줄일 수 있는 몇 가지 방법을 제안한다. 제안한 방법을 통하여, $GF(2^{163})$상의 유한체 곱셈과 감산의 수행시간을 각각 21.1%와 24.7% 줄일 수 있으며 이것은 Elliptic Curve Digital Signature Algorithm (ECDSA)의 sign과 verify 연산 시간을 약 15~19% 단축시킬 수 있다.

ABSTRACT

In this paper, we revisit a generally accepted opinion: implementing Elliptic Curve Cryptosystem (ECC) over $GF(2^m)$ on sensor motes using small word size is not appropriate because partial XOR multiplication over $GF(2^m)$ is not efficiently supported by current low-powered microprocessors. Although there are some implementations over $GF(2^m)$ on sensor motes, their performances are not satisfactory enough due to the redundant memory accesses that result in inefficient field multiplication and reduction. Therefore, we propose some techniques for reducing unnecessary memory access instructions. With the proposed strategies, the running time of field multiplication and reduction over $GF(2^{163})$ can be decreased by 21.1% and 24.7%, respectively. These savings noticeably decrease execution times spent in Elliptic Curve Digital Signature Algorithm (ECDSA) operations (Signing and verification) by around 15~19%.

Keywords : Elliptic Curve Cryptography, Wireless Sensor Networks, Efficient Implementation

Ⅰ. 서 론

대칭키 기반의 암호 프로토콜은 제약된 기능성으로 인하여 센서 네트워크 (Wireless Sensor Networks : WSNs)에서 공유키 설정과 브로드캐스트 메시지 인증 과정에 적합하지 않다. 지금까지 공개키 암호 시스템, 특히 타원곡선 암호 (Elliptic Curve Cryptography : ECC)를 센서네트워크에 적용하는 것에 관한 많은 연구가 진행되어 왔다. 이러한 연구들은 타원곡선 암호시스템을 실제로 구현하여 동작시간과 코드 크기에 관한 성능을 제시함으로써 센서 네트워크에 타원곡선 암호를 적용하는 것은 가능하다고 결론지었다[1,8,10,15].

지금까지 상대적으로 만족할만한 성능을 제시한 타원곡선 암호 구현들은 모두 $GF(p)$에 기반을 두었다 [1,8,15]. 반면, $GF(2^m)$에 기반을 둔 구현들은 센서네트워크에 적용할 만큼 충분한 성능을 제시하지 못하였다 [4,7,9,10]. 몇몇 논문에서 $GF(2^m)$상에서의 유한체 곱셈이 작은 워드 크기를 사용하는 저전력 마이크로프로세서에 의해 효율적으로 지원되지 않기 때문에 센서 모트에서 $GF(2^m)$에 기반을 둔 타원곡선의 소프트웨어 구현은 비효율적이라고 주장하였다[1,8,10,15]. 본 논문에서는 이러한 주장을 재검증해보고 실제로 $GF(2^m)$상에서의 유한체 곱셈이 $GF(p)$상의 곱셈보다 더 빠를 수 있음을 제시한다. 현재 센서 모트에서 $GF(2^m)$에 기반을 둔 타원곡선의 소프트웨어 구현에 관한 다음과 같은 오해가 있다.

첫째, 비효율적인 유한체 곱셈 : 타원곡선 암호연산 중에서 가장 빈번히 계산되는 연산은 유한체 곱셈과 감산 연산이다. 작은 크기의 워드를 사용하는 저전력 마이크로프로세서에서 $GF(2^m)$상의 유한체 곱셈은 partial XOR 곱셈을 요구하기 때문에 $GF(p)$상의 유한체 곱셈에 비하여 비효율적인 것으로 인식되고 있다[1]. 실제로 현재의 저전력 마이크로프로세서들은 paritial XOR 곱셈을 명령어 레벨에서 제공하지 않고 있다.

둘째, ECDSA 구현에 따른 과중한 메모리 소모량 : $GF(2^m)$에 기반을 둔 ECDSA 구현은 전자 서명의 생성과 검증을 위하여 $GF(2^m)$상의 유한체 연산 뿐만 아니라 $GF(p)$상의 유한체 연산도 필요로 한다. 따라서 $GF(2^m)$상에서의 ECDSA 구현은 $GF(p)$상에서의 ECDSA 구현보다 많은 양의 코드를 요구한다고 생각될 수 있다. 실제로 센서모트에서 $GF(2^m)$상의 타원곡선 구현은 대부분 Elliptic Curve Diffie-Hellman (ECDH)만 제공하고 있다. 하지만 $GF(2^m)$상에서 최적화된 ECDSA 구현의 코드 크기는 $GF(p)$상에 기반을 둔 ECDSA의 코드 크기와 견줄만하다. 본 논문에서 제안하는 TinyECCK는 최적화된 ECDSA 구현으로서 현재 센서 모트에서 가장 효율적인 타원곡선 암호의 소프트웨어 구현으로 알려진 TinyECC보다 더 적은 코드 크기를 이용하여 더 빠른 연산을 제공한다.

본 논문의 기여는 다음과 같다.

첫째, $GF(2^m)$상에서의 유한체 곱셈이 $GF(p)$상의 유한체 곱셈보다 빠를 수 있음을 보인다 : $GF(2^m)$상의 유한체 곱셈과 감산 연산은 많은 수의 메모리 접근 연산 (LOAD와 STORE 명령)을 필요로 한다. 본 논문에서는 유한체 곱셈과 감산과정에서 불필요하게 중복된 메모리 접근 명령의 수를 줄일 수 있는 방법을 제시한다. 제안하는 방법을 통하여 $GF(2^{163})$에서의 유한체 곱셈과 감산에 대한 연산 시간을 각각 21.1%와 24.7% 절약할 수 있다. 이때, 제안된 유한체 곱셈은 sect160r1상에서 최적화된 유한체 곱셈보다 7.4% 빠르다.

둘째, 현재까지 $GF(2^m)$에 기반을 둔 타원곡선의 소프트웨어 구현 중에서 가장 뛰어난 성능을 제공한다 : TinyECCK는 기존의 $GF(2^m)$상에서 구현된 모든 타원곡선 소프트웨어들의 성능을 능가한다. 뿐만 아니라 $GF(p)$의 타원 곡선 암호 구현을 포함하여 ATmega128 프로세서에서 C언어 또는 C언어와 inline assembly를 혼합하여 구현한 것들에 비하여 더욱 빠르다.

셋째, TinyECCK는 Koblitz 커브상에서 타원곡선 암호 연산을 구현하였다. Koblitz 커브에서 두 배 연산은

접수일 : 2007년 11월 26일; 채택일 : 2008년 3월 16일
* "본 연구는 지식경제부 및 정보통신연구진흥원의 대학 IT 연구센터 지원사업의 연구결과로 수행되었음" (IITA-2008-(C1090-0801-0025)).
** "본 연구의 일부는 지식경제부 및 정보통신연구진흥원의 대학 IT연구센터 지원사업의 연구결과로 수행되었음 (IITA-2008-(C1090-0801-0025)), 그리고 정보통신부 및 정보통신연구진흥원의 IT신성장동력핵심기술개발사업[2005-S-088-04,안전한 RFID/USN 을 위한 정보보호 기술] 사업의 일환으로 수행하였음."
† 주저자, seosc@cist.korea.ac.kr
‡ 교신저자, christa@etri.re.kr

1) Partial XOR 곱셈 : $GF(2^m)$상에서 곱셈 연산이 수행될 때 부분적인 곱셈 값들이 XOR연산을 통하여 더해지는 것을 의미한다. 이때 $GF(2^m)$의 특성상 캐리는 발생하지 않는다. 이 연산은 일반적으로 명령어 레벨에서 지원되지 않는다.

간단한 유한체 제곱연산들로 대체될 수 있기 때문에 일반적인 커브에 비하여 더욱 빠른 연산을 제공한다.

Ⅱ. 관련 연구

지금까지 센서모트 상에서 $GF(2^m)$과 $GF(p)$에 기반을 둔 여러 타원곡선 암호 소프트웨어가 구현되었다. 이러한 구현들은 센서 네트워크에서 타원곡선 암호의 실행가능성을 검증하기 위하여 개발되었다.

