PAPER *Special Section on Information and Communication System Security*

TinyECCK16: An Efficient Field Multiplication Algorithm on 16-bit Environment and Its Application to Tmote Sky Sensor Motes

Seog Chung SEO[†a)], ***Student Member***, **Dong-Guk HAN**[††b)], ***and*** **Seokhie HONG**[†c)], ***Nonmembers***

SUMMARY Recently, the result of TinyECCK (Tiny Elliptic Curve Cryptosystem with Koblitz curve) shows that both field multiplication and reduction over $GF(2^m)$ are related to a heavy amount of duplicated memory accesses and that reducing the number of these duplications noticeably improves the performance of elliptic curve operations such as scalar multiplications, signing and verification. However, in case that the underlying word size is extended from 8-bit to 16-bit or 32-bit, the efficiency of the techniques proposed in TinyECCK is decreased because the number of memory accesses to load or store an element in $GF(2^m)$ is significantly reduced. Therefore, in this paper, we propose a technique which makes left-to-right (ltr) comb method which is widely used as an efficient multiplication algorithm over $GF(2^m)$ suitable for extended word sizes and present TinyECCK16 (Tiny Elliptic Curve Cryptosystem with Koblitz curve on 16-bit word) which is implemented with the proposed multiplication algorithm on 16-bit Tmote Sky mote. The proposed algorithm is faster than typical ltr comb method by 15.06% and the 16-bit version of the algorithm proposed in TinyECCK by 5.12% over $GF(2^{163})$.

key words: *wireless sensor network, elliptic curve cryptosystem, efficient implementation*

1. Introduction

Wireless sensor networks (WSNs) are comprised of hundreds or thousands of resource-limited small sensor nodes. Since they are deployed in harsh, unattended environments, some combinations of authentication, integrity, and confidentiality are required for reliable and secure network communications. Due to the inherent characteristics of WSNs such as the absence of supervisors and limited resources, new security protocols considering these issues are required rather than conventional protocols. Even if some symmetric key-based security protocols have been proposed with the consideration of limited resources on motes, they lack of functionalities at pairwise key setup and broadcast authentication phases. Thus, many researchers have tried to apply public key cryptosystems to provide functionalities such as key distribution and authentication, especially elliptic curve cryptosystem (ECC), to WSNs because of its much smaller key size compared with other public key system such as RSA and DSA [1], [4], [7]–[10], [13]–[15]. They have implemented ECC on several types of motes and presented the running time and memory consumption in order to prove the feasibility of ECC on WSNs. Until now, implementations over $GF(p)$ give relatively more contented performance than those over $GF(2^m)$. However, recently the result of TinyECCK [14] shows that low performance of field operations over $GF(2^m)$ are caused by a heavy amount of memory accesses, thus reducing the number of these unnecessary operations improves the overall performance of elliptic curve operations over $GF(2^m)$. Furthermore, TinyECCK is the most efficient among ECC softwares over both $GF(p)$ and $GF(2^m)$ running on ATmega128L processor [24].

Both field multiplication and reduction are the most frequent operation in elliptic curve operations. TinyECCK has proposed some techniques which can significantly reduce the number of memory accesses in both operations, thus it could achieve performance improvement. However, the efficiency of techniques proposed in TinyECCK is decreased as the word size on a target platform is increased because the required number of memory accesses are also reduced. Actually, we have got 5.78–7.69% of improvement when TinyECCK is implemented on 16-bit word environment (comp. it was 15–19% on 8-bit platform). This result means that a new field multiplication algorithm is required with considering extended word sizes such as 16-bit and 32-bit.

In this paper, we propose a technique which makes left-to-right comb method which is widely used as an efficient multiplication algorithm over $GF(2^m)$ suitable for extended word sizes and present TinyECCK16 which is implemented with the proposed multiplication algorithm on 16-bit Tmote Sky mote [22] using MSP430 processor [25]. The proposed multiplication algorithm is faster than typical ltr comb method (resp. the improved ltr comb method proposed in TinyECCK) by 15.06% (resp. 5.12%) over $GF(2^{163})$ and with its application to TinyECCK16, 8.4–11.8% (resp. 2.82–6.93%) of running time in elliptic curve operations is saved. Furthermore, TinyECCK16 is superior to existing ECC softwares implemented on Tmote Sky sensor mote with regards to running times and memory requirements. TinyECCK16 with 5TNAF can compute a scalar multiplication in 0.64 secs and it generates (resp., verifies) a signature within 0.77 (resp., 1.27 secs) with 14,422-byte of ROM and

Manuscript received August 4, 2008.
Manuscript revised December 29, 2008.

[†]The authors are with the Graduate School of Information Management and Security, Korea University, Seoul, Korea.
[††]The author is the corresponding author of this manuscript, and is with the Dept. of Mathematics, Kookmin University, Seoul, Korea.
a) E-mail: seosc@cist.korea.ac.kr
b) E-mail: christa@kookmin.ac.kr
c) E-mail: hsh@cist.korea.ac.kr

DOI: 10.1587/transinf.E92.D.918

Table 1 Description of existing ECC implementations on sensor motes using 8-bit or 16-bit CPU (Code size are measured by bytes and timings are measured by secs, '-' means that the corresponding part is neither implemented nor known).

	Implementation on 8-bit mote							Implementation on 16-bit mote			
	Binary field					Prime field		Prime field			Binary field
	[4]	[11]	[12]	[7]	[14]	[8]	[1]	[1]	[9]	[10]	[13]
Code Size	34,342	11,592	8,767	75,088	13,748	-	19,308	13,520	17,823	19,251	20 K
Scalar Multiplication	34.00	13.90	4.14	-	1.14	0.81	-	-	-	-	32.50
Sign	-	-	-	6.88	1.37	-	2.00	1.58	3.35	1.60	-
Verify	-	-	-	24.17	2.32	-	2.43	2.02	6.78	3.32	-

1,750-byte of RAM. Since the efficiency of the proposed algorithm is increased as the word size on the target platform is extended, it seems that our proposal is promising technique for upcoming more improved sensor platforms.

2. Related Work

Until now, there have been several implementations of ECC over both $GF(2^m)$ and $GF(p)$ on sensor motes using 8-bit or 16-bit CPU. They have tried to prove the feasibility of ECC for WSNs.

2.1 Existing Implementations on 16-bit Sensor Motes

There are some ECC softwares over both $GF(p)$ and $GF(2^m)$ on TelosB or Tmote Sky motes using 16-bit MSP430 processor. TinyECC which is tuned for 16-bit Tmote Sky can generate a signature in 1.58 secs and verify it in 2.02 secs at the expense of 13,520-byte of ROM and 1,504-byte of RAM [1]. Wang et al. have implemented ECC software over $GF(p)$ on TelosB mote running at 8 MHz and have applied it for their proposed access control protocol [9]. Their implementation consumes 3.35 and 6.78 secs for signing and verification, respectively at the cost of 17,823-byte of ROM and 1,638-byte of RAM. After their early work, they have significantly improved the performance of their code [10]. The updated code takes 1.60 and 3.32 secs for signing and verification, respectively. At this time, it requires 19,251-byte of ROM and 1,392-byte of RAM except for SHA-1 code more than 30-Kbyte. Arazi et al. have tuned EccM for 16-bit TelosB mote. The modified implementation takes 32.5 secs for a scalar multiplication with 20 K-byte of ROM and 1,500-byte or RAM [13].

2.2 NanoECC

NanoECC [15] is based on MIRACL library [19] and supports ECC operations and pairing-based cryptographic operations on both MICAz and Tmote Sky mote over both $GF(p)$ and $GF(2^m)$. For efficient elliptic curve operations over $GF(2^m)$, NanoECC is based on Koblitz curve and uses Karasutba-Ofman multiplication algorithm and fast reduction algorithm using pentanomial. NanoECC over $GF(p)$ makes use of hybrid multiplication algorithm and optimized reduction algorithm using Mersenne-prime. It applies fixed-based comb algorithm with window size 4 for fast scalar multiplications. On a MICAz (resp. Tmote Sky) mote, it takes NanoECC 2.16 (resp. 1.04) and 1.27 (resp. 0.72) secs to compute a scalar multiplication over $GF(2^m)$ and $GF(p)$, respectively.

