IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received September 5, 2018, accepted October 23, 2018, date of publication October 30, 2018, date of current version December 3, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2878777

Highly Efficient Implementation of NIST-Compliant Koblitz Curve for 8-bit AVR-Based Sensor Nodes

SEOG CHUNG SEO[1], (Member, IEEE), AND HWAJEONG SEO[2]
[1]The Affiliated Institute of ETRI, Daejeon 305-600, South Korea
[2]IT Convergence Division, Hansung University, Seoul 02860, South Korea

Corresponding author: Hwajeong Seo (hwajeong84@gmail.com)

This work was supported by ETRI.

ABSTRACT This paper presents an efficient implementation of elliptic curve cryptography (ECC) over the National Institute of Standards and Technology (NIST) K-233 curve for 8-bit AVR microcontrollers commonly used for sensor nodes in wireless sensor networks. Until now, several ECC implementations over NIST-compliant curves have been presented on 8-bit sensor nodes. However, most of them do not provide 112-bit security level currently recommended by NIST. Although some works provide more than 112-bit security level, their performance needs to be improved in order to be executed properly on resource-constrained sensor nodes. For optimizing the performance of ECC, we focus on the efficiency of field arithmetics and propose several optimization techniques. First, we present a novel polynomial multiplication technique based on multiplier encoding. The proposed method significantly reduces the required number of registers for a multiplier, which allows the larger block size for the Karatsuba Block-Comb method. The proposed method provides around 17.05% of improvement compared with the best result previously presented. Second, we optimize modular squaring and reduction algorithms considering the features of 8-bit AVR, and each of them provides around 21.86% and 3.7% improvements compared with the related works. With proposed methods, we present two versions of ECC implementation: (highly fast) *HF* and (highly secure) *HS* over NIST K-233 curve on an 8-bit ATmega128. Especially, *HF* version outperforms the best result previously implemented on the same curve by 18.6% and 34.5% for a variable and a fixed-based scalar multiplication, respectively. Furthermore, on the 8-bit AVR platform, our ECC implementation shows the best performance compared with other existing implementations over both NIST-standardized prime or binary curves.

INDEX TERMS Polynomial multiplication, elliptic curve cryptography, software implementation, simple power analysis, wireless sensor networks, NIST curves, 8-bit AVR, Koblitz curves.

I. INTRODUCTION

Wireless sensor networks (WSNs) are ad-hoc networks composed mainly of hundreds or thousands of tiny sensor nodes and they have been widely used for various applications such as monitoring services, battlefield reconnaissance, emergency rescue operations, and so on, in the recent past. Sensor nodes use wireless channel which is considered to be easy to eavesdrop or change messages and WSNs are often deployed in unattended environments where they can be easily captured or compromised by adversaries. That is why security mechanism is an integral factor for ensuring reliable and secure services on WSNs. However, achieving security on WSNs is not easy because sensor nodes in WSNs are battery-powered and very resource-constrained with respect to computing and memory capabilities. For example, MICAz mote, one of the most widely used sensor motes, equips an 8-bit AVR ATmega128 microcontroller, which is clocked at 7.3728 MHz and has 4Kbytes of RAM and 128Kbytes of ROM.

In early days, it had been thought that Public-Key Cryptosystems (PKCs) were infeasible in resource-constrained sensor nodes and thus, many symmetric-key-based security protocols were proposed for securing communications on WSNs. However, they have several limitations such as inefficient key management, complex key exchange structure, and so on. Thus, many researches have been tried to apply PKCs such as RSA, DSA, and elliptic curve cryptography (ECC)

for secure key establishment and efficient key management to WSNs. ECC has been regarded as a better choice for WSNs than conventional PKCs such as RSA, DSA, and DH (Diffie-Hellman), because it provides equivalent security with much smaller key sizes. For example, the security level of 160-bit key in ECC is equivalent to that of 1024-bit key in RSA. Thus, using ECC results in lower memory and bandwidth consumption on WSNs. Furthermore, through many researches, it is known that a scalar multiplication, the most expensive part of all ECC-based protocols, takes only 5% to 10% of the execution time of a modular exponentiation needed for RSA at the same security level [1], [2].

Recently, many ECC implementations over NIST-compliant curves on 8-bit sensor motes have been reported in the literature and their main purpose is to optimize the performance of a scalar multiplication, which is the most time-consuming part of ECC-based protocols [1]–[12]. Even though they could prove the feasibility of using ECC on 8-bit AVR microcontrollers by making full use of well-known optimization algorithms at each level of ECC implementation hierarchy (We will cover the detail about the ECC implementation hierarchy in Section II), most of them do not provide 112-bit security level currently recommended by U.S. NIST (National Institute of Standards and Technology). According to the recommended key sizes from NIST [13], [14], ECC implementations providing at least 112-bit security should be used from 2016 to 2030. Therefore, we present an optimized ECC implementation over NIST-compliant K-233 curve providing 112-bit security on the popular 8-bit AVR ATmega128 microcontroller. To optimize the performance of the scalar multiplication, we focus on developing efficient field arithmetic algorithms in polynomial multiplication, modular squaring, and reduction. In particular, since the polynomial multiplication occupies almost 80% of a scalar multiplication, it is crucial to optimize the performance of the polynomial multiplication for efficiency.

A. ECC IMPLEMENTATIONS ON 8-bit AVR MICROCONTROLLERS

In the past decades, several ECC implementations have been presented on 8-bit AVR microcontrollers. They can be divided into two types of implementation: one over prime curves and the other over binary curves. In this section, we use *cc* notation for clock cycles timing.

1) ECC IMPLEMENTATIONS OVER PRIME CURVES

Gura *et al.* [3] presented the first implementation results of prime field ECC and RSA on 8-bit AVR microcontrollers. The work demonstrated that if traditional PKCs were efficiently implemented considering the characteristics of target devices, they can be feasible for resource-constrained devices. For optimizing the performance of PKCs, they focused on the underlying field multiplication occupying around 80% of PKC operation such as an exponentiation in RSA or a scalar multiplication in ECC. Then they proposed the new hybrid multiplication method, which makes full use of available registers on 8-bit AVR by combining the row-wise and the column-wise multiplication methods. Gura *et al.*'s implementation over NIST P-224 curve could compute a scalar multiplication within 17,520,000 *cc* on 8-bit AVR microcontroller clocked at 8MHz. TinyECC [4] was the first well-known classic cryptographic library for 8-bit AVR-based, 16-bit MSP-based, and 32-bit ARM-based sensor nodes. TinyECC supports several NIST-recommended elliptic curve domain parameters, ranging from P-128 to P-192. For efficiency, TinyECC applied several well-known optimization techniques such as hybrid multiplication for field multiplication and sliding window methods for scalar multiplication. In case of P-192, TinyECC could compute a scalar multiplication within 21,381,120 *cc*. Uhsadel *et al.* [5] presented an estimated timing of 5,603,328 *cc* for computing a scalar multiplication over P-160 curve on an 8-bit AVR microcontroller running at 7.3728MHz. The latest work regarding ECC implementation over NIST prime curve on 8-bit AVR microcontroller is from [1] and [2]. Namely, Liu *et al.* implemented all primitives for ECDH and ECDSA protocols over NIST P-192 on 8-bit ATmega128 processor and got timings of 3,460,000 *cc* and 8,620,000 *cc* for a fixed-base scalar multiplication and a variable-base scalar multiplication, respectively.

Recently, several new types of ECC curves such as Curve25519 [15], MoTE Curve [16], [17], Kummer surface [18], and Four$\mathbb{Q}$ [19] have been developed for high security level. Hutter and Schwabe [20] presented the first result of the Networking and Cryptography library (NaCl) on the 8-bit AVR family of microcontrollers. They included Curve25519 curve providing 128-bit security as an ECC primitive in NaCl, and a scalar multiplication on the curve consumed 22,791,579 *cc*. DÃijll *et al.* also optimized X25519 key-exchange protocol over Curve25519 for 8-bit AVR, 16-bit MSP430X, and 32-bit ARM Cortex-M0 microcontrollers, and they achieved 13,900,397 *cc* for 8-bit ATmega2560 processor [21]. Renes *et al.* presented μKummer implementation on 8-bit AVR ATmega and got a timing of 9,513,536 *cc* for a full scalar multiplication. In [17], Liu *et al.* presented ECC implementations over MoTE curves [16], the next generation lightweight curves, on 8-bit AVR and 16-bit MSP processors. On 8-bit ATmega128 processor, their implementation over P223 MoTE curve consumes 6,048,000 *cc* and 12,879,000 *cc* for a fixed-base scalar multiplication and a variable-base scalar multiplication. Among the new types of aforementioned curves on 8-bit AVR platforms, Four$\mathbb{Q}$ provides the best performance. Liu *et al.* presented highly optimized Four$\mathbb{Q}$ implementation on 8-bit AVR ATxmega256A3 microcontroller and their implementation consumed 3,007,300 *cc* and 6,561,500 *cc* for a fixed-base scalar multiplication and a variable-base scalar multiplication [22].

In summary, with regard to NIST prime curves providing at least 112-bit security on 8-bit AVR platforms, Liu *et al.*'s work over NIST P-192 curve provides the best performance as 3,460,000 *cc* and 8,620,000 *cc* for a fixed-base

scalar multiplication and a variable-base scalar multiplication, respectively [2]. Regarding the new types of curves, Four$\mathbb{Q}$ implementation provides the best performance as 3,007,300 *cc* and 6,561,500 *cc* for each of a fixed-base and a variable-base scalar multiplication [22].

2) ECC IMPLEMENTATIONS OVER BINARY CURVES

Compared to ECC implementations over prime curves on 8-bit AVR platforms, the number of ECC implementations over binary curves is relatively small. This is because the most time-consuming operation in ECC operations is polynomial multiplication, and 8-bit AVR does not provide a generic carryless multiplier for polynomial multiplication.[1] Therefore, multiplication over binary fields is much slower than that over prime fields on 8-bit AVR platforms. Malan *et al.* [6] implemented ECC over 163-bit binary curve for ECDH key agreement protocol. However, their implementation showed that the computation cost of 251,862,220 *cc* was expensive for the embedded processors. Yan and Shi implemented ECC over $GF(2^{163})$ and it computed a scalar multiplication over 163-bit binary curve in 111,183,513 *cc* [7], which was still high to be used on 8-bit AVR. Eberle *et al.* implemented ECC over $GF(2^{163})$ in Assembly language and obtained a timing of 331,20,000 *cc* for computing a scalar multiplication [8]. NanoECC showed that a scalar multiplication over K-163 curve could be computed in 15,925,248 *cc* on 8-bit MICAz sensor clocked at 7.3728MHz [9]. For efficiency of polynomial multiplication, they utilized López-Dahab's LUT-based polynomial multiplication method (so-called *wLtR* comb method) and Karatsuba method [23]. Seo *et al.* [10] presented TinyECCK, which is the first ECC implementation over K-163 Koblitz curve on 8-bit ATmega128. They showed that the ECC implementation over binary Koblitz curve can be faster than that over prime curve if it is optimized properly by improving the performance of field arithmetics such as polynomial multiplication and reduction. For efficiency of polynomial multiplication, they optimized *wLtR* comb method by reducing the number of redundant memory accesses, and they replaced computation of *ECDBL* with cheap field squarings by utilizing the property of Koblitz curves. Finally, they achieved a timing of 8,404,992 *cc* for a scalar multiplication. Kargl *et al.* implemented a scalar multiplication over $GF(2^{167})$ optimal extension field (OEF) with Montgomery Ladder for SPA-resistance and got a timing of 6,100,462 *cc* on an 8-bit AVR clocked at 8MHz [24]. Aranha *et al.* presented highly optimized ECC implementations over Koblitz curves [11]. They implemented K-163 and K-233 Koblitz curves on an 8-bit ATmega128 processor clocked at 7.3728MHz, and they achieved timings of 2,138,112 *cc* (resp. 4,866,048 *cc*) and 2,359,296 *cc* (resp. 5,382,144 *cc*) for a fixed-base scalar multiplication and a variable-base scalar multiplication over K-163 curve (resp. K-233 curve). For efficiency, they introduced a rotating register mechanism for López *et al.*'s LUT-based polynomial multiplication. Seo *et al.* [12] optimized the polynomial multiplication over $GF(2^{163})$ with their proposed Karatsuba Block-Comb (*KBC*) multiplication method, and achieved a timing of 2,138,112 *cc* for a variable-base scalar multiplication over K-163 Koblitz curve on 8-bit AVR.