2.1 $GF(2^m)$에 기반을 둔 소프트웨어 구현들

Malan 외는 8 비트 센서 모트에서 $GF(2^m)$에 기반을 둔 첫 번째 구현인 EccM을 개발하였다. 그들은 UC Berkeley에서 개발된 TinySec[2] 암호 모듈에 키 분배 메커니즘을 제공하기 위하여 타원곡선 암호를 이용하였다[4]. EccM의 코드 크기는 34,342 바이트이며 하나의 공개키를 생성하는데 34초가 걸린다. Yan과 shi는 센서 모트와 같은 작은 임베디드 장치에서 $GF(2^m)$에 기반을 둔 타원곡선 암호의 소프트웨어 구현은 여전히 느리다고 지적하였으며, 8 비트 ATmega128 프로세서에서 $GF(2^{163})$상의 빠른 모듈러 감산을 적용한 타원곡선 암호를 구현하였다[9]. Yan과 shi가 구현한 코드 크기는 11,592 바이트이며 한 번의 스칼라 곱셈을 계산하는데 13.9초가 걸렸다. Eberle 외는 $GF(2^m)$상에서의 유한체 연산들이 (특히 유한체 곱셈) 저전력 마이크로프로세서에 의해서 효율적으로 지원되지 않기 때문에 매우 느리다고 지적하였다[10]. 또한 이들은 명령어 집합 확장 (Instruction Set Extension)을 이용하는 추가적인 아키텍처 확장을 통하여 $GF(2^m)$상의 타원곡선 연산이 $GF(p)$ 상에서의 연산 보다 빨라질 수 있다고 주장하였다. 실제로 $GF(2^{163})$상에서 스칼라 곱셈을 어셈블리 언어로 구현한 결과 4.14 초가 걸렸지만 아키텍처 확장을 이용한 경우 0.29 초 만이 걸렸다. 이 결과는 $GF(2^m)$상의 타원곡선 암호를 저전력 프로세서 상에서 구현할 때 소프트웨어로 구현하는 것보다는 하드웨어로 구현하는 것이 더욱 바람직하다는 주장을 뒷받침해준다. Blaß와 Zitterbart는 $GF(2^{113})$ 위에서 ECDH, ECDSA 그리고 El-Gamal를 구현하고 이것들의 성능을 EccM과 비교하였다[7]. 이들의 구현은 서명을 생성할 때와 검증할 때 각각 6.88초와 24.17초가 걸리며 코드의 크기는 75,088 바이트였다.

2.2 $GF(p)$에 기반을 둔 소프트웨어 구현들

Gura 외는 센서네트워크에서 공개키 암호 시스템의 적용가능성을 증명하기 위하여 8 비트 ATmega128 프로세서 상에서 어셈블리 코드와 명령어 집합 확장을 이용하여 RSA와 $GF(p)$상의 타원곡선 암호를 구현하였으며 구현된 RSA와 타원곡선 암호의 성능을 비교하였다[15]. 구현된 타원곡선 암호는 한 번의 스칼라 곱셈을 계산하는데 0.81초 걸리며 이것은 센서 네트워크에서 타원곡선 암호의 사용가능성을 뒷받침 한다. 그들은 또한 유한체 곱셈 연산중에 메모리 접근 명령의 수를 줄이기 위하여 오퍼랜드 곱셈 방식과 프로덕트 곱셈 방식의 장점을 결합한 혼합 곱셈 알고리즘 (Hybrid multiplication algorithm)을 제안하였다. Liu 외가 개발한 TinyECC[1]는 TinyOS[16] 상에서 동작하는 타원곡선 암호의 소프트웨어 패키지로서 $GF(p)$상에 기반을 둔 스칼라 곱셈 및 ECDSA 연산의 기능을 제공한다. TinyECC는 효율적인 연산을 위하여 pseudo-Mersenne prime을 이용하는 최적화된 모듈로 감산, 슬라이딩 윈도우 스칼라 곱셈, Jacobian 좌표계, 그리고 혼합 곱셈/제곱 알고리즘을 적용하였다. 유한체 곱셈, 제곱, 감산과 같은 성능의 핵심 부분을 인라인 어셈블리로 구현하여 하나의 서명을 생성하고 검증하는데 각각 2초와 2.43초가 걸리며 이때의 코드 크기는 19,308 바이트였다. C언어로 구현된 TinyECC는 15,872 바이트의 줄어든 코드가 사용되지만 서명을 생성하고 검증하는데 6.26초와 7.92초가 소모된다. 지금까지 8 비트 워드 크기를 사용하는 저전력 장치에서 소프트웨어로 구현된 타원곡선 암호 시스템의 성능은 $GF(p)$상에 기반을 둔 것들이 $GF(2^m)$상에 기반을 둔 것들보다 더욱 뛰어난 성능을 보였다.

이러한 기존의 결과들을 보면 저전력 장치에서 $GF(p)$ 위의 타원곡선 암호의 소프트웨어 구현이 $GF(2^m)$에서의 것보다 더욱 효율적인 것으로 보이며, $GF(2^m)$의 사용은 하드웨어적으로 구현될 때만 적합한 것으로 보인다. 하지만 본 논문에서는 8 비트 센서 모트 상에서 $GF(2^m)$ 상의 최적화된 타원곡선 암호 소프트웨어 구현 (Tiny ECCK)의 성능이 $GF(p)$에서 최적화된 구현 (TinyECC)의 성능보다 더욱 뛰어날 수 있음을 보인다.

Ⅲ. 타원곡선 암호 시스템의 개요

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

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

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

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

아벨군의 규칙에 의거하여 타원곡선 상에 존재하는 두 점 P_1과 P_2의 합의 결과 P_3은 여전히 타원곡선 위에 존재한다. 서로 다른 두 점을 더하는 연산은 타원곡선 점 덧셈 연산 (Elliptic curve point addition : ECADD) 이라고 불리고 같은 점을 더하는 연산은 타원곡선 점 두배 연산 (Elliptic curve point doubling : ECDBL)으로 불린다. 곡선상의 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_2의 합의 결과인 P_3의 아핀 좌표는 아래와 같이 계산될 수 있다.

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

아핀 좌표계에서 ECADD와 ECDBL은 각각 1번의 유한체 역원연산과 두 번의 유한체 곱셈연산을 필요로 한다. 역원연산이 곱셈연산에 비하여 막대한 부하를 가져올 경우 사영 좌표계를 사용하는 것이 유리하다. 예를 들어, López-Dahab (LD) 사영 좌표계에서 ECADD와 ECDBL은 각각 14번의 곱셈과 4번의 곱셈을 요구한다[2]. 따라서 구현 환경에서 I > 7M (I = 역원, M = 곱셈)인 경우에 LD 좌표계를 사용하는 것이 더욱 효율적이다. $GF(2^m)$상에서 하나의 점 P를 연속하여 k번 더하는 것을 스칼라 곱셈 (scalar multiplication)이라고 부르며, $Q=kP$로 표현된다. 이 스칼라 곱셈은 ECDH와 ECDSA 연산에서 핵심이 되는 계산이다.

Ⅳ. 구현 세부 사항

8 비트 ATmega128 프로세서를 기반으로 하는 Micaz [14] 센서모트에서 TinyECCK를 구현하였다. Tiny ECCK를 구현하기 위하여 권고된 도메인 파라미터인 sect163k1 [3]를 이용하였으며, $GF(2^m)$상의 원소를 표현하기 위하여 다항식 기저를 사용하였다. 본 절에서 기술된 알고리즘들은 Guide to Elliptic Curve Cryptography[5,6]에 제시된 32 비트 기반의 $GF(2^m)$ 유한체 연산 알고리즘들에 기반을 둔 것으로서 8 비트 환경에 맞게 수정하였다. 연산의 효율성을 위하여 TinyECCK에 wNAF, wTNAF 리코딩 알고리즘을 적용하였으며 혼합좌표계[12]를 이용하여 ECADD, ECDBL 연산을 하도록 구현하였다. 구현 환경은 TinyOS 1.1.10Jan 상에서 NesC를 이용하였다. 또한 컴파일 시에 optimization level을 2로 설정하였다. 공평한 비교를 위하여 optimization level 2로 컴파일된 TinyECC의 성능과 비교를 행하였다 ([표 6] 참조).

4.1 표기법 정리

$A \oplus B$: 비트와이즈 XOR.

$A \& B$: 비트와이즈 AND.

$A \gg i$: 상위 비트들을 0으로 패딩하면서 A를 오른쪽으로 i 비트만큼 이동시킨다.

$A \ll i$: 하위 비트들을 0으로 패딩하면서 A를 왼쪽으로 i 비트만큼 이동시킨다.

W: 하나의 8 비트 워드.