For extensive description of existing ECC implementations on sensor motes 8-bit or 16-bit CPU, Table 1 is presented. Since NanoECC is a kind of multi-platform implementation, we do not include it in the table.

3. Implementation Details

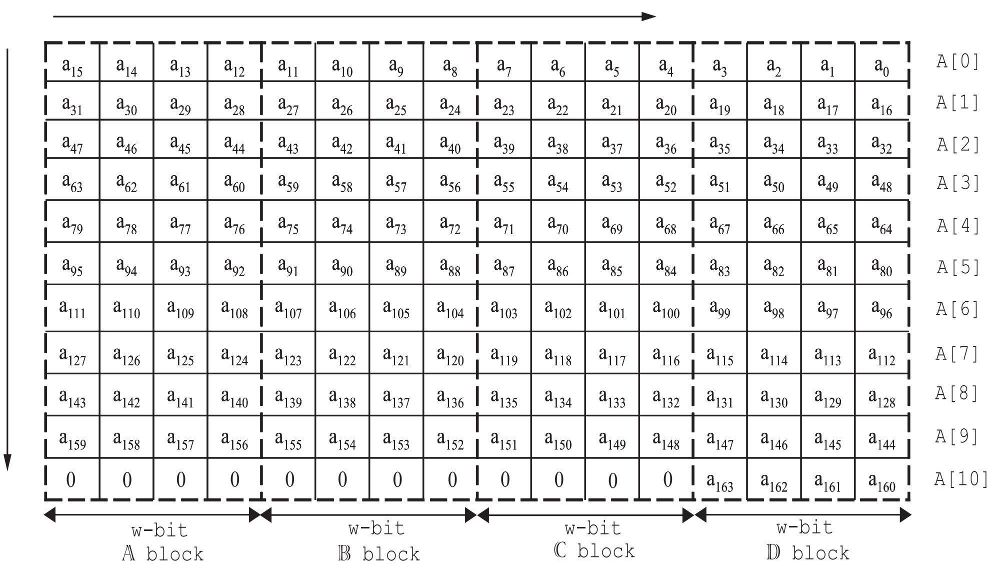
We have implemented TinyECCK16 on a 16-bit Tmote Sky mote [22] using the MSP430 processor [25]. We use the domain parameter (sect163k1) recommended by [3] and polynomial basis to represent elements in $GF(2^m)$. TinyECCK16 has been developed with nesc language [27] in order to run on TinyOS [21]. We modified the original field arithmetic algorithms using 32-bit word size which are presented in Guide to Elliptic Curve Cryptography [5], [6] into the forms suitable for 16-bit word environment. For efficiency, TinyECCK16 makes use of recoding algorithms such as *w*NAF and *w*TNAF [16] and selects the mixed coordinate system [17] rather than affine coordinate.

3.1 Sensor Platform Description

Table 2 describes the features of MICAz and Tmote Sky motes. Tmote Sky mote has more sufficient RAM size than MICAz mote. TinyECCK running on MICAz mote uses 4TNAF recoding algorithm. Because more sufficient RAM size is available in Tmote Sky mote, larger window size can be used for *w*TNAF. Actually, TinyECCK16 makes use of 5TNAF for the maximum performance, at this time, 1,750-byte of RAM is used. 5TNAF cannot be used by TinyECCK running on MICAz mote since only 4-Kbyte of RAM is available on that mote. Since a Tmote Sky has only 48-Kbyte of ROM, the code size is more critical issue than MICAz motes.

3.2 Preliminaries

Let assume word size W be 16-bit since MSP430 uses 16-bit data bus. Following notations are used in the rest of this paper for describing algorithms. We assume that A $(= a(z))$ and B $(= b(z))$ are elements in $GF(2^m)$.



a15 a14 a13 a12 a11 a10 a9 a8 a7 a6 a5 a4 a3 a2 a1 a0 A[0]
a31 a30 a29 a28 a27 a26 a25 a24 a23 a22 a21 a20 a19 a18 a17 a16 A[1]
a47 a46 a45 a44 a43 a42 a41 a40 a39 a38 a37 a36 a35 a34 a33 a32 A[2]
a63 a62 a61 a60 a59 a58 a57 a56 a55 a54 a53 a52 a51 a50 a49 a48 A[3]
a79 a78 a77 a76 a75 a74 a73 a72 a71 a70 a69 a68 a67 a66 a65 a64 A[4]
a95 a94 a93 a92 a91 a90 a89 a88 a87 a86 a85 a84 a83 a82 a81 a80 A[5]
a111 a110 a109 a108 a107 a106 a105 a104 a103 a102 a101 a100 a99 a98 a97 a96 A[6]
a127 a126 a125 a124 a123 a122 a121 a120 a119 a118 a117 a116 a115 a114 a113 a112 A[7]
a143 a142 a141 a140 a139 a138 a137 a136 a135 a134 a133 a132 a131 a130 a129 a128 A[8]
a159 a158 a157 a156 a155 a154 a153 a152 a151 a150 a149 a148 a147 a146 a145 a144 A[9]
0 0 0 0 0 0 0 0 0 0 0 0 a163 a162 a161 a160 A[10]
w-bit 𝔸 block
w-bit 𝔹 block
w-bit ℂ block
w-bit 𝔻 block

Fig. 1 Left-to-right comb method using window (It scans w-bit of A from $\mathbb{A}$ block to $\mathbb{D}$ block in such a way that each block is processed from top to bottom. Then, the precomputed result corresponding to the scanned bit is loaded from the table and xored with the intermediate result C. $A \in GF(2^{163})$, $w = 4$).

Table 2 Sensor mote specifications.

	MICAz [20]	Tmote Sky [22]
MCU	8-bit ATmega128L	16-bit MSP430
Flash Memory	128-Kbyte	48-Kbyte
RAM	4-Kbyte	10-Kbyte

- $A \oplus B$: bitwise exclusive-or.
- A & B: bitwise AND.
- $A \gg i$: right shift of A by i positions with padding i upper bits as 0.
- $A \ll i$: left shift of A by i positions with padding i lower bits as 0.
- W: a 16-bit word.
- $A[j]$ denotes j-th word of the A polynomial.
- $t = \lceil m/W \rceil$ is the required number of words to store A in memory.
- The left-to-right comb method using window w processes the bits of $a(z)$ from left to right direction as follows: $a(z) \cdot b(z) = (\cdots((\tilde{a}_{s-1}b(z)z^w + \tilde{a}_{s-2}b(z))z^w + \tilde{a}_{s-3}b(z))z^w + \cdots + \tilde{a}_1 b(z))z^w + \tilde{a}_0 b(z)$, where $\tilde{a}_i = (a_{wi+w-1} \ldots a_{wi+1} a_{wi})$, $0 \le i \le s-1$, $s = \lfloor \frac{m}{w} \rfloor + 1$.
- Each of $\mathbb{A}$, $\mathbb{B}$, $\mathbb{C}$, and $\mathbb{D}$ means a block in Fig. 1.