In summary, with respect to K-163 curve providing 80-bit security level on 8-bit AVR platforms, Seo *et al.*'s work provides the best performance as 2,138,112 *cc* for a variable-base scalar multiplication [12]. Regarding to K-233 curve providing 112-bit security level, Aranha *et al.*'s work achieved the best timing result as 4,866,048 *cc* and 5,382,144 *cc* for a fixed-base scalar multiplication and a variable-base scalar multiplication, respectively [11].

B. OUR CONTRIBUTIONS

The contributions of this paper are summarized as follows:

- *Presenting a novel polynomial multiplication.* We propose a new method which can significantly improve the performance of Block-Comb (*BC*)-based polynomial multiplication. Through the proposed multiplier encoding, the proposed multiplication method significantly reduces the required number of registers for a multiplier from $O(n)$ to $O(1)$. The method scans the multiplier in bit-wise rather than traditional byte-wise, which increases the size of block and significantly reduces the number of partial products. The method is also well integrated with Karatsuba method. Finally, the polynomial multiplication over $GF(2^{233})$ requires only 6,896 and 6,248 clock cycles for w/ and w/o encoding, which achieves 17.05% and 24.84% of improvements compared with the best result previously presented, respectively.
- *Optimized Modular Squaring and Reduction.* We also present optimized implementations of modular squaring and reduction for $GF(2^{233})$ on 8-bit AVR. The proposed modular squaring and reduction could achieve around 21.86% and 3.7% of improvements compared with the previous best results over $GF(2^{233})$.
- *Developing two versions of HF (Highly-Fast) and HS (Highly-Secure) ECC implementation.* We present two versions of ECC implementation. *HF* version focuses on performance and it outperforms the previous best results on the same curve by 18.6% and 34.5% for a variable and a fixed-based scalar multiplication, respectively. The *HS* ensures security against Simple Power Analysis (SPA) and Differential Power Analysis (DPA) by using regular *w*TNAF method and randomized coordinate system, respectively.

The remainder of this paper is organized as follows. In Section II, we briefly introduce the basic of ECC including Koblitz curves and its implementation hierarchy, and we also describe the main features of the 8-bit AVR ATmega128 processor. Section III overviews the previous state-of-art

[1] Recently, a carryless multiplier for polynomial multiplication has been included in Intel processors and ARM processors. However, until now, 8-bit AVR does not include such a multiplier.

implementations of polynomial multiplication over 8-bit AVR platforms. In Section IV, we describe the proposed field arithmetic algorithms, especially the proposed polynomial multiplication technique, optimized for 8-bit AVR processors. Section V describes the rationale for the selection of proper algorithms at an elliptic curve operation level. Section VI describes our approaches for SCA security. Section VII compares our implementation with other existing ECC implementations over 8-bit AVR platforms. Finally, we conclude the paper with some future works in Section VIII.

II. PRELIMINARIES

In this section, we describe the overview of ECC over $GF(2^m)$, its implementation hierarchy, and the characteristics of 8-bit AVR microprocessors.

A. ELLIPTIC CURVE CRYPTOGRAPHY AND KOBLITZ CURVES

Elliptic Curve Cryptography (ECC) [25]–[27], introduced by Neal Koblitz and Victor S. Miller in 1985, is one of the most widely used Public Key Cryptosystems (PKCs). The security of ECC is based on the hardness of ECDLP (Elliptic Curve Discrete Logarithm Problem), and it is believed that ECC provides an equivalent security level of existing PKCs such as RSA and DSA with a much smaller key size.[2] An elliptic curve E over a field $\mathbb{K}$ is a set of solutions $(x, y) \in \mathbb{K} \times \mathbb{K}$ which satisfies following *Weierstrass equation*

$$E/\mathbb{K} : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \quad (1)$$

where a_1, a_2, a_3, a_4, $a_6 \in \mathbb{K}$ and the curve *discriminant* is $\Delta \neq 0$; together with a *point at infinity* denoted by $\mathcal{O}$. If $\mathbb{K}$ is $GF(2^m)$, then the curve is called a binary curve. Koblitz curves [25], [26], [28] are special curves among binary curves, and they are defined as a following equation.

$$y^2 + xy = x^3 + ax^2 + 1 \quad (2)$$

where $a \in \{0, 1\}$.

Given an elliptic point $P = (x, y) \in GF(2^m)$ and an integer k, the operation $k \cdot P$ is called *scalar multiplication* (*SM*). Since *SM* is a computationally dominant operation in ECC-based security protocols, such as ECDH key agreement, ECIES encryption, and ECDSA signature algorithms, it needs to be computed efficiently. Computing $k \cdot P$ ($\sigma_{i=0}^{m-1} k_i 2^i P$ where k_i is i-th bit of k) is composed of two types of elliptic curve operations: *Elliptic Curve Point Addition* (*ECADD*) which adds two different points such as $(P_1 + P_2)$ and *Elliptic Curve Point Doubling* (*ECDBL*) which doubles a point such as $(2P_1)$.[3] Unlike ordinary binary curves, Koblitz curves have an advantage such that *ECDBL* can be replaced by efficiently computable Frobenius map $\tau(x, y) = (x^2, y^2)$, $\tau(\infty) = \infty$ with τ expansion of scalar k. Thus, the performance of SM on Koblitz curves can be much improved compared with that on ordinary binary curves.

We target on the efficient ECC implementation over K-233 Koblitz curve recommended by NIST standard. This curve provides at least 112-bit security satisfying the current key size recommended by NIST [13], [14]. Indeed, K-233 curve is currently included in approved algorithm list of NIST CAVP (Cryptographic Algorithm Validation Program) [29] in CMVP (Cryptographic Module Validation Program) [30] and thus, it is widely used in certificated cryptographic modules including OpenSSL and BouncyCastle [31].

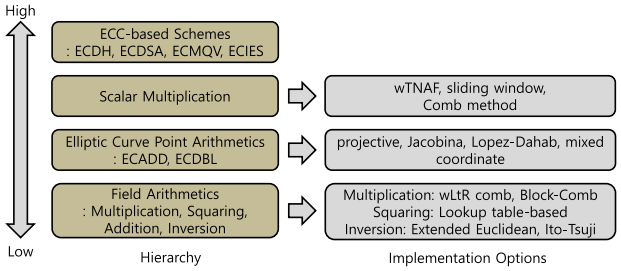


FIGURE 1. The implementation hierarchy of ECC over Koblitz curves.

B. IMPLEMENTATION HIERARCHY OF KOBLITZ CURVES

In order to implement ECC protocols, three levels of operations are required. Fig. 1 describes the implementation hierarchy of Koblitz curve-based ECC and its corresponding arithmetic algorithms. Since the selections of algorithms at each level directly affect the performance of ECC implementation, it is necessary to carefully select proper algorithms to implement them considering the features of target platforms in terms of instruction sets, memory sizes, and a length of register files (In Section V, we will describe our selection of proper algorithms for an optimal performance in detail).

A scalar multiplication (SM) which computes $k \cdot P$ for given a scalar k and point P is a computationally dominant operation in ECC-based security protocols. An SM requires elliptic curve point arithmetics such as *ECADD*s and *ECDBL*s. Many optimization algorithms including sliding window [32], Comb-based method [33], *w*TNAF [28] have been proposed for optimizing the performance of SM, and their aim is to reduce the number of *ECADD*s when computing an SM (Note that with *w*TNAF, *ECDBL* can be replaced by efficiently computable Frobenius map).

ECADD and *ECDBL* are composed of field arithmetics such as field multiplication, squaring, reduction, addition, and inversion. Especially, each of *ECADD* and *ECDBL* in an affine coordinate system requires operating a field inversion, and this is the most expensive operation in field arithmetics (It is known that $\frac{I}{M} > 30$ on 8-bit AVR platforms). To replace field inversion with other relatively cheap field arithmetics, efficient coordinate systems, such as projective,

[2] It is known that ECC using about 160-bit key provides equivalent security level to RSA using 1024-bit key.

[3] *ECDBL* is computed at each bit and *ECADD* is computed when the bit is set to 1 [26].

Jacobian, and López–Dahab (*LD*), and mixed coordinate systems [26], [34], have been proposed. Except for field inversion, field multiplication (polynomial multiplication in $GF(2^m)$) is the most expensive operation when computing an SM (it occupies almost 80% of computational overhead). Thus, the main purpose in this paper is to develop an efficient polynomial multiplication method by maximizing the use of available registers in 8-bit AVR platforms. Field squaring and reduction need to be carefully implemented because they are also one of the most frequently used operations in elliptic curve operations.

C. 8-bit AVR MICROCONTROLLER

Nowadays, 8-bit AVR microprocessors are widely used for various applications such as smartcards and sensor nodes in WSN. An 8-bit AVR, such as Atmel ATmega128, has 32 general-purpose registers $(R_{31}, \ldots, R_1, R_0)$ and six of them are used for memory address pointers (Each pair of (R_{26}, R_{27}), (R_{28}, R_{29}), and (R_{30}, R_{31}) are aliased as X, Y, and Z pointer register, respectively) [35]. The AVR microprocessors have separate memory areas and buses for program and data in a simple single-issue pipeline. It has a total of 133 instructions, and each instruction has a fixed latency. For example, the arithmetic/logical instructions (e.g. arithmetic add `ADD`, bit-wise XOR `EOR`, logical shift left `LSL` and so on) are executed in a single clock cycle, while memory access instructions (e.g. load `LD`, store `ST` and so on) take two clock cycles [35]. 8-bit AVR has limited computation and memory capabilities. For example, in case of 8-bit Atmega128, it has only 4Kbytes of RAM and 128Kbytes of ROM, and it runs at 7.3728MHz.

III. POLYNOMIAL MULTIPLICATION ON 8-bit AVR PLATFORMS

Since polynomial multiplication is the most performance-critical operation when computing an SM, many studies for optimizing an arithmetic on 8-bit AVR platforms have been conducted [10]–[12], [36]–[39]. They are mainly divided into two categories: LUT-based methods (known as wLtR Comb) [10], [11], [36], [37] and Block-Comb (BC)-based methods [12], [38], [39]. Since the number of available registers is limited on 8-bit AVR (only 26 general purpose registers, except for 6 memory address pointer registers, are available), a number of memory accesses occur when computing polynomial multiplication. For example, at least 120 registers are required to hold a whole set of a multiplicand, a multiplier, and a result of polynomial multiplication over $GF(2^{233})$. However, since only 26 registers are available, only some parts of operands can be maintained in the registers, which results in a number of redundant memory accesses. Thus, the main concern of researches on polynomial multiplication on 8-bit AVR microcontrollers is to minimize the number of redundant memory accesses by optimizing the use of available registers.