$U(W)$: $(W \gg 4)$를 반환한다.

$L(W)$: $(W \& 0x0F)$를 반환한다.

$A[j]$는 다항식 A의 j번째 워드를 가리킨다.

$A\{j\}$는 다항식 A의 j번째 워드부터 n번째 최상위 워드까지를 가리킨다, $(A[n],...,A[j+1],A[j])$.

$A\{j,...,i\}$는 다항식 A의 i번째 워드부터 j번째 워드까지를 가리킨다, $(A[j],...,A[i+1],A[i])$, $j \geq i$.

$t=\lceil m/W \rceil$ 는 다항식 A를 메모리에 저장하기위해 필요한 워드의 개수이다.

ATmega128 프로세서가 8 비트 워드의 메모리 주소로 동작하기 때문에 알고리즘에서 워드의 크기를 8 비트로 가정한다. 다음 표기법들은 본 논문에서 알고리즘 기술 시에 사용된다. A와 B는 $GF(2^m)$의 원소라고 가정한다.

---

2) López-Dahab 사영좌표계에 관한 자세한 내용은 [5,6,12]를 참조하라.

4.2 $GF(2^m)$상에서의 유한체 연산

4.2.1 유한체 제곱 연산

$GF(2^m)$의 한 원소 $a(z)=a_{m-1}z^{m-1}+...+a_2z_2+a_1z+a_0$를 제곱하는 것은 $a(z)$의 이진 표현의 연속한 두 비트 사이에 0을 삽입하여 결과적으로 $a(z)^2=a_{m-1}z^{2m-2}+\cdots+a_2z^4+a_1z^2+a_0$이 된다. 따라서 사전계산을 통한 table lookup을 통하여 효율적으로 계산할 수 있다[3]. Algorithm 1은 하나의 워드를 제곱하여 두 워드로 확장한다. 상위 4 비트와 하위 4 비트의 홀수 위치에 0을 삽입하여 각각 8 비트 워드가 생성된다.

Algorithm 1. Polynomial squaring

---

1. **INPUT** : A binary polynomial $a(z)$ of degree at most $m-1$
2. **OUTPUT** : $c(z)=a(z)^2$

---

3. **Precomputation**. For each 4-bit $d=(d_3,d_2,d_1,d_0)$, compute the 8-bit value $T(d)=(0,d_3,0,d_2,0,d_1,0,d_0)$.
4. **for** $i\leftarrow 0$ **to** $t-1$ **do**
5. Let $A[i]=(d_7,d_6,d_5,d_4,d_3,d_2,d_1,d_0)$ where d_j is a bit.
6. $C[2i]\leftarrow T(d_3,d_2,d_1,d_0)$, $C[2i+1]\leftarrow T(d_7,d_6,d_5,d_4)$.
7. **end for**
8. **Return** (C)

4.2.2 유한체 곱셈 연산

유한체 곱셈은 스칼라 곱셈 연산 중에 계산되는 가장 빈번한 연산중의 하나이기 때문에 효율적으로 구현되어야 한다. 비록 shift-and-add 방법이 직관적이지만 많은 양의 메모리 접근과 워드 shift 때문에 소프트웨어 구현에서는 바람직하지 않다. 실험을 통하여 shift-and- add, right-to-left comb 방법에 비하여 윈도우 기법을 적용한 left-to-right comb 방법이 8 비트 Atmega128 프로세서에서 더욱 효율적인 것을 알아낼 수 있었다[4]. 이때 최적의 윈도우 크기는 4였다. 비록 크기 4의 윈도우는 15개의 구성요소에 대하여 사전계산을 해야 하지만 (0번째 요소는 제외), 이를 통하여 유한체 곱셈의 연산은 현저히 빨라질 수 있다. Algorithm 2는 8 비트 워드 크기를 사용하는 환경에서 윈도우 기법 ($w=4$)을 적용한 left-to-right comb 방법을 기술하고 있다. 워드 크기가

Algorithm 2. Left-to-right comb method using window width $w=4$

---

1. **INPUT** : $a(z)$ and $b(z)$ in $GF(2^m)$
2. **OUTPUT** : $c(z)=a(z)\cdot b(z)$

---

3. Compute $T_u=u(z)\cdot b(z)$ for all polynomials $u(z)$ of degree at most $w-1$.
4. $C\leftarrow 0$.
5. for $j\leftarrow 0$ to $t-1$ do
6. $u\leftarrow U(a[j])$, $C\{j\}\oplus T_u$.
7. end for
8. $C\leftarrow C\cdot z^w$.
9. for $j\leftarrow 0$ to $t-1$ do
10. $u\leftarrow L(a[j])$, $C\{j\}\oplus T_u$.
11. end for
12. Return (C)

---

8 비트이고 윈도우의 크기가 4이기 때문에, $u\leftarrow U(a[j])$와 $u\leftarrow L(a[j])$는 $u\leftarrow(a[j]\gg 4)$, $u\leftarrow(a[j]\&0x0F)$와 같이 효율적으로 계산될 수 있다. Algorithm 2의 과정 6과 10의 partial XOR 곱셈인 $C\{j\}\oplus T_u$는 실제로 for-loop로 구현된다. 즉, $C\{j\}\oplus T_u$의 실제 코드는 "for $(i=0;i\le t;i{+}{+})$ $C[i+j]\leftarrow C[i+j]\oplus T_u[i]$;"이다[5] (LOAD된 $C[i+j]$와 $T_u[i]$ 값에 대하여 XOR연산이 수행된 후 다시 $C[i+j]$에 STORE된다). 따라서 워드의 크기가 작아질수록 더 많은 수의 메모리 접근 명령이 필요하다. 예를 들어 $GF(2^{163})$ 상에서 $W=32$와 $W=8$인 경우 각각의 LOAD와 STORE 명령어를 비교해보자. $W=32$의 경우는 $\lceil 163/32\rceil=6$이기 때문에 14번의 LOAD와 7번의 STORE 연산을 필요로 하는 반면 $W=8$인 경우는 $\lceil 163/8\rceil=21$이므로 44번의 LOAD와 22번의 STORE 연산을 요구한다.

4.2.3 모듈러 감산

$GF(2^m)$상의 곱셈과 제곱의 결과는 기약다항식 f로 감산된다. FIPS 186-2 표준에서 NIST가 권고한 빠른 모

---

3) 4비트에 대한 lookup table이므로 15바이트 크기의 table이 사용된다.

4) right-to-left comb 방법이 윈도우의 크기가 1인 left-to-right comb 방법에 비하여 조금 더 빠르긴 하지만 윈도우 방법을 적용할 수 없다는 단점이 있다.

5) Algorithm 2의 과정 3에서 만들어지는 Tu는 t개의 워드로 구성된 $b(z)$와 $u(z)$의 곱의 결과이기 때문에 $t+1$개의 워드로 구성된다. 따라서 $C[i+j]\leftarrow C[i+j]\oplus T_u[i]$ 연산이 $0\le i\le t$의 범위에서 수행되어야 한다.

Algorithm 3. Fast reduction modulo $f(z)=z^{163}+z^7+z^6+z^3+1$

1. **INPUT** : A binary polynomial $c(z)$ of degree at most 324
2. **OUTPUT** : $c(z) \bmod f(z)$
3. **for** $i \leftarrow 41$ **to** 21 **do**
4. $T \leftarrow C[i]$.
5. $C[i-21] \leftarrow C[i-21] \oplus (T \ll 5)$.
6. $C[i-20] \leftarrow C[i-20] \oplus (T \ll 4) \oplus (T \ll 3) \oplus T \oplus (T \gg 3)$.
7. $C[i-19] \leftarrow C[i-19] \oplus (T \gg 4) \oplus (T \gg 5)$.
8. end for
9. $T \leftarrow C[20] \gg 3$.
10. $C[0] \leftarrow C[0] \oplus (T \ll 7) \oplus (T \ll 6) \oplus (T \ll 3) \oplus T$.
11. $C[1] \leftarrow C[1] \oplus (T \gg 1) \oplus (T \gg 2)$.
12. $C[20] \leftarrow C[20] \& 0x07$.
13. Return $(C[20], \ldots, C[2], C[1], C[0])$

들로 감산을 위한 감산 다항식이 존재한다[3]. 이러한 다항식들은 3차항 (Trinomial) 또는 5차항 (Pen- tanomial)이기 때문에 $c(z) \bmod f$는 효율적으로 계산될 수 있다. Algorithm 3은 곱셈과 제곱의 결과를 $GF(2^{163})$의 원소로 감산시킨다. 앞에서 언급한 유한체 곱셈 알고리즘과 마찬가지로 Algorithm 3은 8 비트 워드기반이기 때문에 많은 수의 메모리 접근 연산이 연관되어 있다.