Algorithm 1 Typical left-to-right comb multiplication method on 16-bit word (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: $masking \leftarrow 0xf000$.
 6: for $k \leftarrow 3$ to 0 decrements $k$ by 1 do
 7:     for $j \leftarrow 0$ to $t - 1$ increments $j$ by 1 do
 8:         $u \leftarrow ((a[j]$ & $masking) \gg (k * 4))$.
 9:         for $i \leftarrow 0$ to $t$ increments $i$ by 1 do
10:             $C[i + j] \leftarrow C[i + j] \oplus T_u[i]$.
11:         end for
12:     end for
13:     if $k \neq 0$ then
14:         $C \leftarrow C \cdot z^w$.
15:         $masking \leftarrow masking \gg 4$.
16:     end if
17: end for
18: Return $(c)$
```

3.3 Field Arithmetics Over $GF(2^m)$

3.3.1 Field Multiplication

Left-to-right (ltr) comb method [18] using window is widely used when implementing field multiplication over $GF(2^m)$. It is a kind of lookup table-based multiplication algorithm. Algorithm 1 is typical ltr comb method using window (See Fig. 1). It first builds a precomputation table about all possible results of $u(z) \cdot b(z)$ for all polynomials $u(z)$ of degree at most $w - 1$ ($b(z)$ is multiplicand and w is window size). Then, it scans w-bit of multiplier $a(z)$ at a time and takes the corresponding result from the precomputation table instead of computing it. Namely, this method can save the number of XOR operations and memory accesses compared with naive shift-and-xor multiplication algorithm at the expense of more memory consumption. Considering the tradeoff between the overhead of precomputation and its advantage during computing partial multiplications, the proper window size of ltr comb method is known as 4. Thus, the field multiplication algorithms discussed in this paper are all based on window size 4.

Algorithm 2 is the 16-bit version of the ltr comb method proposed in [14]. This algorithm is a kind of improved version of Algorithm 1. Actually, step 7–12 in Algorithm 1, partial multiplication, requires many duplicated

Algorithm 2 16-bit version of left-to-right comb multiplication method proposed in TinyECCK [14] (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: $masking \leftarrow 0xf000$.
6: for $k \leftarrow 3$ to 0 decrements $k$ by 1 do
7:     for $j \leftarrow 0$ to $t - 1$ increments $j$ by 2 do
8:         $u_1 \leftarrow ((a[j]\ \&\ masking) \gg (k * 4))$.
9:         $u_2 \leftarrow ((a[j+1]\ \&\ masking) \gg (k * 4))$.
10:        $C[j] \leftarrow C[j] \oplus T_{u_1}[0], C[j+t+1] \leftarrow C[j+t+1] \oplus T_{u_2}[t]$.
11:        for $i \leftarrow 1$ to $t$ increments $i$ by 1 do
12:            $C[i+j] \leftarrow C[i+j] \oplus T_{u_1}[i] \oplus T_{u_2}[i-1]$.
13:        end for
14:    end for
15:    if $k \neq 0$ then
16:        $C \leftarrow C \cdot z^w$.
17:        $masking \leftarrow masking \gg 4$.
18:    end if
19: end for
20: Return ($c$)
```

memory accesses. Thus, Algorithm 2 reduces this overhead by combining two instances of these steps (Refer to [14] for details) into one. This results in saving the number of duplicated LOAD and STORE instructions. Following is the procedure of Algorithm 2.

1. Step 3 builds a precomputation table about $u(z) \cdot b(z)$ (Since the used window size is 4, it contains the result from $1 \cdot b(z)$ to $(z^3 + z^2 + z + 1) \cdot b(z)$).
2. Each of step 8 and 9 scans w-bit from $a(z)$ for processing two consecutive partial multiplications.
3. Step 10–13 executes two combined partial multiplications. In other words, it takes the precomputed results corresponding to the scanned bits from the table, then the loaded results are xored with the intermediate result C at the propoer bit positions.
4. Step 15–18 shifts the intermediate result C because this algorithm executes left-to-right manner. For example, results of computing partial multiplications from $\mathbb{A}$, $\mathbb{B}$, $\mathbb{C}$ blocks in Fig. 1 should be shifted to left as much as 12-bit, 8-bit, 4-bit, respectively.

Remark 1. The intermediate result, C, consists of $2t$ words since it is the result of $a(z) \cdot b(z)$. Thus, $2t$ words should be left-shifted three times (Step 15–18 in Algorithm 2). Actually, 8-bit version of Algorithm 2 requires shifting $2t$ words only once; this overhead is relatively small compared with the overhead due to the redundant memory accesses. However, on 16-bit word environment, while the number of memory accesses during a field multiplication is reduced in half, the number of shifting C is increased from once to three times. Thus, the overhead from shifting C occupies larger portion during a field multiplication than 8-bit environment. For solving this problem, we present a promising technique in Sect. 4.

Algorithm 3 16-bit version fast reduction modulo $f(z) = z^{163} + z^7 + z^6 + z^3 + 1$ proposed in TinyECCK [14]

```
1: INPUT: A binary polynomial $c(z)$ of degree at most 324
2: OUTPUT: $c(z)$ mod $f(z)$
3: for $i \leftarrow 21$ to 14 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:     $C[i-9] \leftarrow C[i-9] \oplus (T_1 \gg 12) \oplus (T_1 \gg 13)$.
6:     $C[i-10] \leftarrow C[i-10] \oplus (T_1 \ll 4) \oplus (T_1 \ll 3) \oplus T_1 \oplus (T_1 \gg 3) \oplus (T_2 \gg 12) \oplus (T_2 \gg 13)$.
7:     $C[i-11] \leftarrow C[i-11] \oplus (T_1 \ll 13) \oplus (T_2 \ll 4) \oplus (T_2 \ll 3) \oplus T_2 \oplus (T_2 \gg 3) \oplus (T_3 \gg 12) \oplus (T_3 \gg 13)$.
8:     $C[i-12] \leftarrow C[i-12] \oplus (T_2 \ll 13) \oplus (T_3 \ll 4) \oplus (T_3 \ll 3) \oplus T_3 \oplus (T_3 \gg 3) \oplus (T_4 \gg 12) \oplus (T_4 \gg 13)$.
9:     $C[i-13] \leftarrow C[i-13] \oplus (T_3 \ll 13) \oplus (T_4 \ll 4) \oplus (T_4 \ll 3) \oplus T_4 \oplus (T_4 \gg 3)$.
10:    $C[i-14] \leftarrow C[i-14] \oplus (T_4 \ll 13)$.
11: end for
12: $T_1 \leftarrow C[13], T_2 \leftarrow C[12], T_3 \leftarrow C[11], T_4 \leftarrow C[10] \gg 3$.
13: $C[0] \leftarrow C[0] \oplus (T_3 \ll 13) \oplus (T_4 \ll 7) \oplus (T_4 \ll 6) \oplus (T_4 \ll 3) \oplus T_4$.
14: $C[1] \leftarrow C[1] \oplus (T_2 \ll 13) \oplus (T_3 \ll 4) \oplus (T_3 \ll 3) \oplus T_3 \oplus (T_3 \gg 3) \oplus (T_4 \gg 10) \oplus (T_4 \gg 9)$.
15: $C[2] \leftarrow C[2] \oplus (T_1 \ll 13) \oplus (T_2 \ll 4) \oplus (T2 \ll 3) \oplus T_2 \oplus (T_2 \gg 3) \oplus (T_3 \gg 12) \oplus (T_3 \gg 13)$.
16: $C[3] \leftarrow C[3] \oplus (T_1 \ll 4) \oplus (T_1 \ll 3) \oplus T_1 \oplus (T_1 \gg 3) \oplus (T_2 \gg 12) \oplus (T_2 \gg 13)$.
17: $C[4] \leftarrow C[4] \oplus (T_1 \gg 12) \oplus (T_1 \gg 13)$.
18: $C[10] \leftarrow C[10]\ \&\ 0x0007$
19: Return $C[10], \ldots, C[2], C[1], C[0]$.
```

Table 3 Comparison of field operations in $GF(2^{163})$ (times for both multiplication and squaring include the time for modular reduction, all timings are measured by secs).