Before describing our proposed polynomial multiplication method, this section briefly describes existing polynomial multiplication methods and some notations on 8-bit AVR platforms.

A. POLYNOMIAL MULTIPLICATION AND SOME NOTATIONS

Polynomial multiplication is $A \cdot B$ where $A = \sum_{i=0}^{m-1} a_i z^i$, $B = \sum_{i=0}^{m-1} b_i z^i \in GF(2^m)$. A and B mean a multiplicand and a multiplier, and the result of polynomial multiplication C can be expressed as $C = \sum_{i=0}^{m-1} A \cdot b_i z^i$. The most basic polynomial multiplication algorithm is shift-and-add method. It scans the multiplier from 0-th bit to $(m-1)$-th bit. At each iteration, multiplicand A is left-shifted such as $A \cdot z$, and if the bit of multiplier B is set 1, then the left-shifted multiplicand is XORed with the accumulator (Namely, if i-th bit of the multiplier is set 1, then, $A \cdot z^i$ is XORed with the accumulator). Comb method, the basic algorithm for LUT-based methods and Block Comb method, improves the performance of polynomial multiplication by utilizing the fact that if $A \cdot z^k$ has been computed for some $k \in [0, W-1]$ (in case of 8-bit AVR platforms, W is 8), $A \cdot z^{Wj+k}$ can be easily obtained by appending j zero words to the right of the vector representation of $A \cdot z^k$. Thus, it can reduce the number of shift operations compared with the basic shift-and-add algorithm. There are two types of Comb methods: LtR Comb method and RtL Comb method. While LtR Comb method proceeds from MSB (Most Significant Bit) to LSB (Least Significant Bit), RtL Comb method operates from LSB to MSB [26], [36].

Throughout the paper, we will use following notations. R_i means the i-th general purpose register where $0 \leq i \leq 31$. The operators $\oplus$, $\ll$, and $\gg$ denote XOR, logical left shifts, and logical right shifts. $A[i]$ means the i-th byte (or word) of A and it is composed of eight bits like $(a_{8i+7}, \ldots, a_{8i})$. Finally, $A[i, \ldots, j]$ represents bytes (words) from $A[j]$ to $A[i]$, respectively.

B. LOOK-UP TABLE METHOD

Look-Up Table (LUT)-based polynomial multiplication, originally introduced by Hankerson *et al.* [26] and López and Dahab [36], is an extension of LtR Comb method (so-called wLtR Comb method), and it executes polynomial multiplication by w-bit unit rather than single bit unit at the expense of a precomputation table, which results in the reduction of the number of bit operations such as shift and XOR operations. It first builds a precomputation table about all possible results of $A \cdot u(z)$ for all polynomials $u(z)$ of degree at most $w-1$. Then, it scans multiplier B by w-bit unit at a time from MSB to LSB and takes the corresponding value from the precomputation table, and the value is XORed with the intermediate result without computing it. On 8-bit AVR platforms, it is known that 4-bit is the optimal width w for LUT-based method, which requires $16 \times m$-bit of RAM memory for accommodating a table composed of 16 polynomials from $0 \cdot A$ to $(z^3+z^2+z+1) \cdot A$. LUT-based method has been widely implemented on 8-bit AVR platforms [10], [11], [37]. Seo *et al.* implemented 163-bit polynomial multiplication with NesC language on

8-bit ATmega128 microcontroller, and they improved the original LUT-based method by reducing the number of redundant memory accesses by combining two iterations of the main loop into one [10]. Aranha *et al.* [11] and Oliveira *et al.* [37] introduced a rotating register mechanism which could significantly reduce the number of memory accesses for LUT-based polynomial multiplication method. They implemented the proposed method in Assembly language and achieved timings of 4,508 *cc*, 8,314 *cc*, and 11,727 *cc* for computing each polynomial multiplication over $GF(2^{163})$, $GF(2^{233})$, and $GF(2^{271})$, respectively. Even though LUT-based methods provide good performance, they are inherently vulnerable to side channel analysis (SCA) using memory-address information [40], [41] due to huge memory accesses. Furthermore, they require relatively large RAM consumption considering the limited RAM size of 4Kbytes on 8-bit AVR platforms.

C. BLOCK-COMB METHOD

As an alternative to LUT-based methods, Block-Comb (*BC*) method was firstly introduced in [38] for efficient polynomial multiplication of η_T pairing computation on 8-bit Atmega128 microcontroller. In *BC* method, a multiplier and a multiplicand are divided into equal-sized blocks of *s*-byte, and partial products of divided multiplicands and multipliers are computed in a column-wise fashion. In other words, in *BC* method, the available registers are divided into three parts: *s* registers for a block-sized multiplicand, *s* registers for a block-sized multiplier, and $2s+1$ for the result of partial products. Since the intermediate results are maintained in $2s+1$ working registers, the results of partial products belonging to the same column can be directly updated to the registers without accessing memory, which reduces the number of redundant memory accesses. Shirase *et al.* [38] concluded that the optimal block size *s* is 6 because $(4s+1) < 26$. The original *BC* computes a polynomial multiplication over $GF(2^{239})$ within 9,511 clock cycles (*cc*).

Seo *et al.* extended the size of block from 6 to 7 by suggesting Unbalanced Block-Comb method (*UBC*) for $GF(2^{163})$ multiplication [39]. They utilized the fact that the tested bits of a multiplier are no longer necessary during the process of a partial product, and recycled this register for holding the Most Significant Byte of the multiplicand. As a result, the extended block size reduces the number of partial products from 16 to 9 when computing a polynomial multiplication over $GF(2^{163})$ (Note that 7-word (resp. 6-word) block size divides 163-bit polynomial into three blocks (resp. four blocks)). Afterward, Seo *et al.* proposed Karatsuba Block-Comb method by combining Karatsuba technique with Block-Comb [12], which reduces the number of partial products further from 9 to 6 at the expense of several cheap field additions when computing a polynomial multiplication over $GF(2^{163})$.

From these researches, it is widely believed that the maximum block size in *BC*-base methods is 7 (56-bit). However, we need to extend the block size for efficient polynomial multiplication over larger fields for providing at least 112-bit security.

IV. PROPOSED FIELD ARITHMETIC ALGORITHMS

In this section, we describe our optimization strategies for efficient arithmetics in multiplication, squaring, and reduction over $GF(2^{233})$ on 8-bit AVR platform. Especially, with respect to polynomial multiplication, our method breaks the common idea that the maximum block size of *BC*-based methods on 8-bit AVR microcontrollers is 7-byte.

Algorithm 1 Block-Comb Method for 56-bit Polynomial Multiplication on 8-bit AVR Microcontroller (Sets of Registers $(R_{13}, \ldots, R_0)$, $(R_{21}, \ldots, R_{14})$, and $(R_{28}, \ldots, R_{22})$ Are Reserved for Holding an Accumulator *C*, a Multiplicand *A*, and a Multiplier *B*, Respectively)

Require: Two seven 8-bit operands *A* and *B*.
Ensure: $C(112\text{ bit}) = A \cdot B$.
1: **for** $l = 0$ to 13 **do**
2: Load $R_l \leftarrow 0$
3: **end for**
4: **for** $l = 0$ to 6 **do**
5: Load $R_{14+l} \leftarrow A[l]$
6: Load $R_{22+l} \leftarrow B[l]$
7: **end for**
8: $R_{21} \leftarrow 0$
9: **for** $l = 0$ to 7 **do**
10: **for** $m = 0$ to 6 **do**
11: **if** the l-th bit of $R_{22+m} == 1$ **then**
12: **for** $n = 0$ to 7 **do**
13: $R_{m+n} \leftarrow R_{m+n} \oplus R_{14+n}$
14: **end for**
15: **end if**
16: **end for**
17: **if** $l \neq 7$ **then**
18: $(R_{21}, \ldots, R_{14}) \leftarrow (R_{21}, \ldots, R_{14}) \ll 1$
19: **end if**
20: **end for**
21: Return *C*

A. PROPOSED POLYNOMIAL MULTIPLICATION METHOD

1) BLOCK-COMB METHOD WITH MULTIPLIER-ENCODING

Our proposed polynomial multiplication method is basically based on *BC* method, and its main idea is reordering the process of *BC* method. Alg. 1 describes the basic of *BC* method for 56-bit polynomial multiplication [12], [39].[4] The sets of registers are reserved for holding a multiplicand $(R_{21}, \ldots, R_{14})$, a multiplier $(R_{28}, \ldots, R_{22})$, and an accumulator $(R_{13}, \ldots, R_0)$. Step 4–7 load multiplicand *A* and multiplier *B* and Step 9–20 compute partial products with *RtL* (Right-to-Left) Comb fashion[5] [26]. In the main loop of the product, l-th bit of the registers holding the bit of the multiplier is tested from $l = 0$ to $l = 7$. In other words, l-th bit of each register $(R_{22+m}, 0 \leq m \leq 6)$ for the multiplier is tested and if the bit is set, the multiplicand is XORed with the accumulator through Step 12–14. In particular, the multiplier should be maintained in the set of registers, since every register in this set is scanned during each loop of the partial product

[4] In case of 233-bit polynomial multiplication, each of a multiplicand and a multiplier are divided into five 56-bit blocks and each of 25 partial products is computed with Alg. 1.

[5] *RtL* means that the partial product proceeds as scanning the multiplier from LSB to MSB.

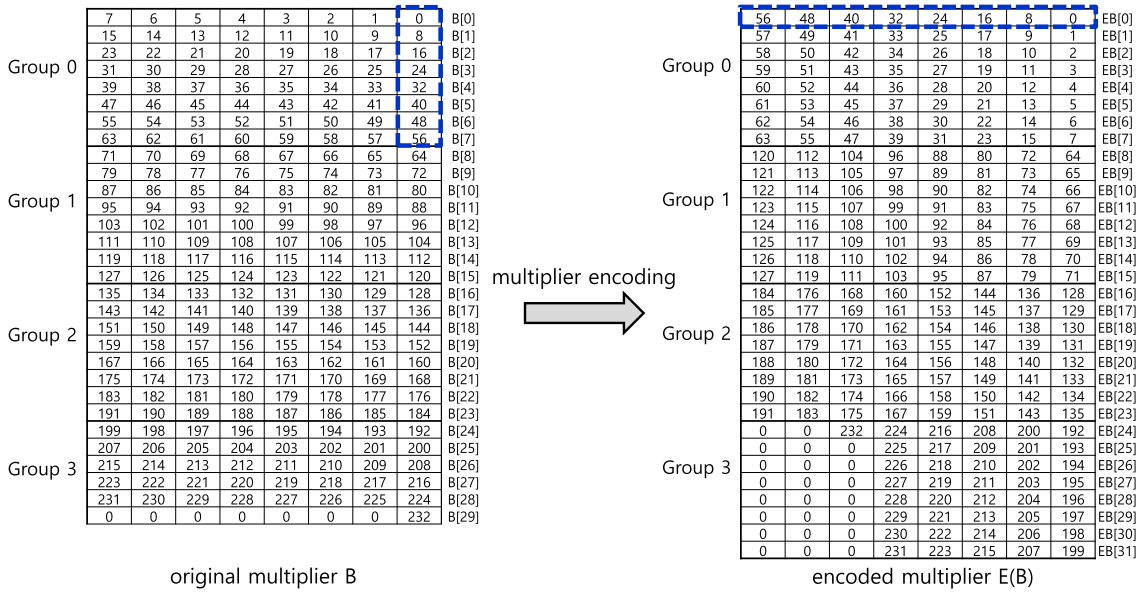


FIGURE 2. Proposed multiplier encoding(Left: an original multiplier, Right: the encoded multiplier).

computation. In the proposed method, on the other hand, only one register is required for maintaining the multiplier in the process of partial product through multiplier-encoding. Thus, the proposed Block-Comb method can extend the block size of *BC* method with the saved registers and significantly reduces the number of partial products required for computing polynomial multiplication.