4.3 좌표계 선택

8 비트 워드의 ATmega128에서 $GF(2^m)$상의 곱셈 연산에 대한 역원 연산의 비율을 계산해 본 결과 24.99의 결과를 얻었다 ($M: I = 1:24.99$). 따라서 스칼라 곱셈 과정에서 가능한 한 역원 연산을 제거하는 것이 성능을 위하여 바람직하다. 따라서 TinyECCK를 구현할 때 아핀 좌표계 대신 López-Dahab (LD)좌표계를 적용하였다. [표 1]은 TinyECCK에서의 적절한 좌표 선택을 뒷받침해준다. ECADD 연산에 관련된 두 개의 점이 서로 다른 좌표계 (P_1 : LD 좌표계, P_2 : 핀 좌표계)로 표현되어 더해지는 것이 두 점이 서로 같은 좌표계로 표현되었을 때보다 더욱 효율적으로 계산될 수 있다

[표 1] ATmega128 프로세서에서 $GF(2^{163})$상의 유한체 연산들의 계산 시간 비교 (곱셈과 제곱의 시간은 감산 시간까지 포함하고 있으며 모든 시간은 초로 측정되었다.)

유한체 연산	계산 시간	역원/해당연산들
곱셈 연산	0.00292224	24.99
제곱 연산	0.00036982	197.47
역원 연산	0.07302550	1

[12]. 따라서 사전계산 테이블에 저장되는 점들은 아핀 좌표계로 표현하였으며, 실제 ECADD와 ECDBL 과정에서는 혼합좌표계를 사용하였다.

4.4 Width-w NAF (Non-Adjacent Form)와 Width-w TNAF(Tau-adic Non-Adjacent Form)

$GF(2^m)$ 위의 한 점 $p=(x,y)$의 역원은 $-p=(x, x+y)$가 된다. 이와 같이 $E(GF(2^m))$의 원소의 역원은 적은 비용의 연산으로 계산될 수 있으며, 두 점의 뺄셈은 덧셈과 같은 방법으로 계산될 수 있기 때문에 스칼라 곱셈과정에서 부호화된 이진 표현 ($k=\sum_{i=0}^{l-1} k_i 2^i, l=\log_2 k$, $k_i \in \{0, \pm 1\}$)을 쉽게 적용할 수 있다. Non-adjacent form (NAF)은 모든 부호화된 이진 표현 중에서 최적의 non-zero 밀도 (1/3)를 제공한다. NAF를 이용한 스칼라 곱셈은 $l \cdot ECDBL + l/3 \cdot ECADD$의 연산으로 계산될 수 있다 (비교 : 이진표현을 사용할 경우 $l \cdot ECDBL + l/2 \cdot ECADD$가 필요하다). 만약 추가적인 메모리가 사용가능하다면 스칼라 k를 한번에 w 비트씩 처리할 수 있는 슬라이딩 윈도우 방법을 적용하여 스칼라 곱셈의 연산 시간을 더욱 줄일 수 있다. w크기의 윈도우를 사용하는 width-w NAF (wNAF)는 $2^{w-2}-1$개의 사전계산 점들을 담는 사전계산 테이블을 이용하여 $1/(w+1)$의 nonzero 밀도를 제공한다. 따라서 wNAF를 적용한 스칼라 곱셈은 $l \cdot ECDBL + 1/(w+1) \cdot ECADD$의 연산으로 계산된다. Micaz 센서모트에서는 128 킬로 바이트의 ROM 메모리와 4 킬로 바이트의 RAM 메모리가 사용가능하기 때문에 Tiny ECCK에 wNAF를 충분히 적용할 수 있었다. Tiny ECCK의 구현은 sect163k1에 기반을 두었기 때문에 스칼라 곱셈에 width-w tau-adic non-adjacent form (wTNAF)[11]을 적용할 수 있다. 식 (2)가 $a=0,1$ 그리고 $b=0$ 인 경우, Koblitz 커브라고 불리며 이것의 큰 장점은 스칼라 곱셈에서 ECDBL연산을 몇 개의 간단한 유한체 제곱 연산으로 대체시킬 수 있다는 것이다. 따라서 wTNAF를 적용한 스칼라 곱셈은 $l/(w+1) \cdot ECADD$ 만으로 계산될 수 있다. 하지만 wTNAF 리코딩 알고리즘은 추가적인 부분감산 함수 (Partial reduction modulo function)와 라운딩 오프 (Rounding off)[11] 프로시저를 필요로 하기 때문에 wTNAF를 구현하는 것은 wNAF에 비하여 더 많은 코드를 요구한다. TinyECCK에 wNAF와 wTNAF를 적용하였을 때 TinyECCK의 코드 크기는 각각 10870 바

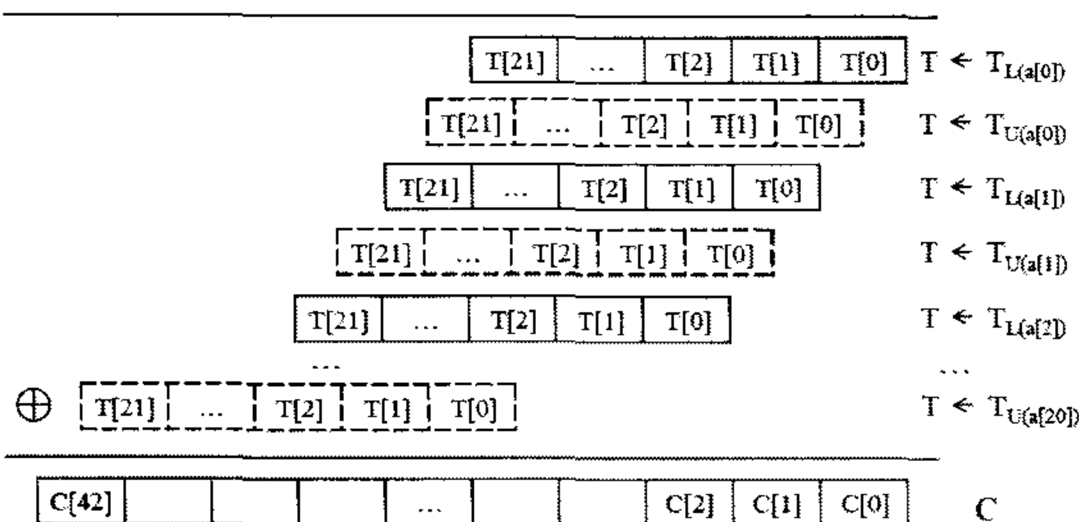
이트와 13748 바이트였다. 이것은 Micaz 모트의 사용 가능한 ROM 메모리 크기 (128 킬로 바이트)의 단지 8.3%와 10.5% 였다. 실험을 통하여 Micaz모트에서 최적의 윈도우 크기는 4임을 알아냈다.

V. 성능 개선을 위한 제안 테크닉들

4. 2절에서 제시한 곱셈 알고리즘과 감산 알고리즘의 성능은 중복된 메모리 접근을 줄임으로써 더욱 향상될 수 있다. 메모리 접근 연산은 알고리즘의 전체 연산 시간에서 많은 비중을 차지하기 때문에 중복된 메모리 접근을 줄임으로써 알고리즘의 성능을 향상시킬 수 있는 것이다.

5.1 유한체 곱셈 과정에서 중복된 메모리 접근을 줄이기 위한 방법

$GF(2^m)$ 상에서의 XOR 곱셈 연산은 많은 수의 중복된 메모리 접근 연산과 연관되어있다. 이 때문에 일반적으로 $GF(2^m)$ 상에서의 곱셈 연산이 $GF(p)$ 상에서의 곱셈 연산보다 비효율적이다. [그림 1]은 Algorithm 2의 곱셈 연산의 계산과정을 보이고 있다. 그림에서 홀수 행들은 Algorithm 2의 두 번째 for-loop (과정 9~11)이고 짝수 행들은 첫 번째 for-loop(과정 5~7)이다. 결과적으로 모든 행들은 마지막 결과인 C를 계산하기 위하여 해당 위치에서 서로 XOR된다 (Partial XOR 곱셈). Algorithm 2에서 첫 번째 for-loop의 결과들은 왼쪽으로 윈도우 크기인 w만큼 쉬프트 되는데 이는 [그림 1]에서 잘 나타나있다. Algorithm 2의 각 for-loop에서 $b(z)$에 관한 사전계산 테이블에 접근하여 해당 요소를 가져오기 위하여 $L(a[j])$와 $U(a[j])$가 계산된다. 그 후에 테이블에서 해당 값이 로드되고 이는 다시 C와 j 위치



[그림 1] 윈도우 방법을 적용한 left-to-right comb 방법의 계산 과정

에서부터 $j+N$ 위치까지 XOR된다 ($N=t$). [그림 1]을 살펴보면 Algorithm 2의 XOR 곱셈이 많은 수의 중복된 메모리 접근 연산과 연관되어 있음을 알 수 있다. 다음 예제는 이것을 확인시켜준다 (설명의 간결함을 위하여 두 번째 for-loop과정만을 고려하였다).