Field oepration	Execution time	Inversion/operation
Multiplication	0.00178422	23.57
Squaring	0.00023538	178.68
Inversion	0.04205775	-

3.3.2 Field Reduction

The result of both field multiplication and field squaring over $GF(2^m)$ should be reduced by irreducible polynomial $f(z)$. Reduction polynomials such as sparse trinomial and pentanomial recommended by NIST [3] in FIPS 186-2 are often used for efficient reduction. TinyECCK16 makes use of $f(z) = z^{163} + z^7 + z^6 + z^3 + 1$. Algorithm 3 is the 16-bit version of the fast reduction algorithm proposed in [14]. Step 3-11 conducts the reduction to C of $2t$ words in such a way that redundant memory accesses are reduced. Step 12-17 is a postprocessing part for remaining words and it is more complex than 8-bit version.

3.4 Selection of Coordinate System

Table 3 shows the ratio of inversion to multiplication and squaring over $GF(2^{163})$ on MSP430 processor. Since the ratio of inversion to multiplication is 23.57, it is more desirable to eliminate the inversions during scalar multiplications

(For inversion, binary extended Euclidean algorithm [6] is used)[†]. Therefore, we choose to use the López-Dahab coordinate rather than affine coordinate [17]. TinyECCK16 uses the mixed coordinates for elliptic curve point addition (ECADD) since the addition of two points represented in different coordinate system is more efficient than that of two points using the same representation [5], [6]. Hence, we build a precomputation table of the points represented in affine coordinate.

4. Proposed Efficient Field Multiplication on 16-bit Environment

With algorithm 2 and 3, TinyECCK16 saves only 5.78–7.69% of running time in elliptic curve operations. This is contrastive to the result that applying 8-bit versions of these algorithms on ATmega128 processor save 15-19% of TinyECCK's running time. The main reason of this result is because, on the one hand, the use of extended word size (16-bit word) reduces the number of memory accesses during multiplication and reduction algorithm, on the other hand, the number of shifting C of $2t$ words is increased from once to three times. This section describes a proposal which can make algorithm 2 use only one of shifting C of $2t$ words.

4.1 Proposed Left-to-Right Comb Method on 16-bit Word

The number of shifting C can be reduced from three to one by rearranging the sequence of processing blocks in Fig. 1. Namely, new ltr comb method operates in such a manner that $\mathbb{A} \to \mathbb{C} \to \mathbb{B} \to \mathbb{D}$ blocks instead of $\mathbb{A} \to \mathbb{B} \to \mathbb{C} \to \mathbb{D}$ blocks.

4.1.1 Main Idea and How to Implement

A ($= \sum_{i=0}^{43} \tilde{a}_i \cdot z^{4\cdot i}$, where $\tilde{a}_{41}, \tilde{a}_{42}, \tilde{a}_{43}$ are set to zero) in Fig. 1 can be expressed as follows.

$$(z^{12} \textstyle\sum_{i=0}^{10}(\tilde{a}_{4i+3} \cdot z^{4i})) + (z^8 \sum_{i=0}^{10}(\tilde{a}_{4i+2} \cdot z^{4i})) \\ +(z^4 \textstyle\sum_{i=0}^{10}(\tilde{a}_{4i+1} \cdot z^{4i})) + (\sum_{i=0}^{10}(\tilde{a}_{4i} \cdot z^{4i})).$$

At this time, $\mathbb{A}$, $\mathbb{B}$, $\mathbb{C}$ and $\mathbb{D}$ blocks can be expressed as $(z^{12} \sum_{i=0}^{10}(\tilde{a}_{4i+3} \cdot z^{4i}))$, $(z^8 \sum_{i=0}^{10}(\tilde{a}_{4i+2} \cdot z^{4i}))$, $(z^4 \sum_{i=0}^{10}(\tilde{a}_{4i+1} \cdot z^{4i}))$, and $(\sum_{i=0}^{10}(\tilde{a}_{4i} \cdot z^{4i}))$, respectively. According to above equations, $\mathbb{A}$ (resp. $\mathbb{B}$) block is left-shifted as 8-bit compared with $\mathbb{C}$ (resp. $\mathbb{D}$) block. $A \cdot B = a(z) \cdot b(z)$ can be expressed as

$$(z^{12} \textstyle\sum_{i=0}^{10}(\tilde{a}_{4i+3} \cdot z^{4i} \cdot b(z)) + (z^8 \sum_{i=0}^{10}(\tilde{a}_{4i+2} \cdot z^{4i} \cdot b(z)) \\ +(z^4 \textstyle\sum_{i=0}^{10}(\tilde{a}_{4i+1} \cdot z^{4i} \cdot b(z)) + (\sum_{i=0}^{10}(\tilde{a}_{4i} \cdot z^{4i} \cdot b(z))$$

Thus, the partial products of $\mathbb{A}$ block and $b(z)$ are stored at 8-bit left-shifted position from where those of $\mathbb{C}$ block and $b(z)$ are stored. Now, two shift operations are replaced by storing the results of $\mathbb{A}$ (resp. $\mathbb{B}$) block at 8-bit incremented address from where the results of $\mathbb{C}$ (resp. $\mathbb{D}$) block are stored. Consequently, the results of partial multiplications in $\mathbb{A}$, $\mathbb{B}$ and

Algorithm 4 Proposed left-to-right comb method combined with the technique in [14]

```
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: $PTR_C \leftarrow \&C+1$. // assigning 1-byte incremented address of $C$
6: $masking \leftarrow 0xf000$.
7: // Processing $\mathbb{A}$, $\mathbb{C}$ blocks
8: for $k \leftarrow 3$ to 1 decrements $k$ by 2 do
9:     for $j \leftarrow 0$ to $t-1$ increments $j$ by 2 do
10:        $u_1 \leftarrow ((a[j] \,\&\, masking) \gg (k*4))$.
11:        $u_2 \leftarrow ((a[j+1] \,\&\, masking) \gg (k*4))$.
12:        $*(PTR_C + j) \leftarrow *(PTR_C + j) \oplus T_{u_1}[0]$.
13:        $*(PTR_C + j + t + 1) \leftarrow *(PTR_C + j + t + 1) \oplus T_{u_2}[t]$.
14:        for $i \leftarrow 1$ to $t$ increments $i$ by 1 do
15:            $*(PTR_C + i + j) \leftarrow *(PTR_C + i + j) \oplus T_{u_1}[i] \oplus T_{u_2}[i-1]$.
16:        end for
17:    end for
18:    $masking \leftarrow masking \gg 8$.
19:    $PTR_C \leftarrow \&C$. // assigning C's original address
20: end for
21: $C \leftarrow C \cdot z^w$.
22: $PTR_C \leftarrow \&C+1$. // assigning 1-byte incremented address of $C$
23: $masking \leftarrow 0x0f00$.
24: // Processing $\mathbb{B}$, $\mathbb{D}$ blocks
25: for $k \leftarrow 2$ to 0 decrements $k$ by 2 do
26:     for $j \leftarrow 0$ to $t-1$ increments $j$ by 2 do
27:        $u_1 \leftarrow ((a[j] \,\&\, masking) \gg (k*4))$.
28:        $u_2 \leftarrow ((a[j+1] \,\&\, masking) \gg (k*4))$.
29:        $*(PTR_C + j) \leftarrow *(PTR_C + j) \oplus T_{u_1}[0]$.
30:        $*(PTR_C + j + t + 1) \leftarrow *(PTR_C + j + t + 1) \oplus T_{u_2}[t]$.
31:        for $i \leftarrow 1$ to $t$ increments $i$ by 1 do
32:            $*(PTR_C + i + j) \leftarrow *(PTR_C + i + j) \oplus T_{u_1}[i] \oplus T_{u_2}[i-1]$.
33:        end for
34:    end for
35:    $masking \leftarrow masking \gg 8$.
36:    $PTR_C \leftarrow \&C$. // assigning C's original address
37: end for
38: Return $(c)$
```

$\mathbb{C}$ blocks are left-shifted as 12-bit, 8-bit, and 4-bit, respectively, like what Algorithm 2 does.