In order to encode the bits of multiplier B, the following Equation 3 is required.

$$E(i) = \{[(i \mod 64) \times 8] \mod 63 + [(i \gg 6) \times 64]\} \quad (3)$$

The Equation moves the i-th bit of the multiplier to $E(i)$-th bit position in the encoded multiplier. Each bit column of eight-byte groups in the original multiplier is packed and transposed into a byte row in the encoded multiplier. On 8-bit AVR microcontroller, this encoding process can be efficiently implemented with *lsr* (logical shift right) and *ror* (rotate right) instructions. Alg. 2 shows the proposed encoding method over $GF(2^{233})$ on 8-bit AVR microcontroller.[6] The original 233-bit multiplier is divided into four groups, consisting of 8-byte, as on the left side of Figure 2. Each group ranging from 0 to 2 has 8-byte, but Group 3 has 6-byte because the multiplier is in $GF(2^{233})$, which only occupies 233-bit. This is why the encoding process for Group 3 is slightly different from that for Group 0, 1, and 2. In Alg. 2, each byte of a group is right-shifted for packing each bit column of bytes groups with a *lsr* instruction and the packed bits are held in R_8 register with a *ror* instruction. The encoded multiplier is stored in *EB* and the size of *EB* is slightly larger than *B* (The size of *EB* is 32-byte while the size of *B* is 30-byte). The right side of Figure 2 shows the result of the proposed multiplier encoding over $GF(2^{233})$.

[6]Since the proposed method is generic, the method can be applied to the other bit lengths and microprocessors with simple modifications.

Algorithm 2 Proposed Multiplier Encoding Over $GF(2^{233})$ on 8-bit AVR Microcontroller

Require: Multiplier B over $GF(2^{233})$.
Ensure: Encoded multiplier *EB*.
(Encoding from Group 0 to Group 2)
for $i = 0$ to 3 **do**
  Load $R_7, \ldots, R_0 \leftarrow B[8i+7, \ldots, 8i]$
  for $j = 0$ to 7 **do**
    clr R_8
    for $k = 0$ to 7 **do**
      lsr R_k
      ror R_8
    end for
    Store R_8 at $EB[8i+j]$
  end for
end for
(Encoding Group 3)
Load $R_5, \ldots, R_0 \leftarrow B[29, \ldots, 24]$
clr R_8
for $j = 0$ to 7 **do**
  clr R_8
  for $k = 0$ to 4 **do**
    lsr R_k
    ror R_8
  end for
  if $j == 0$ **then**
    lsr R_5
  end if
  for $k = 5$ to 7 **do**
    ror R_8
  end for
  Store R_8 at $EB[24+j]$
end for
return (*EB*)

In Alg. 3, the proposed Block-Comb multiplication algorithm for computing a 64-bit partial product is described. It computes a partial product with *RtL* Comb fashion using the encoded multiplier *EB*. In particular, it only uses a single register for holding target bits of the multiplier, since each byte of *EB* contains each bit column of the original

Algorithm 3 Proposed Block-Comb Multiplication on 64-bit With Multiplier Encoding in Register Level, Where $(R_{15}, \ldots, R_0)$, $(R_{24}, \ldots, R_{16})$, and R_{25} Are Reserved for an Accumulator, a Multiplicand, and a Multiplier

Require: Two eight 8-bit multiplicand A and encoded multiplier EB.
Ensure: $C(128\text{-}bit) = A \cdot B$.
(Initializing accumulator)
for $j = 0$ to 15 **do**
  $R_j \leftarrow 0$
end for
(Loading multiplicand)
for $l = 0$ to 7 **do**
  Load $R_{16+l} \leftarrow A[8j + l]$
end for
for $l = 0$ to 7 **do**
  $R_{25} \leftarrow EB[8k + l]$ // (Loading multiplier)
  for $m = 0$ to 7 **do**
    if the m-th bit of $R_{25} == 1$ **then**
      for $n = 0$ to 8 **do**
        $R_{m+n} \leftarrow R_{m+n} \oplus R_{16+n}$
      end for
    end if
  end for
  (Shifting multiplicand)
  if $l \neq 7$ **then**
    $(R_{24}, \ldots, R_{16}) \leftarrow (R_{24}, \ldots, R_{16}) \ll 1$
  end if
end for

multiplier's eight-byte groups (Alg. 3 reserves 16 registers, 9 registers, and a single register for holding an accumulator, a multiplicand, and a multiplier during the process of computing partial products, respectively). Thus, the proposed method reduces the number of required registers for holding a multiplier of n words from $O(n)$ to $O(1)$. With the proposed Block-Comb, a polynomial multiplication over $GF(2^{233})$ can be computed with sixteen 64-bit partial products.

2) INTEGRATION WITH KARATSUBA TECHNIQUE

The number of partial products is optimized again by applying Karatsuba method. Since the size of the proposed Block-Comb method is 8 words (64-bit) on 8-bit AVR microcontroller, the operands over $GF(2^{233})$ are divided into four-term polynomials so that a multiplicand $A(x) = A_3x^3 + A_2x^2 + A_1x^1 + A_0$ and a multiplier $B(x) = B_3x^3 + B_2x^2 + B_1x^1 + B_0$ where x is 2^{64}. The detailed computations are given in Equation 4.

$$\begin{aligned}
&A(x) \cdot B(x) \\
&\quad = (M_3x^3 + M_2x^2 + M_1x + M_0) \cdot (x^3 + x^2 + x + 1) \\
&\qquad + M_8x^3 + (M_5x^2 + M_4) \cdot (x^3 + x) \\
&\qquad + (M_7x + M_6) \cdot (x^3 + x^2) \\
&M_0 = A_0 \times_{64-bit} B_0, M_1 = A_1 \times_{64-bit} B_1, \\
&M_2 = A_2 \times_{64-bit} B_2, M_3 = A_3 \times_{64-bit} B_3, \\
&M_4 = (A_1 \oplus A_0) \times_{64-bit} (B_1 \oplus B_0), \\
&M_5 = (A_3 \oplus A_2) \times_{64-bit} (B_3 \oplus B_2), \\
&M_6 = (A_2 \oplus A_0) \times_{64-bit} (B_2 \oplus B_0), \\
&M_7 = (A_3 \oplus A_1) \times_{64-bit} (B_3 \oplus B_1), \\
&M_8 = (A_3 \oplus A_2 \oplus A_1 \oplus A_0) \times_{64-bit} (B_3 \oplus B_2 \oplus B_1 \oplus B_0) \quad (4)
\end{aligned}$$

Algorithm 4 Proposed Karatsuba Block-Comb Multiplication (A, B in $GF(2^{233})$)

Require: Binary polynomial $A = (A_{29}, \ldots, A_0)$, $B = (B_{29}, \ldots, B_0)$ where A, B in $GF(2^{233})$.
Ensure: $(C_{58}, \ldots, C_0) = AB$.
(Encoding Multiplier B by calling Alg. 2 and storing the encoded multiplier at EB)
1: $C = A[7 \cdots 0] \times_{64-bit} EB[7 \cdots 0]$
2: $C = C \oplus A[15 \cdots 8] \times_{64-bit} EB[15 \cdots 8]$
3: $C = C \oplus A[23 \cdots 16] \times_{64-bit} EB[23 \cdots 16]$
4: $C = C \oplus A[29 \cdots 24] \times_{64-bit} EB[31 \cdots 24]$
5: $C = C \oplus (C \ll 64) \oplus (C \ll 128) \oplus (C \ll 192)$
6: $C = C \oplus [(A[7 \cdots 0]) \oplus A[15 \cdots 8] \oplus A[23 \cdots 16] \oplus A[29 \cdots 24]) \times_{64-bit} (EB[7 \cdots 0]) \oplus EB[15 \cdots 8] \oplus EB[23 \cdots 16] \oplus EB[31 \cdots 24])]$
7: $T = (A[29 \cdots 24] \oplus A[23 \cdots 16]) \times_{64-bit} (EB[31 \cdots 24] \oplus EB[23 \cdots 16])$
8: $T = T \oplus [(A[15 \cdots 8] \oplus A[7 \cdots 0]) \times_{64-bit} (EB[15 \cdots 8] \oplus EB[7 \cdots 0])]$
9: $C = C \oplus (T \ll 64) \oplus (T \ll 192)$
10: $T = (A[29 \cdots 24] \oplus A[15 \cdots 8]) \times_{64-bit} (EB[31 \cdots 24] \oplus EB[15 \cdots 8])$
11: $T = T \oplus [(A[23 \cdots 16] \oplus A[7 \cdots 0]) \times_{64-bit} (EB[23 \cdots 16] \oplus EB[7 \cdots 0])]$
12: $C = C \oplus (T \ll 128) \oplus (T \ll 192)$
13: return (C)

Algorithm 5 Regular Window TNAF (wTNAF) Method [46]

Require: n-bit scalar $k = (k_{n-1}, \ldots, 1, 0)_2$, point $P \in E(\mathbb{F}_{2^m})$
Ensure: $R = k \cdot P$.
1: Compute the width-w regular wTNAF of $r_0' + r_1'\tau$ as $\sum_{i=0}^{\lceil \frac{n+3}{w-1} \rceil} u_i \tau^i (w-1)$
2: Generate look-up table T consisting of points $P_u = \alpha_u P$, for $\{\pm 1, \pm 3, \ldots, \pm(2^{w-1} - 1)\}$
3: $R \leftarrow \infty$
4: **for** i from $l - 1$ downto 0 **do**
5: $R \leftarrow \tau^{w-1} R$
6: $R \leftarrow R + P_{u_i}$
7: **end for**
8: $R \leftarrow R - (r_0' - r_0)P - (r_1' - r_1)\tau P$
9: Return R

Note that $\times_{64-bit}$ means the proposed Block-Comb multiplication on 64-bit (Alg. 3) in the Equation.

Alg. 4 shows the proposed Karatsuba-based Block-Comb multiplication method. In order to utilize the Karatsuba technique, the original multiplier B is firstly turned into the encoded multiplier EB by calling Alg. 2. Afterward, Karatsuba is performed according to Equation 4. Note that unlike the original Karatsuba formula, the proposed method makes use of the encoded multiplier EB like $EB[7 \ldots 0]$, $EB[15 \ldots 8]$, $EB[23 \ldots 16]$, and $EB[31 \ldots 24]$.