$C_0 \leftarrow C_0 \oplus T_0;$
// C_0 : 1회 LOAD, T_0 : 1회 LOAD, STORE : 1회
$C_1 \leftarrow C_1 \oplus T_0, C_1 \leftarrow C_1 \oplus T_1;$
// C_1 : 2회 LOAD, T_0 : 1회 LOAD, T_1 : 1회 LOAD, STORE : 2회
$C_2 \leftarrow C_2 \oplus T_0, C_2 \leftarrow C_2 \oplus T_1, C_2 \leftarrow C_2 \oplus T_2;$
// C_2 : 3회 LOAD, T_0 : 1회 LOAD, T_1 : 1회 LOAD, T_2 : 1회 LOAD, STORE : 3회
$C_3 \leftarrow C_3 \oplus T_0, C_3 \leftarrow C_3 \oplus T_1, C_3 \leftarrow C_3 \oplus T_2, C_3 \leftarrow C_3 \oplus T_3;$
// C_3 : 4회 LOAD, T_0 : 1회 LOAD, T_1 : 1회 LOAD, T_2 : 1회 LOAD, T_3 : 1회 LOAD, STORE : 4회...

C_1, C_2, C_3의 계산은 각각 두 번, 세 번, 네 번의 STORE 명령이 연관되었으며 이것은 중복된 것이다. 또한 각 과정에서 C_i의 값이 $i+1$번 LOAD되고 있음을 알 수 있다 ($i \le 20$까지, $20 < i \le 41$ 범위에서는 C_i 값이 $42-i$번 LOAD된다). 따라서 Algorithm 2의 성능은 중복된 STORE 연산의 수와 C_i의 LOAD 횟수를 줄임으로써 향상될 수 있다. 제안하는 방법은 몇 개의 연속한 XOR 곱셈을 하나로 합침으로써 XOR 곱셈 ($C\{j\} \oplus T_u$)에 연관된 STORE와 LOAD 연산의 수를 줄이는 것이다[6].

예를 들어, $C\{0\} \oplus T_{U(a[0])}$과 $C\{1\} \oplus T_{U(a[1])}$은 "$C\{1\} \leftarrow C\{1\} \oplus T_{U(a[0])}\{N,...,1\} \oplus T_{U(a[1])}[N-1,...,0]$, $C[0] \leftarrow \leftarrow C[0] \oplus T_{U(a[0])}[0], C[N+1] \leftarrow C[N+1] \oplus T_{U(a[1])}[N]$" 로 통합할 수 있다. 이 전략은 Algorithm 4로 일반화된다. 실제로 더 많은 수의 XOR 곱셈을 통합한다면 더 많은 수의 중복 메모리 접근을 줄일 수 있으나 더 많은 양의 코드를 필요로 한다. TinyECCK의 경우에는 코드의 크기를 고려하여 두 개의 연속한 XOR 곱셈을 하나로 통합하였기 때문에 for-loop의 카운터가 2씩 증가한다. Algorithm 4를 이용하여 절약되는 STORE 연산의 수

6) 곱셈과정에서 XOR 곱셈의 수를 줄이는 것이 아니라 XOR 곱셈과정에서 발생하는 중복된 메모리 접근 연산 (STORE와 LOAD)의 수를 줄이는 것이다.

[표 2] 제안 방법과 기본의 방법의 성능 비교 (제안 방법을 통하여 절약되는 메모리 접근 연산의 수를 제시하기 위하여 각 알고리즘의 for-loop안에서의 연산수를 계산하여 비교하였다. 모든 시간은 초로 측정되었다.)

유한체 연산		기존 알고리즘	제안 방법을 적용한 알고리즘	향상 비율 (%)
곱셈 연산	시간	0.00370277	0.00292224	21.08
	연산	1,890LOAD+966STORE+924XOR	1,430LOAD+506STORE+924XOR	460LOAD+460STORE 절약
감산 연산	시간	0.00034239	0.00025801	24.68
	연산	80LOAD+80STORE+140XOR	50LOAD+50STORE+140XOR	30LOAD+30STORE 절약

Algorithm 4. Proposed left-to-right comb method using window width $w=4$

1. **INPUT** : $a(z)$ and $b(z)$ in $GF(2^m)$
2. **OUTPUT** : $c(z)=a(z)\cdot b(z)$

3. Compute $T_u=u(z)\cdot b(z)$ for all polynomials $u(z)$ of degree at most $w-1$.
4. $C\leftarrow 0$.
5. for $j\leftarrow 0$ to $t-1$ increments j by 2 do
6. $u_1\leftarrow U(a[j]), u_2\leftarrow U(a[j+1])$.
7. $C[j]\leftarrow C[j]\oplus T_{u_1}[0]$,
   $C[j+N+1]\leftarrow C[j+N+1]\oplus T_{u_2}[N]$,
   $C\{j+1\}\oplus T_{u_1}\{N,...,1\}\oplus T_{u_2}\{N-1,...,0\}$.
8. end for
9. $C\leftarrow C\cdot z^w$.
10. for $j\leftarrow 0$ to $t-1$ increments j by 2 do
11. $u_1\leftarrow L(a[j]), u_2\leftarrow L(a[j+1])$.
12. $C[j]\leftarrow C[j]\oplus T_{u_1}[0]$,
    $C[j+N+1]\leftarrow C[j+N+1]\oplus T_{u_2}[N]$,
    $C\{j+1\}\oplus T_{u_1}\{N,...,1\}\oplus T_{u_2}\{N-1,...,0\}$.
13. end for
14. Return (C)

를 계산할 수 있다. Algorithm 2에서는 for-loop의 카운터가 0에서부터 $t-1$까지 증가하며 또한 $C\{j\}\oplus T_u$도 for-loop이기 때문에 첫 번째와 두 번째 for-loop 과정에서 필요한 STORE 연산의 수는 $2t^2+4t$이 된다. Algorithm 4의 STORE 명령의 수를 계산하면 t^2+3t+2가 된다. 따라서 t^2+t-2만큼의 STORE 연산이 절약된다. 또한 LOAD 연산의 경우 Algorithm 2는 $4t^2+6t$번의 LOAD 연산이 연관된 반면 Algorithm 4는 $3t^2+5t+2$번의 LOAD 연산을 사용한다. 따라서 약 t^2+t-2만큼의 LOAD 연산을 절약할 수 있다. [표 2]를 통하여 Algorithm 4가 Algorithm 2에 비하여 약 21.1% 빨라진 것을 확인할 수 있다. 또한 $t=21$일 때, Algorithm 4가 Algorithm 2에 비하여 460번의 LOAD와 STORE 연산을 절약하는 것을 확인할 수 있다.

5.2 모듈러 감산 과정에서 중복된 메모리 접근을 줄이기 위한 방법

Algorithm 3 역시 중복된 메모리 접근 명령과 관련되어 있다. Algorithm 3의 동작 과정에서 카운터 i가 30에서 27로 감소하는 경우를 살펴보자. 카운터 i가 30에서 27일 동안 연산 과정은 아래와 같다.