To implement our proposal, the results of $\mathbb{A}$ and $\mathbb{B}$ blocks should be stored at 8-bit incremented address from where $\mathbb{C}$ and $\mathbb{D}$ blocks are stored. This can be possible with following codes written in C language.

`BYTE16 C[DOUBLENUMWORDS];` // arrary storing the running result C
`BYTE8* ptrC8 =(BYTE8*)C;` // assigning the base address of C at, an 8-bit pointer, *ptrC*8
`BYTE16* ptrC16 = ptrC8+1;` // incrementing the address of *ptrC*8 as 8-bit and storing it at, a 16-bit pointer, *ptrC*16

The results of $\mathbb{A}$ and $\mathbb{B}$ blocks are stored from the *ptrC*16 address. With this manner, the results of partial multiplica-

[†]Even if the running time of a inversion is significantly reduced by using 16-bit word, it is still much bigger than that of field multiplication.

tions of both $\mathbb{A}$ and $\mathbb{B}$ blocks equivalent to be left-shifted by 8-bit.

Algorithm 4 is the improved version of Algorithm 2 by reducing the number of shifting the running result C of $2t$ words from three to one on 16-bit environment. Step 5 and 22 of Algorithm 4 increments the address of running result C and stores it at the $PTRc$, a 16-bit pointer. In this manner, step 12–16 and step 29–33 are equivalent to shifting and storing the results of partial multiplications in $\mathbb{A}$ and $\mathbb{B}$ blocks to 8-bit left direction compared with $\mathbb{C}$ and $\mathbb{D}$ blocks' results. The base address is restored at step 19 and 36 because the results of $\mathbb{C}$ and $\mathbb{D}$ blocks should be stored at the original address of C. The proposed algorithm requires only one shifting operation of $2t$ words compared with Algorithm 2.

4.1.2 Theoretical Analysis

We can count how many instructions are saved with the proposed method.

Theorem 1. Typical ltr comb method using window 4 requires $[(8t^2+52t-4)L+(4t^2+35t+3)S+(4t^2+11t)X+(30t-4)SH]$ instructions for computing $a(z)\cdot b(z)$ over $GF(2^m)$ (L, S, X, and SH mean LOAD, STORE, XOR, and SHIFT instruction, respectively, $t=\frac{m}{W}$, $W=16$).

Proof. The computation of ltr comb method using window can be divided into three parts; precomputation, computing partial multiplications and shifting the intermediate result C. Thus, the cost of Algorithm 1 can be computed as follows.

- `Precomputation`
  $[(28t-7)L+(21t)S+(7t)X+(14t-7)SH]$.
- `Computing partial multiplications`
  $[(8t^2+12t)L+(4t^2+8t)S+(4t^2+4t)X+(4t)SH]$ [†].
- `Shifting the running result` C
  $3[(4t+1)L+(2t+1)S+(4t+1)SH]$.

The precomputation for $u(z)\cdot b(z)$ with window size 4 requires seven XOR additions and seven 1-bit left shifting operations on arrays of t words. We optimize this precomputation process by reducing the number of loops. Namely, we combine XOR addition and 1-bit left shift operation. For example, $z\cdot b(z)$ and $(z+1)\cdot b(z)$, $z^2\cdot b(z)$ and $(z^2+1)\cdot b(z)$ can be consecutively calculated. In our implementation, the cost of combined XOR addition and 1-bit shifting is $[(4t-1)L+(3t)S+(t)X+(2t-1)SH]$.

The following is the pseudo code for shifting the intermediate result C as 4-bit to the left direction.

```
for(i = 2t; i > 0; i--)
{ C[i] = (C[i] << 4) | (C[i-1] >> 12); }
C[0] = C[0] << 4;
```

Since $C[i]$ and $C[i-1]$ are loaded and two shift operations are used at each iteration, the cost for a shifting the C is $[(4t+1)L+(2t+1)S+(4t+1)SH]$.

Thus, total cost of Algorithm 1 is $[(8t^2+52t-4)L+(4t^2+35t+3)S+(4t^2+11t)X+(30t-4)SH]$ by summing the cost of each part. □

Theorem 2. Algorithm 4 requires $[(6t^2+44t-6)L+(2t^2+31t+1)S+(4t^2+11t)X+(22t-6)SH]$ instructions.

Proof. Since the precomputation step of Algorithm 4 is same as Algorithm 1, the cost is identical. However, the costs for computing partial multiplications and shifting the C are reduced in Algorithm 4. The costs for two parts are described as follows.

- `Computing partial multiplications`
  $[(6t^2+12t)L+(2t^2+8t)S+(4t^2+4t)X+(4t)SH]$ [††].
- `Shifting the running result` C
  $[(4t+1)L+(2t+1)S+(4t+1)SH]$.

According to the above analysis, the total cost of Algorithm 4 are $[(6t^2+44t-6)L+(2t^2+31t+1)S+(4t^2+11t)X+(22t-6)SH]$. □

Since the difference between Algorithm 2 and Algorithm 4 is the number of shifting the C, we can easily compute the cost of Algorithm 2; $[(6t^2+52t-4)L+(2t^2+35t+3)S+(4t^2+11t)X+(30t-4)SH]$.

On the ground of Theorem 1 and Theorem 2, we can count the number of saved instructions with Algorithm 4 instead of Algorithm 1 and Algorithm 2. In other words, Algorithm 4 saves $[(2t^2+8t+2)L+(2t^2+4t+2)S+(8t+2)SH]$ and $[(8t+2)L+(4t+2)S+(8t+2)SH]$ instructions compared with Algorithm 1 and Algorithm 2, respectively. Since $GF(2^{163})$ and 16-bit word are used, t is equal to 11 ($=\lceil\frac{163}{16}\rceil$). Therefore, Algorithm 4 saves $(332L+288S+90SH)$ (resp. $(90L+46S+90SH)$) instructions in comparison with Algorithm 1 (resp. Algorithm 2).

Because each of Algorithm 1 and Algorithm 2 requires $(1536L+872S+605X+326SH)$ and $(1294L+630S+605X+326SH)$, Algorithm 4 contributes to 21.26% (resp. 7.91%) of saving instead of Algorithm 1 (resp. Algorithm 2) [†††].

4.2 Application to More Extended Word Size

The proposed technique can be applied for more extended word size such as 32-bit word. Actually, Imote2 [23], a state-of-art sensor mote, uses 32-bit PXA271 processor [26].

4.2.1 How to Apply

Typical ltr comb multiplication algorithm on 32-bit word

[†] In Algoritm 1, step 8 and step 9–11 require $(1L+1S+1SH)$ and $(t+1)(2L+1S+1X)$, respectively.

[††] Step 10–13 and Step 14–16 in Algorithm 4, the combined partial multiplication, requires $(6L+4S+2X+2SH)$ and $t(3L+1S+2X)$, respectively.