3) COMPARISON TO OTHER POLYNOMIAL MULTIPLICATION METHODS

Table 1 compares the proposed method with other BC-based methods with respect to the block size and the required number of block-wise partial products when computing a polynomial multiplication over different bit lengths. Compared with BC, UBC, and KBC, the proposed method requires much reduced number of block-wise partial products by up-to 77%, 73%, and 30%, respectively. In Table 2, the required number of registers for a multiplicand, a multiplier, and an

TABLE 1. Comparison of *BC*-based methods on 8-bit AVR microcontroller with respect to the number of partial products (*BC*, *UBC*, and *KBC* represent Block-Comb [38], Unbalanced Block-Comb [39], and Karatsuba Block-Comb [12], respectively).

bit size	*BC* [38]	*UBC* [39]	*KBC* [12]	This work
128	9	9	6	3
163	16	9	6	6
192	16	16	9	6
233	25	25	13	9
256	36	25	13	9
320	49	36	17	13
384	64	49	22	17
448	100	64	27	22
512	121	100	39	27

TABLE 2. The number of required registers for *BC*-based methods (*BC*, *UBC*, and *KBC* represent Block-Comb [38], Unbalanced Block-Comb [39], and Karatsuba Block-Comb [12], respectively. *acc* means the number of registers for an accumulator) when computing each partial product in the process of polynomial multiplication over $GF(2^{233})$ on 8-bit AVR microcontroller.

Methods	multiplicand	multiplier	*acc*	total
BC [38]	6	6	13	25
UBC [39]	8	7	14	29
KBC [12]	8	7	14	29
This work	9	1	16	26

TABLE 3. Execution time of polynomial multiplication over $GF(2^{233})$ (*cc* means clock cycles). The timing results of our works include function call overheads such as register `POP`/`PUSH` instructions.

Methods	Timing (*cc*)
LUT [11]	8,314
BC [38]	9,511
KBC [12]	8,902
This work w/ encoding	**6,896**
This work w/o encoding	**6,248**

accumulator is given when computing each partial product in the process of polynomial multiplication. The proposed method reduces the number of registers from 29 to 26 with even smaller number of block-wise partial products than the previous works do.

We have implemented the proposed multiplication method with AVR assembly language on 8-bit AVR ATmega128 microcontroller. Table 3 compares the performance with the previous state-of-art results over $GF(2^{233})$. The proposed multiplication algorithm shows about 27.49% better performance than the original *BC* method [38] computing binary field multiplication over $GF(2^{239})$. This performance enhancement can be further optimized by using offline

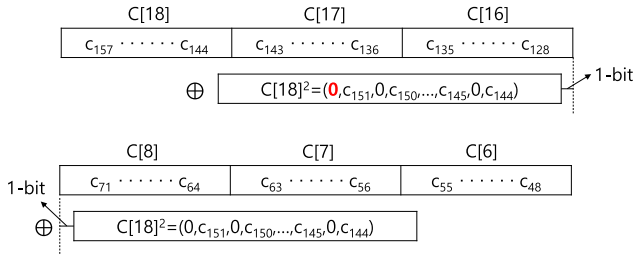


FIGURE 3. Example of one byte modular squaring by $f_{233}(z) = z^{233} + z^{74} + 1$.

encoding for certain fixed parameters.[7] The method without encoding requires 34.3% lesser than the original *BC*. The proposed method with (resp. without) the encoding process also provides about 22.53% (resp. 29.81%) of improved performance compared with *KBC* [12]. Note that the proposed multiplication method with (resp. without) the encoding process provides about 17.05% (resp. 24.84%) of enhanced performance compared with the previous best-known result from [11].

B. OTHER FIELD ARITHMETICS PROPOSED

Modular squaring and reduction operations are also one of the most frequently used operations during scalar multiplication. Thus, we additionally optimize them by considering the characteristics of the underlying field and by maximizing the use of registers on 8-bit AVR microcontrollers.

1) OPTIMIZATION OF FIELD SQUARING

To optimize the modular squaring, we propose table-based modular squaring, which combines squaring and reduction processes. Our method makes use of two kinds of tables: 8-bit wise ordinary squaring table $T8$ and 8-bit wise modular squaring table $TR8$. $T8$ outputs two bytes of the squaring result as one byte input. For example, if 8-bit input is $(a_7, a_6, \ldots, a_0)$, the result of $T8$ is $(0, a_7, 0, a_6, \ldots, 0, a_0)$. Thus, $T8$ requires 512 bytes. While ordinary squaring table $T8$ is constant regardless of the base field, modular squaring table needs to be carefully constructed considering the property of the used irreducible polynomial. The implementation for 8-bit modular squaring with $f_{233}(z) = z^{233} + z^{74} + 1$[8] requires five bytes for one byte modular squaring. Figure 3 shows the example of one byte modular squaring with $f_{233}(z)$. In the figure, one byte $C[18] = (c_{151}z^{151} + \ldots + c_{144}z^{144})$ is squared and then reduced by $f_{233}(z)$. The result of squaring is two bytes $C[18]^2 = (0 \cdot z^{303} + c_{151}z^{302} + \ldots + c_{144}z^{288})$. Since the result of $C[18]^2$ is larger than $f_{233}(z)$, each bit of $C[18]^2$ is reduced. From the figure, we find that the reduced bytes are XORed with six bytes (Namely, $C[18]^2$ is XORed with upper three bytes ($C[19]$, $C[18]$, and $C[17]$) and lower

[7] For example, when computing two binary field multiplications such as $X_1 \cdot Y$ and $X_2 \cdot Y$, only one multiplier encoding for Y is required and the encoded multiplier can be shared in two multiplications.

[8] K-233 curve uses this irreducible polynomial for field reduction.

Algorithm 6 Proposed Modular Squaring Over $GF(2^{233})$ on 8-bit AVR Microprocessor

Require: An operand A over $GF(2^{233})$.
Ensure: $C = A^2 \mod f(z)$ where $f(z) = z^{233} + z^{74} + 1$. (Table-based squaring)
1: $C[39 \cdots 18] \leftarrow T8[A[19 \cdots 9]]$ (Handling $A29$)
2: $T \leftarrow A[29]$
3: $C[38] \leftarrow C[38] \oplus T \ll 1$
4: $C[28] \leftarrow C[28] \oplus T \ll 7$
(Table-based modular squaring (handling $A28$–$A24$))
5: $j = 36, k = 26$
6: **for** $i = 28$ downto 24 **do**
7: $C[j+1, j] \leftarrow C[j+1, j] \oplus TR8H[A[i]]$
8: $C[k+2, k+1, k] \leftarrow C[k+2, k+1, k] \oplus TR8L[A[i]]$
9: $j -= 2, k -= 2$
10: **end for**
(Table-based modular squaring (handling $A23$–$A20$))
11: $j = 26, k = 16, l = 8$
12: **for** $i = 23$ downto 20 **do**
13: $C[j+1, j] \leftarrow C[j+1, j] \oplus TR8H[A[i]]$
14: $C[k+1, k] \leftarrow T8[A[l]]$
15: $C[k+2, k+1, k] \leftarrow C[k+2, k+1, k] \oplus TR8L[A[i]]$
16: $j -= 2, k -= 2, l -= 1$
17: **end for**
(On the fly reduction (handling $C39$–$C30$))
18: $j = 18, k = 8, l = 4$
19: **for** $i = 38$ downto 30 **do**
20: $C[j+2] \leftarrow C[j+2] \oplus (C[i+1] \gg 7)$
21: $C[j+1] \leftarrow C[j+1] \oplus (C[i+1] \ll 1) \oplus (C[i] \gg 7)$
22: $C[j] \leftarrow C[j] \oplus (C[i] \ll 1)$
23: $C[k+1, k] \leftarrow T8[A[l]]$
24: $C[k+2] \leftarrow C[k+2] \oplus (C[i+1] \gg 1)$
25: $C[k+1] \leftarrow C[k+1] \oplus (C[i+1] \ll 7) \oplus (C[i] \gg 1)$
26: $C[k] \leftarrow C[k] \oplus (C[i] \ll 7)$
27: $j -= 2, k -= 2, l -= 1$
28: **end for**
(Handling $C29$)
29: $T \leftarrow C[29] \& 0xFE$
30: $C[10] \leftarrow C[10] \oplus T \gg 7$
31: $C[9] \leftarrow C[9] \oplus T \ll 1$
32: $C[0] \leftarrow C[0] \oplus T \gg 1$
33: $C[29] \leftarrow C[29] \& 0x01$
34: Return $(C[29], \ldots, C[0])$

three bytes ($C[8]$, $C[7]$, and $C[6]$)). We can reduce the size of one byte modular squaring from six bytes to five bytes by using the fact that MSB (Marked as red-colored zero) of squaring is always zero (Refer to the upper reduction process in Figure 3). In our implementation, modular squaring table $TR8$ is composed of an upper part ($TR8H$) and a lower part ($TR8L$). $TR8H$ and $TR8L$ contain elements to be XORed with upper two bytes and lower three bytes, respectively. Thus, the modular squaring table in our implementation requires 1,280 bytes. Since our strategy requires relatively large table (1,792 bytes in total), we make use of FLASH instead of RAM similar to the works from [12].

Figure 4 depicts the detailed execution of the proposed modular squaring mechanism including register arrangement for accumulator C. In the figure, a set of register files $(R_{21}, \ldots, R_0)$ is used to hold accumulator C containing the intermediate reduction result with the rotating register manner. It mainly consists of table-based squaring, table-based modular squaring, and on-the-fly reduction. With respect to squaring and modular squaring, it makes use of $T8$, $TR8H$, and $TR8L$ tables. When computing table-based modular squaring, the upper two bytes and the lower three bytes are XORed with $TR8H$ and $TR8L$ as $A[i]$ index. Note that in our implementation the accumulator is accommodated within general-purpose registers. Since the number of registers is limited, we extend the concept of the rotating register mechanism in [11] to our implementation to keep the values of accumulator C. Since $C[39], \ldots, C[30]$ are larger than degree z^{233}, they need to be reduced once again. Our implementation reduces $C[39], \ldots, C[30]$ in an on-the-fly manner. Note that since $C[39], \ldots, C[30]$ are kept in the registers, on-the-fly reduction can be faster than table-based reduction (They are kept in registers from R_{21} to R_{12}). For reducing the number of redundant Store operations, our implementation reduces them by two words. In the figure, registers marked by yellow color are stored in accumulator C at each step. The detailed algorithm description is given in Appendix (Refer to Alg. 6 in Appendix).

2) OPTIMIZATION OF FIELD REDUCTION

With respect to field reduction over $GF(2^{233})$, we extend the rotating register mechanism in [11] by maximizing the use of registers on 8-bit AVR microcontroller to reduce the number of redundant memory accesses. When an 8-bit word $C[i]$ where $i > 30$ is reduced by $f_{233}(z)$, four bytes of $C[i] \gg 7$, $C[i] \ll 1$, $C[i] \gg 1$, and $C[i] \ll 7$ are XORed at $C[i-19]$, $C[i-20]$, $C[i-29]$, and $C[i-30]$ positions, respectively. With respect to byte reduction, $C[i]$ where $i \leq 28$ is XORed with four bytes of $C[i] = C[i] \oplus \{(C[i+30] \ll 7) \oplus (C[i+29] \gg 1) \oplus (C[i+20] \ll 1) \oplus (C[i+19] \gg 7)\}$. Thus, we maintain the shifted values and the intermediate reduction results in the available registers to reduce redundant LOAD operations, and combine the three bytes reduction processes into one for reducing redundant STORE operations. Thus, the two-byte results are stored in memory at each step of our reduction mechanism. Then, the registers used to keep the shifted values and the intermediate results are assigned for reducing the next three bytes. Figure 5 describing the detailed process of the proposed reduction method is given in Appendix.