1. $T\leftarrow C[30]$;
2. $C[9]\leftarrow C[9]\oplus(T\ll 5)$;
3. $C[10]\leftarrow C[10]\oplus(T\ll 4)\oplus(T\ll 3)\oplus T\oplus(T\gg 3)$;
4. $C[11]\leftarrow C[11]\oplus(T\gg 4)\oplus(T\gg 5)$;
5. $T\leftarrow C[29]$;
6. $C[8]\leftarrow C[8]\oplus(T\ll 5)$;
7. $C[9]\leftarrow C[9]\oplus(T\ll 4)\oplus(T\ll 3)\oplus T\oplus(T\gg 3)$;
8. $C[10]\leftarrow C[10]\oplus(T\gg 4)\oplus(T\gg 5)$;
9. $T\leftarrow C[28]$;
10. $C[7]\leftarrow C[7]\oplus(T\ll 5)$;
11. $C[8]\leftarrow C[8]\oplus(T\ll 4)\oplus(T\ll 3)\oplus T\oplus(T\gg 3)$;
12. $C[9]\leftarrow C[9]\oplus(T\gg 4)\oplus(T\gg 5)$;
13. $T\leftarrow C[27]$;
14. $C[6]\leftarrow C[6]\oplus(T\ll 5)$;
15. $C[7]\leftarrow C[7]\oplus(T\ll 4)\oplus(T\ll 3)\oplus T\oplus(T\gg 3)$;
16. $C[8]\leftarrow C[8]\oplus(T\gg 4)\oplus(T\gg 5)$;

Algorithm 3은 $C[30], C[29], C[28], C[27]$을 계산하기 위하여 16번의 LOAD와 STORE 명령을 사용한다. 하지만 위와 같이 중복된 LOAD와 STORE 명령을 제거하기 위하여 같은 곳에 저장되는 중간 값들을 함께 XOR시켜 한 번에 저장하는 전략을 사용할 수 있다. 이 전략을 적용하여 아래와 같이 계산할 수 있다.

1. $T_1\leftarrow C[30]$, $T_2\leftarrow C[29]$, $T_3\leftarrow C[28]$, $T_4\leftarrow C[27]$;
2. $C[6]\leftarrow C[6]\oplus(T_4\ll 5)$;
3. $C[7]\leftarrow C[7]\oplus(T_4\ll 4)\oplus(T_4\ll 3)\oplus T_4$
   $\oplus(T_4\gg 3)\oplus(T_3\ll 5)$;
4. $C[8]\leftarrow C[8]\oplus(T_4\gg 4)\oplus(T_4\gg 5)\oplus(T_3\ll 4)$
   $\oplus(T_3\ll 3)\oplus T_3\oplus(T_3\gg 3)\oplus(T_2\ll 5)$;
5. $C[9]\leftarrow C[9]\oplus(T_3\gg 4)\oplus(T_3\gg 5)\oplus(T_2\ll 4)$
   $\oplus(T_2\ll 3)\oplus T_2\oplus(T_2\gg 3)\oplus(T_1\ll 5)$;
6. $C[10]\leftarrow C[10]\oplus(T_2\gg 4)\oplus(T_2\gg 5)\oplus(T_1\ll 4)$
   $\oplus(T_1\ll 3)\oplus T_1\oplus(T_1\gg 3)$;
7. $C[11]\leftarrow C[11]\oplus(T$

〔표 3〕 제안한 방법을 TinyECCK에 적용하였을 때의 향상된 성능 (TinyECCK는 sect163k1을 이용하였고, 알고리즘 2와 3을 적용하였을 때와 알고리즘 5, 6을 적용하였을 때의 수행 시간을 비교하였다. 모든 시간은 초로 측정되었다.)

		Binary	2NAF	3NAF	4NAF	2TNAF	3TNAF	4TNAF
기존 방법	Sign	5.4900	4.3061	3.8897	3.5750	2.2003	1.8770	1.6111
	Verify	7.5789	5.9157	5.0642	4.5383	4.2179	3.3391	2.7815
제안방법 적용	Sign	4.4036	3.4718	3.1430	2.8949	1.8287	1.5745	1.3613
	Verify	6.0825	4.7473	4.0750	3.6590	3.4499	2.7425	2.3116
Saving (%)	Sign	19.78	19.37	19.19	19.02	16.88	16.11	15.50
	Verify	19.74	19.74	19.53	19.37	18.20	17.86	16.89

Algorithm 5. Proposed fast reduction modulo $f(z)=z^{163}+z^7+z^6+z^3+1$

1. INPUT : A binary polynomial $c(z)$ of degree at most 324
2. OUTPUT : $c(z) \bmod f(z)$

3. for $i \leftarrow 41$ to 21 decrements i by 4 do
4. $T_1 \leftarrow C[i], T_2 \leftarrow C[i-1], T_3 \leftarrow C[i-2],$ $T_4 \leftarrow C[i-3]$.
5. $C[i-24] \leftarrow C[i-24] \oplus (T_4 \ll 5)$
6. $C[i-23] \leftarrow C[i-23] \oplus (T_3 \ll 5) \oplus (T_4 \ll 4) \oplus (T_4 \ll 3) \oplus T_4 \oplus (T_4 \gg 3)$.
7. $C[i-22] \leftarrow C[i-22] \oplus (T_2 \ll 5) \oplus (T_3 \ll 4) \oplus (T_3 \ll 3) \oplus T_3 \oplus (T_3 \gg 3) \oplus (T_4 \gg 4) \oplus (T_4 \gg 5)$
8. $C[i-21] \leftarrow C[i-21] \oplus (T_1 \ll 5) \oplus (T_2 \ll 4) \oplus (T_2 \ll 3) \oplus T_2 \oplus (T_2 \gg 3) \oplus (T_3 \gg 4) \oplus (T_3 \gg 5)$.
9. $C[i-20] \leftarrow C[i-20] \oplus (T_1 \ll 4) \oplus (T_1 \ll 3) \oplus T_1 \oplus (T_1 \gg 3) \oplus (T_2 \gg 4) \oplus (T_2 \gg 5)$
10. $C[i-19] \leftarrow C[i-19] \oplus (T_1 \gg 4) \oplus (T_1 \gg 5)$
11. end for
12. $T_1 \leftarrow C[21], T_2 \leftarrow C[20]$.
13. $C[0] \leftarrow C[0] \oplus (T_1 \ll 5) \oplus (T_2 \ll 4) \oplus (T_2 \ll 3) \oplus T_2 \oplus (T_2 \gg 3)$
14. $C[1] \leftarrow C[1] \oplus (T_1 \ll 4) \oplus (T_1 \ll 3) \oplus T_1 \oplus (T_1 \gg 3) \oplus (T_2 \gg 4) \oplus (T_2 \gg 5)$.
15. $C[2] \leftarrow C[2] \oplus (T_1 \gg 4) \oplus (T_1 \gg 5)$.
16. $C[20] \leftarrow C[20] \,\&\, 0x07$.
17. Return $(C[20], \ldots, C[2], C[1], C[0])$

위와 같이 LOAD와 STORE 명령의 빈도를 16번에서 10번으로 줄일 수 있다. Algorithm 3은 for-loop안에서 20 • 4 = 80번의 LOAD와 STORE 명령을 사용한다. 이에 반하여 제안한 기법은 5 • 10 = 50번의 LOAD와 STORE 명령만을 이용하여 30번의 LOAD와 STORE 명령을 줄일 수 있다. 이 전략을 일반화한 것이 Algorithm 5이며, 이곳에서는 4번의 for-loop의 실행을 하나로 통합하였다. [표 2]가 제시한 것처럼 Algorithm 5은 Algorithm 3보다 약 24.7% 빠르게 감산 연산을 수행한다. 더 많은 for-loop의 실행을 통합하며 더 많은 수의 메모리 접근 명령의 수를 줄일 수 있지만, 코드의 크기가 증가한다는 단점이 있다. 따라서 최적의 통합 정도를 찾는 것이 필요하다. 실험을 통하여 4번의 for-loop 실행을 통합하는 것이 메모리와 성능을 고려할 때 최적인 것을 알아냈다. [표 3]은 5. 1절과 5. 2절에서 제안한 방법을 TinyECCK를 구현하는데 적용한 결과, 향상된 성능을 제시하고 있다. Algorithm 4와 6를 적용하여 ECDSA의 sign과 verify의 연산 시간을 약 15~19% 정도 단축시킬 수 있었다. wTNAF를 사용하였을 때보다 wNAF를 이용하였을 때의 성능향상이 큰 것을 확인할 수 있는데 이것은 wTNAF에서는 ECDBL 연산이 몇 개의 유한체 제곱 연산으로 대체되기 때문이다.

VI. 실험적인 결과와 분석

본 절에서는 TinyECCK의 성능을 동작 시간, 메모리 사용량 및 지원하는 서비스 측면에서 분석하고, 센서 모트 상에서 기존의 소프트웨어 구현들과 성능을 비교한다.