[†††] Because one instruction requires different number of cycles according to addressing modes, we assume that LOAD, STORE, XOR and SHIFT operation use same number of clock cycles

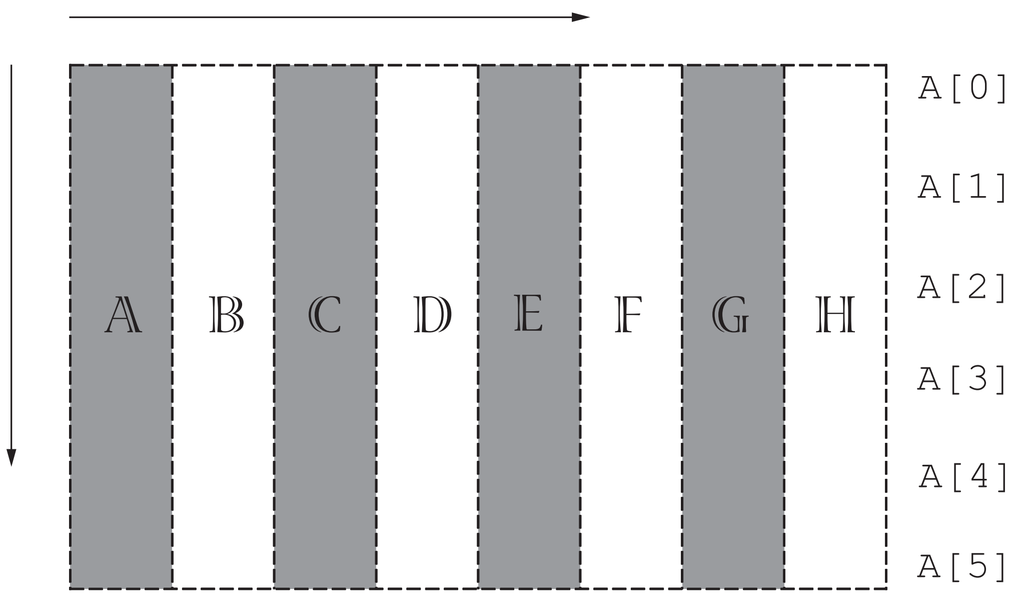


Fig. 2 Left-to-right comb method on 32-bit word (If typical ltr comb method using window size 4 is implemented on a 32-bit processor, A is divided into 8 blocks. Thus, 7 times of shifting C of $2t$ words are necessary).

operates as Fig. 2. In this case, the word is divided into eight blocks. Each block is scanned from top to bottom and the corresponding result is loaded from a precomputation table and xored with the intermediate result C. $a(z) \cdot b(z)$ can be expressed as

$$\begin{aligned}&(z^{28} \textstyle\sum_{i=0}^{5}(\tilde{a}_{4i+7} \cdot z^{4i} \cdot b(z))) + (z^{24} \sum_{i=0}^{5}(\tilde{a}_{4i+6} \cdot z^{4i} \cdot b(z)))\\ &\quad +(z^{20} \textstyle\sum_{i=0}^{5}(\tilde{a}_{4i+5} \cdot z^{4i} \cdot b(z))) + (z^{16} \sum_{i=0}^{5}(\tilde{a}_{4i+4} \cdot z^{4i} \cdot b(z)))\\ &\quad +(z^{12} \textstyle\sum_{i=0}^{5}(\tilde{a}_{4i+3} \cdot z^{4i} \cdot b(z))) + (z^{8} \sum_{i=0}^{5}(\tilde{a}_{4i+2} \cdot z^{4i}) \cdot b(z))\\ &\quad +(z^{4} \textstyle\sum_{i=0}^{5}(\tilde{a}_{4i+1} \cdot z^{4i}) \cdot b(z)) + (\sum_{i=0}^{5}(\tilde{a}_{4i} \cdot z^{4i}) \cdot b(z)).\end{aligned}$$

Thus, Algorithm 1 and Algorithm 2 using window size 4 on 32-bit word require seven shifting of the C. However, this overhead can be reduced into only one shifting operation by rearranging the sequence of processing blocks and the position where the partial multiplications to be stored. The sequence of processing blocks are $\mathbb{A} \rightarrow \mathbb{C} \rightarrow \mathbb{E} \rightarrow \mathbb{G} \rightarrow \mathbb{B} \rightarrow \mathbb{D} \rightarrow \mathbb{F} \rightarrow \mathbb{H}$. Details are as follows.

1. The results of partial multiplications on $\mathbb{A}$ block are stored from the 3-byte incremented address than the base address of C.
2. The results of partial multiplications on $\mathbb{C}$ block are stored from the 2-byte incremented address than the base address of C.
3. The results of partial multiplications on $\mathbb{E}$ block are stored from the 1-byte incremented address than the base address of C.
4. The results of partial multiplications on $\mathbb{G}$ block are stored from the base address of C.
5. Shift the intermediate result C to left direction as 4-bit. Through this operation, results of block $\mathbb{A}$, $\mathbb{C}$, $\mathbb{E}$, and $\mathbb{G}$ are left-shifted as 28-bit, 20-bit, 12-bit and 4-bit, respectively.
6. The results of partial multiplications on $\mathbb{B}$ block are stored from the 3-byte incremented address than the base address of C.
7. The results of partial multiplications on $\mathbb{D}$ block are stored from the 2-byte incremented address than the base address of C.
8. The results of partial multiplications on $\mathbb{F}$ block are stored from the 1-byte incremented address than the base address of C.
9. The results of partial multiplications on $\mathbb{H}$ block are stored at the base address of C.
10. As a result, the results of blocks $\mathbb{A}$, $\mathbb{B}$, $\mathbb{C}$, $\mathbb{D}$, $\mathbb{E}$, $\mathbb{F}$ and $\mathbb{G}$ are shifted to left direction as 28-bit, 24-bit, 20-bit, 16-bit, 12-bit, 8-bit, and 4-bit, respectively.

When a field multiplication algorithm over $GF(2^m)$ (m is fixed) is implemented on 32-bit word, the number of memory accesses to load and store an element is more reduced compared with using 8-bit or 16-bit word. Thus, the performance gain from the techniques, proposed in [14], reducing memory accesses become less effective on 32-bit environment. However, the proposed method will be more promising since it reduces the number of shifting $2t$ words from seven to one.

4.2.2 Estimation of Performance Gain

We can count the saved number of instructions on 32-bit word when the proposed algorithm is used instead of Algorithm 1 and Algorithm 2.

Theorem 3. Algorithm 1 on 32-bit word requires $[(16\bar{t}^2 + 80\bar{t})L + (8\bar{t}^2 + 51\bar{t} + 7)S + (8\bar{t}^2 + 15\bar{t})X + (50\bar{t})SH]$, where $\bar{t} = \frac{m}{\bar{W}}$, $\bar{W} = 32$.

Table 4 Comparison the running times among three field multiplication algorithms over $GF(2^{163})$. The improvement ratio in second row is for Algorithm 1 and the ratio in third row is for Algorithm 2 (all times are measured by secs).

Field operations	Algorithm 1	Algorithm 2	Algorithm 4	Improvement (%)
Multiplication	0.00210057	0.00188060	0.00178422	15.06% 5.12%

Proof. The cost of Algorithm 1 can be easily computed from Theorem 1.

- `Computing partial multiplications` $[(16\bar{t}^2 + 24\bar{t})L + (8\bar{t}^2 + 16\bar{t})S + (8\bar{t}^2 + 8\bar{t})X + (8\bar{t})SH]$.
- `Shifting the running result` C $7[(4\bar{t} + 1)L + (2\bar{t} + 1)S + (4\bar{t} + 1)SH]$.

The total number of required instructions in Algorithm 1 is $[(16\bar{t}^2 + 80\bar{t})L + (8\bar{t}^2 + 51\bar{t} + 7)S + (8\bar{t}^2 + 15\bar{t})X + (50\bar{t})SH]$ including the cost of precomputation. □

Theorem 4. Algorithm 4 on 32-bit word requires $[(12\bar{t}^2 + 56\bar{t} - 6)L + (4\bar{t}^2 + 39\bar{t} + 1)S + (8\bar{t}^2 + 15\bar{t})X + (26\bar{t} - 6)SH]$ instructions.

Proof. The cost of Algorithm 4 can be derived from Theorem 2 as follows.

- `Computing partial multiplications` $[(12\bar{t}^2 + 24\bar{t})L + (4\bar{t}^2 + 16\bar{t})S + (8\bar{t}^2 + 8\bar{t})X + (8\bar{t})SH]$.
- `Shifting the running result` C $[(4\bar{t} + 1)L + (2\bar{t} + 1)S + (4\bar{t} + 1)SH]$.

By summing the cost of each part in Algorithm 4, it requires $[(12\bar{t}^2+56\bar{t}-6)L+(4\bar{t}^2+39\bar{t}+1)S+(8\bar{t}^2+15\bar{t})X+(26\bar{t}-6)SH]$ instructions for computing a field multiplication. □

By the same manner, the cost of Algorithm 2 can be computed as $[(12\bar{t}^2 + 80\bar{t})L + (4\bar{t}^2 + 51\bar{t} + 7)S + (8\bar{t}^2 + 15\bar{t})X + (50\bar{t})SH]$.