We have implemented the proposed modular squaring and reduction mechanisms with AVR assembly language on 8-bit ATmega128 microcontroller. Table 4 compares the performance of the proposed modular squaring and reduction with the best results previously published. The proposed modular squaring and reduction achieve 21.86% and 3.7% of improvements compared to the work from [11], respectively.

V. OPTIMIZATION OF ECC IMPLEMENTATION

In addition to the proposed methods in the field arithmetic level, we apply well-known state-of-the-art algorithms in the ECC level for optimizing the performance of scalar multiplication. We target the ECC implementation over NIST-compliant K-233 Koblitz curve. This curve provides at least 112-bit security satisfying the currently recommended key size by NIST [13], [14], and it is currently included in

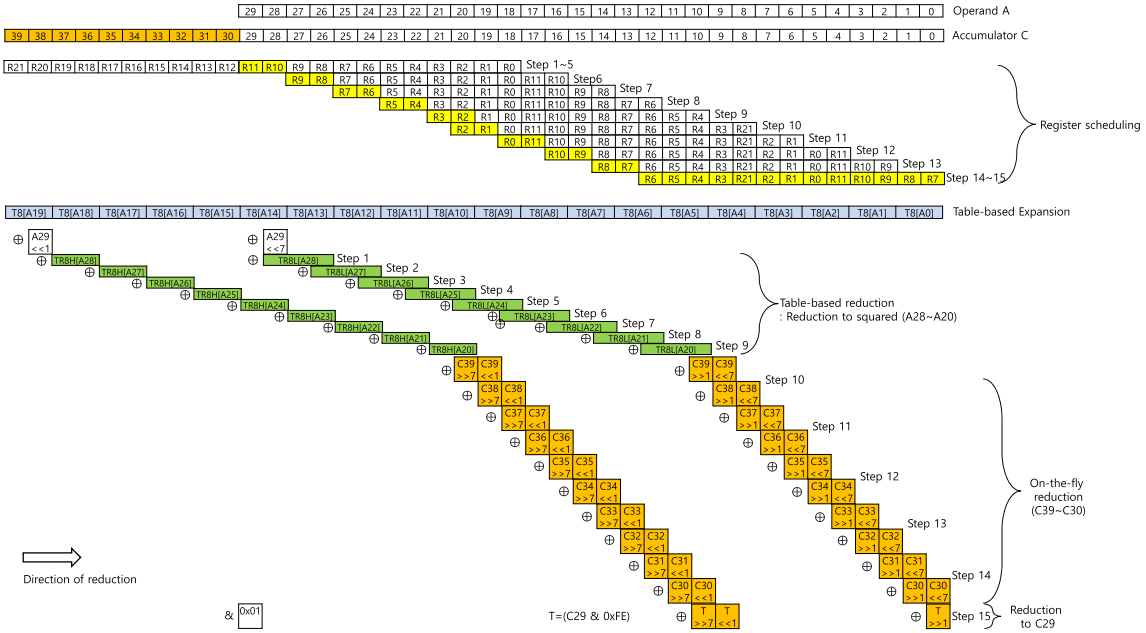


FIGURE 4. Strategy for efficient modular squaring over $GF(2^{233})$ on 8-bit AVR microcontroller (*Ai* and *A*[*i*] mean *i*-th byte in *A*. *Ri* denotes *i*-th register in register files).

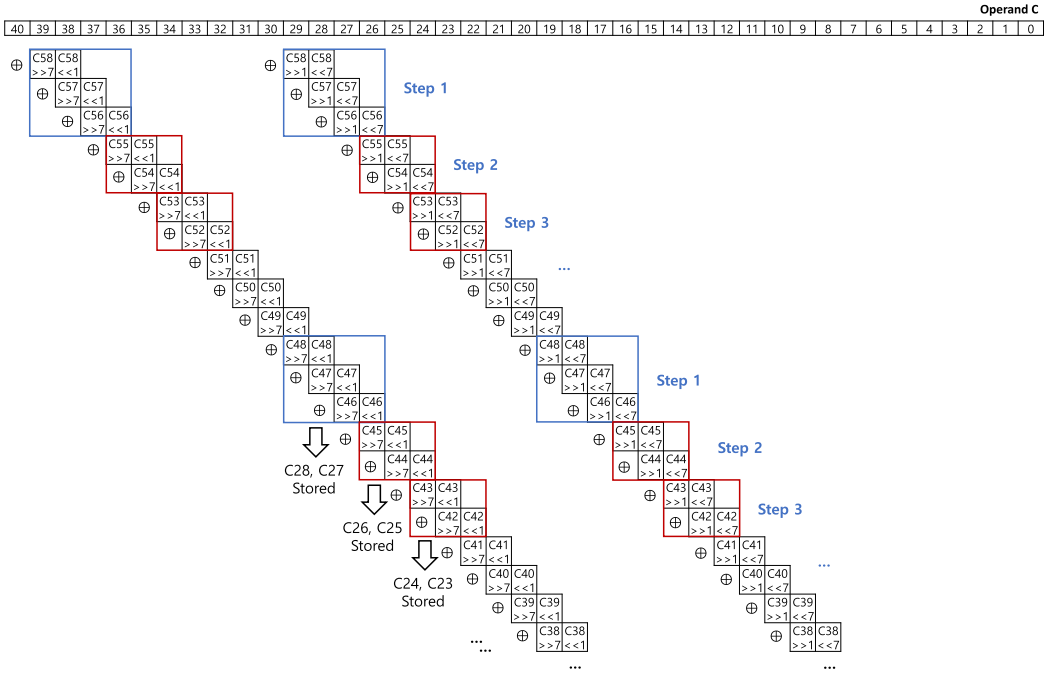


FIGURE 5. Proposed reduction method with $f(z) = z^{233} + z^{74} + 1$ on 8-bit AVR.

an approved algorithm list of NIST CAVP (Cryptographic Algorithm Validation Program) [29] in CMVP (Cryptographic Module Validation Program) [30]. Even though many new ECC curves including Curve25519 [15], Kummer surface [18], Four$\mathbb{Q}$ [19] have been developed recently, NIST curves are still widely used in the industrial area world. Furthermore, since on Koblitz curves, unlike other binary field curves, elliptic curve point doubling (*ECDBL*) can be computed with efficient computable Frobenius map such as $\tau(x, y) = (x^2, y^2)$, $\tau(\infty) = \infty$ with τ expansion of scalar k, the ECC implementation over those curves can have a benefit of efficiency compared with other curves.

Following subsections describe our optimization strategies in elliptic curve operation levels to improve the performance of scalar multiplication.

TABLE 4. Comparison of execution time for modular squaring and reduction over $GF(2^{233})$ (Timing is measured by clock cycles). The timing results of our works include function call overheads such as register `POP/PUSH` instructions.

	Modular Squaring	Reduction
Aranha et al. [11]	956	620
This work	747	597

TABLE 5. Performance result of *HF* version ($k \cdot G$ and $l \cdot P$ mean fixed-base scalar multiplication and variable-base scalar multiplication, respectively). Timing is measured by clock cycles.

w size	#Points	$k \cdot G$	$l \cdot P$
4	3	4,109,102	4,484,518
5	7	3,576,167	**4,381,086**
6	15	**3,186,326**	4,903,248

TABLE 6. Performance result of *HS* version ($k \cdot G$ and $l \cdot P$ mean fixed-base scalar multiplication and variable-base scalar multiplication, respectively). Timing is measured by clock cycles.

w size	#Points	$k \cdot G$	$l \cdot P$
4	3	6,163,020	6,538,436
5	7	4,840,821	**5,645,740**
6	15	**4,056,348**	5,773,270

A. SELECTION OF COORDINATE SYSTEM

Scalar multiplication consists of elliptic curve operations, *ECADD* and *ECDBL*. *ECADD* and *ECDBL* in an affine coordinate system require $1I + 2M$ (I = field inversion, M = field multiplication). Since the field inversion is the most expensive operation in field arithmetics, it is preferable to reduce the number of field inversions when computing elliptic curve operations. Efficient coordinate systems such as projective, Jacobian, López–Dahab (*LD*), and mixed coordinate systems, have been presented [26], [34]. It is known that if the condition ($I > 7M$) is met on the target platform, using *LD* coordinate is more suitable than using affine coordinate (*LD* requires $14M$ and $4M$ for *ECADD* and *ECDBL*, respectively). The cost for the finite field inversion using Extended Euclidean Algorithm (EEA) on 8-bit AVR microcontroller is 142,986 cycle [11], and its cost is much higher than that for modular field multiplication ($\frac{I}{M} > 20$). Thus, our implementation uses *LD* coordinate systems with the mixed addition method [26], [34]. In our implementation using *LD* coordinate system with the mixed addition method over Koblitz curves, *ECADD* and *ECDBL*[9] require $8M + 5S$ and $3S$, respectively.

B. SELECTION OF SCALAR MULTIPLICATION ALGORITHM

The scalar multiplication (computing kP, where a k is scalar and P is a point on the curve) is the most time-consuming part of ECC-based cryptographic protocols and it is composed of a sequence of *ECDBL* and *ECADD*. Since *ECDBL*s in scalar multiplication can be replaced with the efficiently computable Frobenius map, the main concern for improving the performance of scalar multiplication on Koblitz curves is reducing the number of *ECADD*s. We utilize wTNAF [26], [42] for optimizing the performance of scalar multiplication. wTNAF is a windowing method of TNAF and it requires a precomputation table composed of $(2^{w-2} - 1)$ points. wTNAF computes a scalar multiplication with $\frac{m}{w+1} \cdot$ *ECADD*s excluding the cost for building a precomputation table, where m is the bit-length of the scalar's TNAF expression.

There are two types of scalar multiplication in ECC-based applications: fixed-base scalar multiplication and variable-base scalar multiplication. In the fixed-base scalar multiplication, the input point is fixed and known beforehand. The typical case of fixed-base scalar multiplication is an ECC key generation process using the recommended base point, and ECIES and fixed-ECDH require the fixed-base scalar multiplication in its encryption/decryption process and key agreement process, respectively. In the variable-base scalar multiplication, the input point is variable and not given. The variable-base scalar multiplication can be found in an encryption process of ECIES, sign/verify of ECDSA, and a key agreement process of ephemeral-ECDH. The difference between fixed-base and variable-base is whether to build a precomputation table online or not. In other words, variable-base scalar multiplication includes the cost for building a precomputation table. Thus, we need to find the optimal width w of wTNAF for both cases.[10] From our experiments, we find that each optimal width w of wTNAF is 5 and 6 for a variable-base multiplication and a fixed-base scalar multiplication on the 8-bit AVR ATmega128 microcontroller running at 7.3728MHz. The use of width 5 and 6 requires 7 and 15 precomputed points, respectively.

When constructing a precomputation table, the points need to be stored in the affine coordinate system in order to compute scalar multiplication with the mixed coordinate system. Since elliptic curve operations in the affine coordinate system require field inversion, our implementation firstly computes the precomputed points in *LD* coordinate system, and then converts them into the affine coordinate system [44]. Converting n precomputed points in *LD* coordinate system into the affine coordinate system requires n field inversion operations. Thus, we apply Montgomery trick [45] for efficient computation of field inversions such that a^{-1} and b^{-1} are computed as $a^{-1} = (ab)^{-1} \cdot b$ and $b^{-1} = (ab)^{-1} \cdot a$ where a, $b \in GF(2^m)$.