6.1 유한체 연산의 분석

센서모트 상에서 $GF(2^m)$ 위의 유한체 곱셈의 성능이 $GF(p)$ 상의 곱셈의 성능보다 빠를 수 있음을 보이기 위하여 TinyECCK와 TinyECC[1]를 비교한다. Tiny ECC는 불필요한 메모리 접근을 줄이기 위하여 추가적인 레지스터를 사용하는 혼합 곱셈/제곱 방법 (hybrid multiplication/squaring)과 pseudo-Mersenne prime을 사용하는 최적화된 모듈로 감산을 적용하였기 때문에 윈도우를 사용하는 left-to-right comb 방법 (Algorithm 4)

〔표 4〕 TinyECCK와 TinyECC의 유한체 연산의 비교 (공평한 비교를 위하여, C버전의 TinyECC와 비교하였으며 모든 시간은 초로 측정되었다.)

유한체 연산	TinyECC	TinyECCK	향상 비율 (%)
곱셈 연산	0.00315647	0.00292224	7.42
제곱 연산	0.00314779	0.00036982	88.25
역원 연산	0.14856858	0.07302550	50.84

과 빠른 감산 연산 (Algorithm 5)을 적용한 TinyECCK와 비교하는 것은 적당하다. [표 4]는 TinyECCK의 곱셈 연산이 TinyECC의 곱셈 연산보다더 빠름을 보여 준다 (곱셈 연산과 제곱 연산의 시간은 감산 연산의 시간까지 포함하고 있다). 실제로 TinyECCK의 곱셈 연산이 Algorithm 2를 이용하였을 때는 TinyECC의 곱셈 연산보다 느렸으나 Algorithm 4를 적용하여 TinyECC의 곱셈 연산보다 더욱 빨라질 수 있었다 ([표 2] 참조). TinyECCK의 역원 연산이 TinyECC의 역원 연산보다 더 빠른 점을 차치하더라도 TinyECCK의 제곱 연산은 TinyECC에서의 연산보다 더욱 효율적으로 계산된다.

6.2 TinyECCK와 TinyECC의 코드 크기 비교

TinyECCK가 스칼라 곱셈 연산과 ECDSA 서비스를 지원하기 위하여 $GF(2^m)$과 $GF(p)$에서의 연산을 구현하였음에도 불구하고 TinyECC보다 더 작은 코드 크기를 사용한다. [표 5]는 동등한 연산을 하였을 때 TinyECCK와 TinyECC의 계산시간과 코드 크기를 비교한다. 비록 C로만 구현한 TinyECC의 경우 TinyECCK와 비교하여 코드 크기가 조금 더 크고, 성능은 현저히 떨어졌다. 계산 시간을 단축시키기 위하여 곱셈/제곱, 감산 등 성능의 핵심 부분을 inline assembly로 구현하였으나 TinyECCK와 비교하여 코드 크기와 계산 시간 측면에서 뒤떨어진다. 스칼라 곱셈과 ECDSA 연산에서 TinyECCK의 코드 크기가 C로 구현된 TinyECC보다 더 작을 뿐만 아니라 계산 시간 측면에서도 inline assembly를 이용한 TinyECC보다 더욱 뛰어났다. [표 5]를 통하여 TinyECCK가 TinyECC와 비교하여 성능과 코드 크기에서 더욱 효율적임을 알 수 있다. TinyECCK가 TinyECC와 비교하여 코드의 크기가 더 작을 수 있는 가장 큰 이유는 C로만 구현하였음에도 불구하고 assembly나 inline assembly로 구현된 것들보다 뛰어난 성능을 제공하기 때문이다. 이 결과를 통하여 $GF(2^m)$에서 최적화된 ECDSA구현의 코드 크기는 $GF(p)$ 상의 ECDSA구현의 코드 크기보다 더 작을 수 있다는 것을 확인할 수 있다.

6.3 기존의 구현들과 성능 비교

[표 6]은 지금까지 센서 모트 상에서 구현된 타원곡선 암호 소프트웨어들의 성능을 계산 시간, 코드 크기, 지원하는 프로토콜 등의 측면에서 TinyECCK와 비교하고 있다. 기존의 $GF(2^m)$상의 구현들[4,7,9,10]의 성능이 $GF(p)$상에서 구현된 것들[1,8] 보다 뒤떨어지는 것을 확인할 수 있다. 비록 [10]에서는 타원곡선 암호를 완전히 어셈블리언어로 구현 하였지만 핵심 부분에만 인라인 어셈블리언어를 사용한 TinyECC[1]에 비하여 그 성능이 뒤쳐진다. [4], [9]는 도메인 파라미터로 sect163k1을 사용하였지만 스칼라 곱셈 과정에 TNAF 방법을 구현하지 않았기 때문에 Koblitz 커브의 장점을 충분히 활용하지 못하였다. 어셈블리 언어로 구현된

〔표 5〕 TinyECCK와 TinyECC의 성능 및 코드 크기 비교 (시간과 코드 크기는 각각 초와 바이트로 측정되었다.)

		TinyECCK	TinyECC (C만 이용)	TinyECC (C, inline asm이용)
스칼라곱셈	시간	1.1411	6.1418	1.8825
	코드크기	5,592	8,528	10,092
Sign	시간	1.3607	6.2694	2.0016
	코드크기	12,084	13,192	16,478
Verify	시간	2.3237	7.9208	2.43184
	코드크기	13,748	15,872	19,308

〔표 6〕 센서 모트 상에서 구현된 타원곡선 암호 소프트웨어들의 성능 비교 (w 는 사용된 윈도우의 크기를 의미하며 init 은 사전 계산 테이블을 구성하는데 걸린 시간이다. 모든 시간은 초로 측정되었다)

	[4]	[8]	[9]	[7]	[10]	[1]	TinyECCK (스칼라곱셈만 구현)		TinyECCK (ECDSA 포함)
구현 언어	C	C, inline assembly	C	C	assembly	C, inline assembly	C		
플랫폼	Mica2 (8 비트 ATmega128)	TelsoB (16 비트 MSP430)	8 비트 ATmega128	Mica2 (8 비트 ATmega128)	8 비트 ATmega128	Micaz (8 비트 ATmega128)	Micaz (8 비트 ATmega128)		
도메인 파라미터	sect163k1	secp160r1	sect163k1	sect113r1	sect163r1	secp160r1	sect163k1		
기반 유한체	Binary	Prime	Binary	Binary	Binary	Prime	Binary		
ROM 크기 (바이트)	34,342	17,823	11,592	75,088	8,767	19,308	5,592		13,748
RAM 크기 (바이트)	1,055	1,638	820	208	239	1,510	330 (w=2)	618 (w=4)	1,004
스칼라 곱셈	34.161	3.13	13.9	6.74	4.14	1.8825	1.6852 (init : 0.0005)	1.1370 (init : 0.2515)	1.1411 (init : 0.2514)
Sign	-	3.35	-	6.88	-	2.0016 (init : 1.83)	-		1.3613 (init : 0.2515)
Verify	-	6.78	-	24.17	-	2.43184 (init : 3.49)	-		2.3116 (init : 0.2449)
지원하는 프로토콜	스칼라 곱셈, ECDH	스칼라 곱셈, ECDSA	스칼라 곱셈, ECDH	스칼라 곱셈, ECDSA, ElGamal	스칼라 곱셈	스칼라 곱셈, ECDSA[7]	스칼라 곱셈		스칼라 곱셈, ECDSA

[10]보다 C언어로만 구현된 TinyECCK의 성능이 더 뛰어난 이유는 TinyECCK에 Algorithm 4와 6 그리고 wTNAF 기반의 스칼라 곱셈이 적용되었기 때문이다. 표에서 확인할 수 있듯이 TinyECCK는 동작 시간, 사용된 ROM, RAM의 크기 측면에서 기존의 구현들에 비하여 우수한 성능을 제공한다. TinyECCK에서 스칼라 곱셈을 계산하는 모듈의 코드 크기는 5,592 바이트이며 2TNAF와 4TNAF가 적용될 때 각각 330 바이트와 618 바이트의 RAM을요구한다. 더욱이 이것의 성능은 지금까지 C 또는 C와 inline assembly를 이용하여 구현된 타원곡선의 소프트웨어 구현 중에서 가장 뛰어나다. 비록 서명 생성, 검증 등의 추가적인 구현으로 TinyECCK의 ECDSA 모듈의 코드 크기는 13,748 바이트로 증가하였지만 여전히 19,308 바이트를 이용하는 TinyECC와 비교하여 더 적은 메모리를 사용한다. 뿐만 아니라 TinyECCK는 TinyECC와 비교하여 사전 계산 테이블과 도메인 파라미터를 초기화하는데 더욱 적은 시간을 소비한다. TinyECCK에 4TNAF가 적용되었을 때 사전 계산 테이블을 구성하는데 0.2515 초 가 걸린 반면, TinyECC는 4-ary window 방법의 사전 계산 테이블을 초기화하는데 1.83 초가 걸렸다. TinyECCK를 이용하여 두 센서 모트는 1.14 초 안에 공통키를 계산할 수 있으며, 또한 1.37 초와 2.32 초 안에 각각 하나의 서명을 생성하고 검증할 수 있다. 지원하는 프로토콜 측면에서, TinyECCK는 $GF(2^m)$상의 모든 타원곡선 연산 (ECADD, ECDBL, 스칼라 곱셈)을 제공하며 ECDSA의 서비스도 제공한다.