On the basis of Theorem 3 and Theorem 4, Algorithm 4 saves $[(4\bar{t}^2 + 24\bar{t} + 6)L + (4\bar{t}^2 + 12\bar{t} + 6)S + (24\bar{t} + 6)SH]$ and $[(24\bar{t} + 6)L + (12\bar{t} + 6)S + (24\bar{t} + 6)SH]$ compared with Algorithm 1 and Algorithm 2. Since 32-bit word size and $GF(2^{163})$ are used, $\bar{t}$ is equal to 6 ($= \lceil \frac{163}{32} \rceil$). Therefore, Algorithm 4 saves $(294L + 222S + 150SH)$ (resp. $150L + 78S + 150SH$) instructions compared with Algorithm 1 (resp. Algorithm 2). These savings from Algorithm 4 contributes to 28.53% (18.47%) of reduced running time in comparison with Algorithm 1 (resp. Algorithm 2).

On the grounds of this counting, we expect that the proposed method is more promising as the used word size is increased. Thus, it can be efficiently used for more powerful state-of-art sensor motes.

5. Experimental Results and Analysis

5.1 Analysis of Field Operations

Table 4 compares Algorithm 1, Algorithm 2 and Algorithm 4. The proposed algorithm saves 15.06% and 5.12% of running times compared with Algorithm 1 and Algorithm 2. In section 4.1.2, we estimate that the proposed Algorithm 4 contributes to round 21.26% (resp. 7.91%) of improvement as compared to Algorithm 1 (resp. Algorithm 2). However, we got 15.06% (resp. 5.12%) of saving. There are some gaps between high level estimation and actual implementation. For example, we did not count the loop counter and function call overhead. In addition, each instruction may consume different clock cycles according to address mode, the number of involved operands, etc. Above all things, there is an important reason. This is originated from the limitation of Tmote Sky sensor mote. Namely, MSP430 processor does not support word-level instructions at odd address. It supports word-level instructions for data on even address (For data on odd address, only byte-level instructions are operated). For this reason, we can not implement the proposed algorithm with sorely C language. In other words, after incrementing the address of the C as 8-bit, word-level instruction is not properly operated. In our experience, when the word-level instruction is applied for odd address, the address is automatically converted into even address. Therefore, we use inline assembly code for implementing partial multiplications. For example, partial multiplications in $\mathbb{A}$ and $\mathbb{B}$ blocks are implemented as byte-level instructions, because the address of C is incremented when processing $\mathbb{A}$ and $\mathbb{B}$ blocks. The partial multiplications in $\mathbb{A}$ and $\mathbb{B}$ block are implemented with byte-level instructions while those in $\mathbb{C}$ and $\mathbb{D}$ blocks are implemented with word-level instructions (These fined-grained control is possible with only inline assembly code). With this limitation of MSP430 processor, the efficiency of the proposed technique is attenuated. Therefore, with these reasons, we got 15.06% (resp. 5.12%) of actual performance gain instead of 21.26% (resp. 7.91 %) from theoretic analysis. Furthermore, we think that the comparisons between Algorithm 4 partly implemented with inline assembly code and Algorithm 1, 2 sorely implemented with C langulage are fair, because of the limited address manipulation of MSP430 processor.

5.2 Performance Confirmation

We can confirm the data in Table 5 from the running times of multiplication and squaring. Let assume the used window size is 4, namely 4TNAF is used[†]. Since TinyECCK16 uses the mixed coordinate system for (`ECADD`), operations of $(8M + 3S)$ is required for a `ECADD` (Each of M and S mean

[†]Here, the window size means the used window for computing scalar multiplication such as using wNAF and wTNAF. It is different from the window used in ltr comb method.

Table 5 The Performance of TinyECCK16. Init and SM means times consumed for the initialization time including building precomputation tables and scalar multiplication, respectively. Field reduction is conducted using Algorithm 3. Computation time and memory are measured by secs and bytes, separately.

		2TNAF	3TNAF	4TNAF	5TNAF
Algorithm 1	Init	0.000412	0.052207	0.157923	0.359093
	SM	1.200966	0.966194	0.79755	0.711424
	Sign	1.367621	1.105864	0.933709	0.844602
	Verify	2.39366	1.914131	1.594994	1.411898
	ROM Size	12,356	12,676	12,896	13,492
	RAM Size	406	598	982	1,750
Algorithm 2	Init	0.000413	0.051515	0.156613	0.352845
	SM	1.108494	0.898171	0.743177	0.664932
	Sign	1.315236	1.030500	0.873858	0.795766
	Verify	2.21161	1.784629	1.482887	1.317995
	ROM Size	12,586	12,922	13,126	13,722
	RAM Size	406	598	982	1,750
Algorithm 4	Init	0.000412	0.051515	0.155908	0.350419
	SM	1.059872	0.863076	0.714164	0.640614
	Sign	1.224091	0.997892	0.848927	0.773319
	Verify	2.112158	1.699456	1.423765	1.268273
	ROM Size	13,286	13,606	13,826	14,422
	RAM Size	406	598	982	1,750
(Alg. 1 - Alg. 4)/(Alg. 1) (%)	SM	11.75	10.67	10.46	9.95
	Sign	10.49	9.76	9.08	8.44
	Verify	11.76	11.22	10.74	10.17
(Alg. 2 - Alg. 4)/(Alg. 2) (%)	SM	4.39	3.91	3.90	3.66
	Sign	6.93	3.16	2.85	2.82
	Verify	4.50	4.77	3.99	3.77

Table 6 Comparison the running time of existing software implementations (TinyECCK16, [1], [10] and [15])) (w under TinyECCK16 means the window size during wTNAF-scalar multiplication, all times are measured by secs).

	TinyECCK16 ($w = 4$)	TinyECCK16 ($w = 5$)	TinyECC [1]	[10]	NanoECC [15] ($GF(2^m)$)	NanoECC [15] ($GF(p)$)
SM	0.71	0.64	1.48	1.44	1.04	0.72
Sign	0.85	0.77	1.58	1.60	-	-
Verify	1.42	1.27	2.02	3.32	-	-
ROM Size	13,826	14,422	14,708	57,754	32,870	32,051
RAM Size	982	1,750	1,602	1,741	2,867	2,969

field multiplication and field squaring). With this knowledge, we can count the cost of a scalar multiplication as $[33\cdot(8M+3S)+162\cdot 3S] = (264M+585S)$ ($33 = \lceil\frac{163}{5}\rceil$, elliptic curve point doubling (ECDBL) operation is replaced with 3 squarings in wTNAF representation). Thus, the running time of a scalar multiplication using 4TNAF is computed as $(264 \cdot 0.00178422 + 585 \cdot 0.00023538) = (0.6087)sec$. The estimated result is almost 0.1 secs less than real data (0.7141 secs), however, this result can be admitted considering the overhead of function calls and loop countings.

Table 5 shows the running time of TinyECCK16 and its improvement ratio when Algorithm 4 is used instead of Algorithm 1 and Algorithm 2. Algorithm 4 saves 2.82–6.93% (8.44–11.76%) of running times in elliptic curve operations compared with Algorithm 2 (resp. Algorithm 1). Because elliptic curve operations using larger window sized-TNAF representation require less ECADD and ECDBL, namely less field multiplications, the efficiency of the proposed algorithm is attenuated.