[9] *ECDBL* is computed with efficiently computable Frobenius map $\tau(x, y, z) = (x^2, y^2, z^2)$, $\tau(\infty) = \infty$.

[10] In case of the fixed-base scalar multiplication, Hanser and Wagner [43] reported that τ-comb method could achieve performance improvement over the wTNAF method of up-to 25% on NIST K-233 curve on Java environment running on PC. However, through performance profiling, we find out that wTNAF provides better performance than τ-comb when they use the same number of precomputed points.

VI. CONSIDERATION OF SIDE CHANNEL ANALYSIS

A. FIELD INVERSION

Even though field inversion is the most expensive arithmetic, a few inversions need to be executed as in computing a precomputation table and coordinate conversion from *LD* coordinate to affine coordinate. For efficient computation of inversion, Extended Euclidean Algorithm (EEA) has been widely used. However, EEA operates irregularly depending on the input data. Since the precomputation table does not include secret information, the use of EEA does not affect the security of implementation. However, the use of basic EEA affects the security of scalar multiplication when computing the final inversion for coordinate conversion [47]. To defend this kind of attack, Fermat-based inversion or a random multiplicative masking method can be used [48]. Even if Fermat-based inversion, i.e. $a^{-1} = a^{p-2} \bmod p$, allows one to achieve constant execution time, it is significantly slower than the EEA. Thus, our implementation utilizes a simple multiplicative masking method. This multiplicative masking method uses EEA and requires two additional modular multiplications. For example, when computing inverse of x, instead of inverting x directly, we first multiply x by random value r, and then invert the product $X = r \cdot x$ using the EEA to obtain X^{-1}, and finally multiply the inversion result X^{-1} by r to get $x^{-1} = X^{-1} \cdot r$. Since the attacker does not know r, he/she is not able to get the actual value of x. In summary, our *HF* (Highly-Fast) version uses the basic EEA, and *HS* (Highy-Secure) version utilizes the multiplicative random masking method for computing the final inversion. Note that two versions use EEA when computing the precomputation table.

B. SIDE-CHANNEL RESISTANCE FOR SCALAR MULTIPLICATION

Typical wTNAF is vulnerable to side channel attacks [49], [50], such as TA (Timing Attack) and SPA (Simple Power Analysis), because point addition is omitted whenever the value of the tested window is zero, which leaks sensitive information. Oliveira *et al.* [46] proposed a regular wTNAF-based scalar multiplication method on Koblitz curves. Alg. 5 computes scalar multiplication with a regular pattern, which always conducts $(w-1)$ Frobenius maps and single *ECADD* regardless of the scalar value. With the regular wTNAF method, the length of the scalar is $\lceil 1 + \frac{m+2}{w-1} \rceil$ and the density of nonzero is $\frac{1}{w-1}$ [46]. For finding the optimal window size on the target device, we tested the performance of regular wTNAF and found that 5 and 6 are the optimal window widths for a variable-base scalar multiplication and a fixed-base scalar multiplication, respectively.

Furthermore, our implementation applies the randomized projective coordinate system [50] to the scalar multiplication process in order to randomize the timing difference in field multiplication. For example, when computing field multiplication of *ECADD*, our implementation uses the randomized coordinates as its multiplier. Thus, the attacker cannot distinguish the real value of the multiplier with TA or SPA. This approach also provides resistance against DPA (Differential Power Analysis) and CPA (Correlation Power Analysis) since the attacker cannot determine the intermediate result of the scalar multiplication [50].

VII. PERFORMANCE ANALYSIS AND COMPARISON

We have implemented full scalar multiplication over NIST K-233 curve on 8-bit AVR Atmega128 microcontroller. Actually, we provide two versions: *HF* (Highly-Fast) and *HS* (Highly-Secure). We have optimized the performance of field level arithmetics in polynomial multiplication, modular squaring, field addition, and reduction by AVR assembly language. The elliptic curve arithmetics including *ECADD*, *ECDBL*, and scalar multiplication have been implemented with C language.

Table 5 shows the performance of *HF* (Highly-Optimized) version of our ECC implementation using the proposed algorithms. Since this version uses typical wTNAF and EEA as its scalar multiplication method and field inversion method, it does not provide TA (Timing Attack) and SPA (Simple Power Analysis) resistance. When using 4TNAF,[11] with respect to fixed and variable-based scalar multiplication, our software provides about 16.68% and 15.56% of improved performance compared with the work from [11], respectively. Note that in case of fixed-base (resp. variable-base) scalar multiplication, our software using 6TNAF (resp. 5TNAF) provides about 34.5% (resp. 18.6%) better performance than that from [11].

Table 6 shows the performance of *HS* (Highly-Secure) version of our ECC implementation using the proposed algorithms. Since this version uses regular wTNAF in Alg. 5 and applies multiplicative masked inversion instead of typical EEA, it provides resistance against both TA (Timing Attack) and SPA (Simple Power Analysis). Furthermore, we utilize the randomized coordinate system to protect timing attack on the field multiplication, which requires only three additional field multiplications. When using 4TNAF, the performance of *HS* version is lower than that from [11]. This is because *HS* version utilizes the regular wTNAF providing SPA-resistance, while the work in [11] did not. From our experiments, *HS* version provides the best performance of fixed-base and variable-base scalar multiplication when using regular 6TNAF and regular 5TNAF, respectively. At this time, our implementation provides about 16.63% improved performance when computing a fixed-base scalar multiplication. However, in case of the variable-base scalar multiplication, its performance becomes about 4.89% lower than that in [11].

Table 7 compares our implementation with other ECC implementations over NIST standard-compliant curves providing more than 80-bit security with respect to SCA resistance, performance, code size, and stack usage. Among implementations without SPA-resistance mecha-

[11] The work in [11] utilizes 4TNAF for computing scalar multiplication.

TABLE 7. Comparison to other ECC implementations over NIST-compliant curves on 8-bit AVR microcontroller ($k \cdot G$ and $l \cdot P$ mean fixed-base scalar multiplication and variable-base scalar multiplication, respectively). Timing is measured by clock cycles. KB and B mean KBytes and Byte, respectively.

Implementation	Curve	SCA Resistance	$k \cdot G$	$l \cdot P$	Code Size	Stack Usage
Gura et al. [3]	NIST P-224	No	17,520,000	-	4,812 B	422 B
Wenger et al. [51]	NIST P-256	Yes	34,930,000	-	16,112 B	590 B
Liu A. et al. [4]	NIST P-192	No	21,381,120	21,381,120	19.0 KB	1,510 B
Liu Z. et al. [2]	NIST P-192	Yes	3,460,000	8,620,000	26 KB	1.4 KB
Aranha et al. [11]	NIST K-233	No	4,866,048	5,382,144	38.6 KB	3.7 KB
This works (HF)	NIST K-233	No	**3,186,326**	**4,381,086**	39.2 KB	1,410 B
This works (HS)	NIST K-233	Yes	**4,056,348**	**5,645,740**	39.3 B	1,440 B

TABLE 8. Comparison to other implementations on recently developed new curves on 8-bit AVR microcontroller ($k \cdot G$ and $l \cdot P$ mean fixed-base scalar multiplication and variable-base scalar multiplication, respectively). Timing is measured by clock cycles.

Implementation	Curve	Security level	$k \cdot G$	$l \cdot P$
MoTE-224 [16]	Montgomery curve	112	6,635,520	14,819,328
Renes et al. [52]	Kummer surface	128	9,513,500	9,513,500
Düll et al. [21]	Curve25519	128	13,900,397	13,900,397
Liu et al. [22]	Four$\mathbb{Q}$	128	3,007,300	6,561,500
This works (HF)	NIST K-233	112	**2,660,107**	**4,381,086**
This works (HS)	NIST K-233	112	**3,530,172**	**5,645,740**

nisms, *HF* version provides the best performance. Furthermore, *HS* also provides the best performance among implementations equipped with SPA-resistance mechanisms. Regarding code size and stack usage, even though our implementation requires slightly larger code size than that in [11] because the proposed polynomial multiplication algorithm is implemented with a loop unrolling manner, it uses much reduced stack usage. Since ATmega128 has limited RAM size as 4Kbyte, RAM usage needs to be optimized. Thus, our implementation requires reasonable amounts of code size and stack usage compared to other ECC implementations over NIST-compliant curves.

Table 8 compares our implementation with other state-of-art ECC implementations over new curves. In case of the fixed-base scalar multiplication, since Four$\mathbb{Q}$ implementation [22] utilizes 80 precomputed points, we apply 8TNAF to the scalar multiplication, which requires 63 precomputed points. Without a dedicated hardware multiplier on 8-bit AVR microcontroller, our software provides competitive performance compared with other state-of-art implementations.

VIII. CONCLUSIONS

In this paper, we presented a highly efficient ECC implementation over NIST K-233 curve, providing 112-bit security recommended by NIST, on an 8-bit AVR ATmega128 processor. For optimizing the performance of ECC, we focus on improving the underlying field arithmetics and propose several optimization techniques. Particularly, we propose a novel polynomial multiplication method based on multiplier-encoding, and it significantly reduces the required number of registers for a multiplier, which allows a larger block size for Karatsuba Block-Comb (*KBC*) method. The proposed method provides around 17.05% of improvement compared with the previous best result, and it can be used for polynomial multiplication in ECC over larger fields as $GF(2^{283})$ providing a 128-bit security level. With the proposed methods, we present two versions of ECC implementation: *HF* (Highly-Fast) and *HS* (Highly-Secure) equipped with some SCA countermeasures. *HF* version provides the best performance compared with other existing ECC implementations over NIST-compliant curves, and *HS* version also achieves the best performance among ECC implementations equipped with SCA countermeasures. In the future, we will apply the proposed polynomial multiplication method to ECC over K-283 curve for 128-bit security.

ACKNOWLEDGMENT

This work was supported by ETRI.

REFERENCES

[1] Z. Liu, H. Seo, J. Großchädl, and H. Kim, "Efficient implementation of NIST-compliant elliptic curve cryptography for sensor nodes," in *Proc. Int. Conf. Inf. Commun. Secur.* Springer, 2013, pp. 302–317.

[2] Z. Liu, H. Seo, and J. Großchädl, and H. Kim, "Efficient implementation of NIST-compliant elliptic curve cryptography for 8-bit AVR-based sensor nodes," *IEEE Trans. Inf. Forensics Security*, vol. 11, no. 7, pp. 1385–1397, Jul. 2016.

[3] N. Gura, A. Patel, A. Wander, H. Eberle, and S. C. Shantz, "Comparing elliptic curve cryptography and RSA on 8-bit CPUs," in *Proc. Int. Workshop Cryptograph. Hardw. Embedded Syst.* Springer, 2004, pp. 119–132.

[4] A. Liu and P. Ning, "TinyECC: A configurable library for elliptic curve cryptography in wireless sensor networks," in *Proc. 7th Int. Conf. Inf. Process. Sensor Netw.*, Apr. 2008, pp. 245–256.

[5] L. Uhsadel, A. Poschmann, and C. Paar, "Enabling full-size public-key algorithms on 8-bit sensor nodes," in *Proc. Eur. Workshop Secur. Ad-Hoc Sensor Netw.* Springer, 2007, pp. 73–86.