Ⅶ. 결론 및 향후 연구

본 논문에서는 $GF(2^m)$ 상에서의 곱셈, 감산 연산의 비효율성이 중복된 메모리 접근에 기인한다는 것을 보였으며 이러한 불필요한 메모리 접근을 줄일 수 있는 방법을 제시하였다. 제안 방법을 통하여 $GF(2^{163})$상에서의 곱셈 연산과 감산 연산의 계산 시간을 각각 21.1%, 24.7%만큼 절약할 수 있었다. 제안된 곱셈과 감산 연산을 통하여 ECDSA의 sign과 verify의 계산 시간을 15~19%정도 줄일 수 있었다. 뿐만 아니라 이때의 향상된 곱셈 연산의 성능은 sect160r1상에서 혼합 곱셈

7) 2007년 11월 2일자로 TinyECC의 버전이 0.3에서 1.0으로 업데이트 되었다. 1.0에서는 추가적으로 ECDH, ECIES 프로토콜을 지원한다. 표 6에서 TinyECCK와의 비교를 위하여 TinyECC의 ECDSA 모듈의 성능과 메모리 사용량을 이용하였음을 밝힌다.

알고리즘을 이용하여 최적화된 연산보다 7.4% 빠르다.

제안된 방법을 이용하여 Micaz 센서 모트에서 동작하는 타원곡선 암호 라이브러리인 TinyECCK를 개발하였으며 센서 모트 상에서 구현된 기존의 타원곡선 암호 소프트웨어들과 그 성능을 비교하였다. 연산 시간, 코드 크기, 지원하는 프로토콜 측면에서 Tiny ECCK의 성능이 기존의 것들보다 더욱 우수함을 알 수 있었다.

이러한 실험 결과와 분석을 통하여 다음 두 가지의 결론을 얻을 수 있었다. 첫 번째, 작은 크기의 워드를 사용하는 저전력 장치에서 $GF(2^m)$ 상의 타원곡선 암호의 소프트웨어 구현이 비효율적이라는 기존의 생각과는 달리 $GF(2^m)$상에서 타원곡선 암호 소프트웨어를 구현하는 것이 $GF(p)$에 비하여 더욱 적당하다. 세심한 구현을 통하여 $GF(2^m)$상의 곱셈 연산이 $GF(p)$에서의 곱셈 연산보다 더 빠를 수 있었다. 두 번째는, 센서 네트워크를 안전하게 만들기 위해서 타원곡선 암호, 특히 TinyECCK의 사용은 충분히 가능하다는 것이다.

향후 연구로는 8 비트 Micaz 센서 모트에서 개발한 TinyECCK를 16 비트 환경의 telosB[17] 모트에 이식하는 것이다. 8 비트 워드를 기본으로 하여 구현한 Tiny ECCK와 16 비트 워드로 변환하여 구현한 TinyECCK를 각각 telosB 모트에 이식하여 그 성능을 비교할 것이다. 또한 telosB 모트에서 동작하는 TinyECCK의 성능을 기존에 telosB 모트에서 구현된 타원곡선 암호 소프트웨어와 비교할 것이다.

참고문헌

[1] A. Liu, P. Kampanakis, and P. Ning, "Tiny ECC : Elliptic Curve Cryptography for Sensor Networks (Version 1.0)", available at "http://discovery.csc.ncsu.edu/software/TinyECC/," November 2007.

[2] C. Karlof, N. Sastry, and D. Wagner, "TinySec : Link Layer Security Architecture for Wireless Sensor Networks", Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems (SenSys'04) pp.162-175, 2004.

[3] Certicom Research, "SEC 2 : Recommended Elliptic Curve Domain Parameters, Standards for Effcient Cryptography, Version 1.0", September 2000.

[4] 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.

[5] D. Hankerson, J. López, and A. Menezes, "Software Implementation of Elliptic Curve Cryptography over Binary Fields", Workshop on Cryptographic Hardware and Embedded Systems (CHES 2000), LNCS 1965, pp.1-24, 2000.

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

[7] E. O. Blaß and M. Zitterbart, "Towards Acceptable Public-Key Encryption in Sensor Networks", ACM 2nd International Workshop on Ubiquitous Computing, p. 88-93, INSTICC Press, Miami, USA, May 2005.

[8] H. Wang, B. Sheng and Q. Li, "Elliptic curve cryptography-based access control in sensor networks", International Journal of Security and Networks, Vol. 1, Nos. 3/4, 2006.

[9] H. Yan and Z. Shi, "Studying software implementations of elliptic curve cryptography", Third International Conference on Information Technology : New Generations (ITNG 2006), pp.78-83, 2006.

[10] H. Eberle, A. Wander, N. Gura, S. Chang-Shantz, and V. Gupta, "Architectural Extensions for Elliptic Curve Cryptography over $GF(2^m)$ on 8-bit Microprocessors", 16th International Conference on Application-Specific Systems, Architecture and Processors (ASAP 2005), Vol. 00, pp.343-349, 2005.

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

[12] J. López and R. Dahab, "Improved Algorithms for Elliptic Curve Arithmetic in $GF(2^m)$", Selected Areas in Cryptography (SAC'98),

LNCS 1556, pp. 201-212, 1999.
[13] J. López and R. Dahab, "High-Speed Software Multiplication in F_{2^m}", Progress in Cryptology-INDOCRYPT 2000, LNCS 1977, pp.203-212, 2000.
[14] MICAz Hardware Description Available at "http://www.xbow.com/Products".
[15] N. Gura, A. Patel, A. Wander, H. Eberle, and S. Chang-Shantz, "Comparing Elliptic Curve Cryptography and RSA on 8-Bit CPUs", Workshop on Cryptographic Hardware and Embedded Systems (CHES 2004), LNCS 3156, pp.119-132, 2004.
[16] TinyOS forum. Available at "http://www. tinyos.net"
[17] TelosB Hardware Description Available at "http://www.moteiv.com/products"

〈著者紹介〉

서 석 충 (Seog Chung Seo) 학생회원
2005년 2월 : 아주대학교 정보 및 컴퓨터 공학과 학사
2007년 2월 : 광주과학기술원 정보통신 공학과 석사
2007년 9월～현재 : 고려대학교 정보경영공학전문대학원 박사 과정
<관심분야> 암호 구현, 센서 네트워크

한 동 국 (Dong-Guk Han) 정회원
1999년 : 고려대학교 수학과 졸업(학사)
2002년 : 고려대학교 수학과 석사 (이학석사)
2005년 : 고려대학교 정보보호대학원 박사 (공학박사)
2004년 4월～2005년 4월 : 일본 Kyushu Univ., 방문연구원
2005년 4월～2006년 4월 : 일본 Future Univ.-Hakodate, Post.Doc.
2006년 6월～현재 : 한국전자통신연구원 정보보호연구단 선임연구원
<관심분야> 공개키 암호시스템 안전성 분석 및 고속 구현, 부채널 분석, RFID/USN 정보보호 기술

홍 석 희 (Seokhie Hong) 종신회원
1995년 : 고려대학교 수학과 학사
1997년 : 고려대학교 수학과 석사
2001년 : 고려대학교 수학과 박사
1999년 8월~2004년 2월 : (주)시큐리티 테크놀로지스 선임연구원
2003년 3월~2004년 2월 : 고려대학교 시간강사
2004년 4월~2005년 2월 : K.U. Leuven 박사후연구원
2005년 3월~현재 : 고려대학교 정보경영전문대학원 조교수
<관심분야> 대칭키 암호 알고리즘, 공개키 암호 알고리즘, 포렌식