5.3 Performance Comparisons

The code size of softwares is crucial because RAM and ROM size on sensor motes are very limited (see Table 2). Table 6 shows the existing ECC softwares in regards of execution time and memory requirement. TinyECC, and [10] are based on $GF(p)$. NanoECC provides the implementation of ECC over both $GF(p)$ and $GF(2^m)$. TinyECC, [10] and NanoECC over $GF(p)$ utilize many optimization techniques such as hybrid multiplication algorithm, pseudo-Mersenne prime, Jacobian coordinate, and sliding window scalar multiplication (fixed-based comb method in case of NanoECC). NanoECC over $GF(2^m)$ is based on Koblitz curve and uses Karasutba-Ofman multiplication algorithm and fast reduction algorithm using pentanomial. Because NanoECC uses MIRACL [19] for cryptographic operations, it is not optimized for sensor platform with respect to code size. In case of [10], since the code size of SHA-1 is 30-Kbyte, it requires much larger code size compared with

TinyECCK16 and TinyECC. TinyECCK16 is implemented with considering the limited memory of sensor motes. Thus, it requires less code size compared with other implementations. Since TinyECCK16 uses wTNAF-based scalar multiplication, it requires 982-byte and 1,750-byte of RAM when 4TNAF and 5TNAF are applied, respectively. Because TinyECCK16 stores additional negative points such as $\{-P, -3P, \ldots, -((2^{w-1}) - 1)P\}$ for efficient scalar multiplication, it can be further optimized by storing only positive points at the expense of little performance degrade.

TinyECCK16 is the fastest implementation of ECC on the Tmote Sky sensor mote compared with other implementations over both $GF(p)$ and $GF(2^m)$. It takes 0.64 secs to compute a scalar multiplication when 5TNAF is applied. NanoECC over $GF(p)$ provides comparable performance, however it requires much larger memory. Actually, the performance in [10] is measured when it run on 4 MHz clock. Since Tmote Sky mote can run on both 4 MHz and 8 MHz, we can expect that the running time when [10] operates on 8 MHz is going to be half of the time measured on 4 MHz. However, TinyECCK16 is more efficient than [10] with respect to running time and memory consumption even though [10] runs on 8 MHz.

6. Concluding Remarks

In this paper, we show the technique proposed in [14] become less efficient on extended word size such as 16-bit and 32-bit and describe how to improve the performance of ltr comb method on this environment. The proposed method can make the number of shifting the intermediate result C of $2t$ words only one time. It saves two times and six times of shifting the C when 16-bit and 32-bit word is used, respectively. The proposed technique saves 15.06% (resp. 5.12%) of running time compared with typical ltr comb method (resp. the algorithm proposed in TinyECCK). This improvement contributes to saving of 8.4–11.8% (resp. 2.82–6.93%) of the running times of elliptic curve operations such as a scalar multiplication, a signing and verification compared with typical ltr comb method (resp. TinyECCK's multiplication algorithm).

We have shown that the proposed algorithm will be more promising when it is implemented on more extended word size such as 32-bit through theoretic analysis. TinyECCK16 using 5TNAF computes a scalar multiplication within 0.64 secs and it generates (resp., verifies) a signature within 0.77 (resp., 1.26 secs) within 14,422-byte of ROM and 1,750-byte of RAM. This result is better in regard of execution time and memory requirement than existing ECC softwares implemented on 16-bit Tmote Sky sensor mote.

Acknowledgement

This work was supported by the second stage of the Brain Korea 21 Project. And this work was supported by the new faculty research program 2009 of Kookmin Universtiy in Korea.

References

[1] A. Liu, P. Kampanakis, and P. Ning, "TinyECC: Elliptic curve cryptography for sensor networks (Version 1.0)," available at "http://discovery.csc.ncsu.edu/software/TinyECC/," Nov. 2007.

[2] C. Karlof, N. Sastry, and D. Wagner, "TinySec: Link layer security architecture for wireless sensor networks," Proc. 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, Sept. 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," First IEEE International Conference on Sensor and Adhoc Communications and Networks (SECON04), pp.71–80, 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, pp.88–93, INSTICC Press, Miami, USA, May 2005.

[8] 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.

[9] H. Wang, B. Sheng, and Q. Li, "Elliptic curve cryptography-based access control in sensor networks," International Journal of Security and Networks (IJSN), vol.1, no.3/4, pp.127–137, 2006.

[10] H. Wang and Q. Li, "Efficient implementation of pulbic key cryptosystems on mote sensors," ICICS 2006, LNCS 4307, pp.519–528, 2006.

[11] 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.

[12] H. Eberle, A. Wander, N. Gura, S.C. 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), pp.343–349, 2005.

[13] O. Arazi and H. Qi, "Load-balanced key establishment methodologies in wireless sensor networks," International Journal of Security and Networks (IJSN). Special Issue on Security Issues on Sensor Networks 1, no.3/4, pp.158–166, 2006.

[14] S.C. Seo, D.-G. Han, H.C. Kim, and S. Hong, "TinyECCK: Efficient elliptic curve cryptography implementation over $GF(2^m)$ on 8-bit micaz mote," IEICE Trans. Inf. & Syst., vol.E91-D, no.5, pp.1338–1347, May 2008.

[15] 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.

[16] J. Solinas, "Efficient arithmetic on koblitz curves," Des. Codes Cryptogr., vol.19, pp.195–249, 2000.

[17] J. López and R. Dahab, "Improved algorithms for elliptic curve arithmetic in $GF(2^n)$," Selected Areas in Cryptography (SAC'98), LNCS 1556, pp.201–212, 1999.

[18] J. López and R. Dahab, "High-speed software multiplication in $\mathbb{F}_{2^m}$," Progress in Cryptology – INDOCRYPT 2000, LNCS 1977, pp.203–212, 2000.

[19] M. Scott, "MIRACL — A multiprecision integer and rational arith-

metic C/C++ library," Shamus Software Ltd, Dublin, Ireland (2003), available at "http://www.shamus.ie"

[20] MICAz Hardware Description, available at "http://www.xbow.com/Products"

[21] TinyOS Forum, available at "http://www.tinyos.net/"

[22] Tmote Sky Hardware Description, available at "http://www.sentilla.com/pdf/eol/tmote-sky-datasheet.pdf"

[23] Imote2 Datasheet, available at "http://www.xbow.jp/imote2.pdf"

[24] ATmega128L Processor Datasheet, available at "http://www.ortodoxism.ro/datasheets/2467S.pdf"

[25] TI MSP430 Processor Datasheet, available at "http://www.chipdocs.com/pndecoder/datasheets/TI/MSP430.html"

[26] Intel PXA27x Processor Family Datasheet, available at "http://www.xscale.jp/XSDoc/PXA27X/28000304.pdf"

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

Seog Chung Seo received the B.S. in Information & Computer Engineering from Ajou University, Suwon, Korea and the M.S. in Information and Communications from Gwangju Institute of Science and Technology (GIST), Gwangju, Korea in 2005 and 2007, respectively. He is working toward the Ph.D. degree in Graduate School of Information Management and Security. His research interests include cryptography and its efficient implementations.

Dong-Guk Han received his B.S. degree in mathematics from Korea University in 1999, and his M.S. degrees in mathematics from Korea University in 2002, respectively. He received Ph.D. of engineering in Information Security from Korea University in 2005. He was a Post.Doc. in Future University-Hakodate, Japan. After finishing the doctor course, he had been an exchange student in Dep. of Computer Science and Communication Engineering in Kyushu University in Japan from Apr. 2004 to Mar. 2005. He was a senior researcher in Electronics and Telecommunications Research Institute (ETRI), Daejeon, Rep. of Korea. He is currently working as an assistant professor with the Department of Mathematics of Kookmin University, Seoul, Rep. of Korea. He is a member of KIISC, IEEK, and IACR.

Seokhie Hong received Master's and Doctoral degrees from Korea University, Korea, in 1998 and 2000, respectively. He was also working as a postdoctoral researcher of Katholieke Universiteit Levuen, Belgium from April 2004 to February 2005. He is now an associate professor of Graduate School of Information Management and Security at Korea University and as an editorial staff of Journal of Korea Institute of Information Security and Cryptology. His research interests include digital forensic, cryptography, public and symmetric cryptosystems.