[6] D. J. Malan, M. Welsh, and M. D. Smith, "A public-key infrastructure for key distribution in TinyOS based on elliptic curve cryptography," in *Proc. 1st Annu. IEEE Commun. Soc. Conf. Sensor Ad Hoc Commun. Netw. (SECON)*, Oct. 2004, pp. 71–80.

[7] H. Yan and Z. J. Shi, "Studying software implementations of elliptic curve cryptography," in *Proc. IEEE 3rd Int. Conf. Inf. Technol., New Generat. (ITNG)*, Apr. 2006, pp. 78–83.

[8] H. Eberle, A. Wander, N. Gura, S. Chang-Shantz, and V. Gupta, "Architectural extensions for elliptic curve cryptography over GF(2μm) on 8-bit microprocessors," in *Proc. 16th IEEE Int. Conf. Appl.-Specific Syst., Archit. Processors (ASAP)*, Jul. 2005, pp. 343–349.

[9] P. Szczechowiak, L. B. Oliveira, M. Scott, M. Collier, and R. Dahab, "NanoECC: Testing the limits of elliptic curve cryptography in sensor networks," in *Wireless Sensor Networks*. Springer, 2008, pp. 305–320.

[10] 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, 2008.

[11] D. F. Aranha, R. Dahab, J. López, and L. B. Oliveira, "Efficient implementation of elliptic curve cryptography in wireless sensors," *Adv. Math. Commun.*, vol. 4, no. 2, pp. 169–187, 2010.

[12] H. Seo, Z. Liu, J. Choi, and H. Kim, "Karatsuba–block-comb technique for elliptic curve cryptography over binary fields," *Secur. Commun. Netw.*, vol. 8, no. 17, pp. 3121–3130, 2015.

[13] *Nist sp 800-131a rev1, Transitions: Recommendation for Transitioning the use of Cryptographic Algorithms and key Lengths*. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-131Ar1.pdf

[14] E. B. Barker, W. C. Barker, W. E. Burr, W. T. Polk, and M. Smid, "Recommendation for key management," NIST, Gaithersburg, MD, USA, NIST SP 800-57 Part 1: Revision 4, Jan. 2016.

[15] D. J. Bernstein, "Curve25519: New diffie-hellman speed records," in *Proc. Int. Workshop Public Key Cryptogr.* New York, NY, USA: Springer, Apr. 2006, pp. 207–228.

[16] Z. Liu, E. Wenger, and J. Großschädl, "MoTE-ECC: Energy-scalable elliptic curve cryptography for wireless sensor networks," in *Proc. Int. Conf. Appl. Cryptogr. Netw. Secur.* Springer, 2014, pp. 361–379.

[17] Z. Liu, X. Huang, Z. Hu, M. K. Khan, H. Seo, and L. Zhou, "On emerging family of elliptic curves to secure Internet of Things: ECC comes of age," *IEEE Trans. Depend. Sec. Comput.*, vol. 14, no. 3, pp. 237–248, Jun. 2017.

[18] J. W. Bos, C. Costello, H. Hisil, and K. E. Lauter, "Fast cryptography in genus 2," in *Proc. 32nd Annu. Int. Conf. Theory Appl. Cryptograph. Techn.*, Athens, Greece: Springer, May 2013, pp. 194–210.

[19] C. Costello and P. Longa, "Fou $\mathbb{Q}$: Four-dimensional decompositions on a $\mathbb{Q}$-curve over the mersenne prime," in *Proc. 21st Int. Conf. Theory Appl. Cryptol. Inf. Secur.*, Auckland, New Zealand: Springer, Nov./Dec. 2015, pp. 214–235.

[20] M. Hutter and P. Schwabe, "NaCl on 8-Bit AVR microcontrollers," in *Proc. Int. Conf. Cryptol. Africa*. Springer, 2013, pp. 156–172.

[21] M. Düll *et al.*, "High-speed Curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers," *Des., Codes Cryptogr.*, vol. 77, nos. 2–3, pp. 493–514, 2015.

[22] Z. Liu, P. Longa, G. Pereira, O. Reparaz, and H. Seo, "Four$\mathbb{Q}$ on embedded devices with strong countermeasures against side-channel attacks," in *Proc. Int. Conf. Cryptograph. Hardw. Embedded Syst.* Taipei, Taiwan: Springer, Sep. 2017, pp. 665–686.

[23] J. López and R. Dahab, "Improved algorithms for elliptic curve arithmetic in $GF(2^n)$," in *Proc. Int. Workshop Sel. Areas Cryptogr.* Springer, pp. 201–212, 1998.

[24] A. Kargl, S. Pyka, and H. Seuschek, "Fast arithmetic on ATmega128 for elliptic curve cryptography," IACR Cryptol. ePrint Arch., Tech. Rep., 2008, p. 442. [Online]. Available: http://eprint.iacr.org/2008/442

[25] N. Koblitz, "Elliptic curve cryptosystems," *Math. Comput.*, vol. 48, no. 177, pp. 203–209, 1987.

[26] D. Hankerson, A. J. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography*. Springer, 2006.

[27] V. S. Miller, "Use of elliptic curves in cryptography," in *Proc. Conf. Theory Appl. Cryptograph. Techn.*, Santa Barbara, CA, USA: Springer, Aug. 181985, pp. 417–426.

[28] J. A. Solinas, "Efficient arithmetic on koblitz curves," *Des. Codes Cryptogr.*, vol. 19, nos. 2–3, pp. 195–249, 2000.

[29] *CAVP: Cryptographic Algorithm Validation Program*. [Online]. Available: https://csrc.nist.gov/Projects/Cryptographic-Algorithm-Validation-Program.

[30] *CMVP: Cryptographic Module Validation Program*. [Online]. Available: https://csrc.nist.gov/projects/cryptographic-module-validation-program

[31] *Cryptographic Module Validation Program: Validated Modules*. [Online]. Available: https://csrc.nist.gov/Projects/Cryptographic-Module-Validation-Program/Validated-Modules

[32] B. Möller, "Fractional windows revisited:Improved signed-digit representations for efficient exponentiation," in *Proc. 7th Int. Conf. Inf. Secur. Cryptol. (ICISC)*. Seoul, South Korea: Springer, Dec. 2004, pp. 137–153.

[33] C. H. Lim and P. J. Lee, "More flexible exponentiation with precomputation," in *Proc. Annu. Int. Cryptol. Conf.* Santa Barbara, CA, USA: Springer, Aug. 1994, pp. 95–107.

[34] H. Cohen, A. Miyaji, and T. Ono, "Efficient elliptic curve exponentiation using mixed coordinates," in *Proc. Int. Conf. Theory Appl. Cryptol. Inf. Secur.* Beijing, China: Springer, Oct. 1998, pp. 51–65.

[35] *Atmel, AVR Instruction set Manual*. [Online]. Available: http://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf

[36] J. López and R. Dahab, "High-speed software multiplication in f_{2m}," in *Proc. Int. Conf. Cryptol. India*. Springer, 2000, pp. 203–212.

[37] L. B. Oliveira *et al.*, "TinyPBC: Pairings for authenticated identity-based non-interactive key distribution in sensor networks," *Comput. Commun.*, vol. 34, no. 3, pp. 485–493, 2011.

[38] M. Shirase, Y. Miyazaki, T. Takagi, D.-G. Han, and D. Choi, "Efficient implementation of pairing-based cryptography on a sensor node," *IEICE Trans. Inf. Syst.*, vol. E92-D, no. 5, pp. 909–917, 2009.

[39] H. Seo, Y. Lee, H. Kim, T. Park, and H. Kim, "Binary and prime field multiplication for public key cryptography on embedded microprocessors," *Secur. Commun. Netw.*, vol. 7, no. 4, pp. 774–787, 2014.

[40] C.-N. Chen, "Memory address side-channel analysis on exponentiation," in *Proc. Int. Conf. Inf. Secur. Cryptol.* Springer, 2014, pp. 421–432.

[41] Z. Liu *et al.*, "Secure GCM implementation on AVR," *Discrete Appl. Math.*, vol. 241, pp. 58–66, May 2018.

[42] J. A. Solinas, "An improved algorithm for arithmetic on a family of elliptic curves," in *Proc. Annu. Int. Cryptol. Conf.* Springer, 1997, pp. 357–371.

[43] C. Hanser and C. Wagner, "Speeding up the fixed-base comb method for faster scalar multiplication on koblitz curves," in *Proc. Int. Conf. Availability, Rel., Secur.* Springer, 2013, pp. 168–179.

[44] W. R. Trost and G. Xu, "On the optimal pre-computation of window τNAF for koblitz curves," *IEEE Trans. Comput.*, vol. 65, no. 9, pp. 2918–2924, Sep. 2016.

[45] H. Cohen, *A Course in Computational Algebraic Number Theory*, vol. 138. Springer, 2013.

[46] T. Oliveira, D. F. Aranha, J. López, and F. Rodríguez-Henríquez, "Fast point multiplication algorithms for binary elliptic curves with and without precomputation," in *Proc. Int. Workshop Sel. Areas Cryptogr.* Springer, 2014, pp. 324–344.

[47] D. Naccache, N. P. Smart, and J. Stern, "Projective coordinates leak," in *Proc. Int. Conf. Theory Appl. Cryptograph. Techn.* Springer, 2004, pp. 257–267.

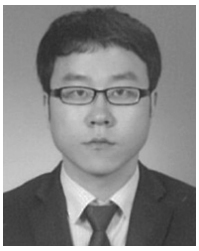
[48] Z. Liu and J. Großschädl, L. Li, and Q. Xu, "Energy-efficient elliptic curve cryptography for MSP430-based wireless sensor nodes," in *Proc. Australas. Conf. Inf. Secur. Privacy* Springer, 2016, pp. 94–112.

[49] P. C. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Proc. Annu. Int. Cryptol. Conf.* Santa Barbara, CA, USA: Springer, Aug. 1999, pp. 388–397.

[50] J.-S. Coron, "Resistance against differential power analysis for elliptic curve cryptosystems," in *Proc. Int. Workshop Cryptograph. Hardw. Embedded Syst.* Worcester, MA, USA: Springer, Aug. 1999, pp. 292–302.

[51] E. Wenger, T. Unterluggauer, and M. Werner, "8/16/32 shades of elliptic curve cryptography on embedded processors," in *Proc. Int. Conf. Cryptol. India*. Springer, 2013, pp. 244–261.

[52] J. Renes, P. Schwabe, B. Smith, and L. Batina, "μ Kummer: Efficient hyperelliptic signatures and key exchange on microcontrollers," in *Proc. Int. Conf. Cryptograph. Hardw. Embedded Syst.* Springer, 2016, pp. 301–320.



SEOG CHUNG SEO received the B.S. degree in information and computer engineering from Ajou University, Suwon, South Korea, in 2005, the M.S. degree in information and communications from the Gwangju Institute of Science and Technology (GIST), Gwangju, South Korea, in 2007, and the Ph.D. degree at Korea University, Seoul, South Korea, in 2011. He was a Research Staff Member at the Samsung Advanced Institute of Technology and the Samsung DMC R&D Center from 2011 to 2014. He has been with The Affiliated Institute of ETRI, South Korea, since 2014. His research interests include public-key cryptography, its efficient implementations on various IT devices, cryptographic module validation program, network security, and data authentication algorithms.

HWAJEONG SEO received the B.S.E.E., M.S., and Ph.D. degrees in computer engineering from Pusan National University. He is currently an Assistant Professor with Hansung University. His research interests include Internet of Things and information security.