The Journal of Supercomputing (2021) 77:4852–4870
https://doi.org/10.1007/s11227-020-03441-5

Efficient implementation of modular multiplication over 192-bit NIST prime for 8-bit AVR-based sensor node

Dong-won Park[1] · Seokhie Hong[1] · Nam Su Chang[2] · Sung Min Cho[3]

Accepted: 25 September 2020 / Published online: 27 October 2020

Abstract

Modular multiplication is one of the most time-consuming operations that account for almost 80% of computational overhead in a scalar multiplication in elliptic curve cryptography. In this paper, we present a new speed record for modular multiplication over 192-bit NIST prime P-192 on 8-bit AVR ATmega microcontrollers. We propose a new integer representation named Range Shifted Representation (RSR) which enables an efficient merging of the reduction operation into the subtractive Karatsuba multiplication. This merging results in a dramatic optimization in the intermediate accumulation of modular multiplication by reducing a significant amount of unnecessary memory access as well as the number of addition operations. Our merged modular multiplication on RSR is designed to have two duplicated groups of 96-bit intermediate values during accumulation. Hence, only one accumulation of the group is required and the result can be used twice. Consequently, we significantly reduce the number of load/store instructions which are known to be one of the most time-consuming operations for modular multiplication on constrained devices. Our implementation requires only 2888 cycles for the modular multiplication of 192-bit integers and outperforms the previous best result for modular multiplication over P-192 by a factor of 17%. In addition, our modular multiplication is even faster than the Karatsuba multiplication (without reduction) which achieved a speed record for multiplication on AVR processor.

Keywords Multi-precision modular multiplication · NIST curve P-192 · Efficient implementation · Wireless sensor networks · AVR ATmega microcontrollers

✉ Seokhie Hong
shhong@korea.ac.kr

Extended author information available on the last page of the article

1 Introduction

With the appearance of the rapid advancement of Internet of Things (IoT), wireless sensor networks (WSNs) are recognized as important enablers consisting of a numerous number of resource-constrained sensor nodes. Recently, many constrained sensor nodes are widely used to monitor and record physical and environmental conditions such as temperature, sound, and pollution levels. Compared with traditional wired networks, it is harder to obtain security in WSNs where sensor nodes are easily captured or eavesdropped by adversaries owing to the environment of wireless communication. Such security issues naturally raise a requirement for the cryptographic mechanism in WSNs which enables secure and reliable communication. However, it is difficult to provide sufficient security on WSNs because of many restrictions on computation capability, energy consumption, and even storage space for constrained sensor nodes. For example, MICAz mote is widely considered as a representative of constrained 8-bit sensor nodes. It is equipped with an AVR ATmega128 processor which has 4 Kbytes of RAM and 128 Kbytes of programmable flash memory with clock frequency of 7.3728 MHz. The energy consumption of cryptographic software executed on a processor is closely related to its execution time, where faster execution time of cryptographic algorithm usually translates to savings in energy.

In early days, it is believed that Public-Key Cryptosystems (PKCs) are infeasible to be implemented for resource-constrained sensor node since they require a significant amount of computation. Until recently, many types of researches have been proposed to apply PKCs for secure communication on WSNs by overcoming the restrictions of resource-constrained sensor nodes [1–4]. Elliptic curve cryptography (ECC) is considered as a better choice for WSNs than conventional PKCs, such as RSA and DSA owing to its short key length. For example, the 160-bit key in ECC scheme provides the same level of security in RSA scheme with 1024-bit key. Such small key in ECC allows lower memory footprint and bandwidth consumption on WSNs. Moreover, only 5% to 10% of the execution time of RSA exponentiation is required for a scalar multiplication which is the most time-consuming part of all ECC-based schemes.

ECC-based schemes such as the Elliptic Curve Diffie–Hellman (ECDH) key exchange and the Elliptic Curve Digital Signature Algorithm (ECDSA) are composed of three levels of operations as described in Fig. 1. The main operation of virtually all ECC-based schemes is scalar multiplication which requires elliptic curve point arithmetic operations such as elliptic curve point addition and elliptic curve point doubling. These point arithmetic operations are composed of field arithmetic operations such as multiplication, squaring, addition, and inversion. Except for field inversion, multiplication is the most time-consuming operation that accounts for almost 80% of computational overhead in computation of scalar multiplication. After multiplication, reduction operation should always be executed to reduce the double sized result.

For efficient ECC implementation on resource-constrained environments, careful design of field arithmetic operations is required where the most

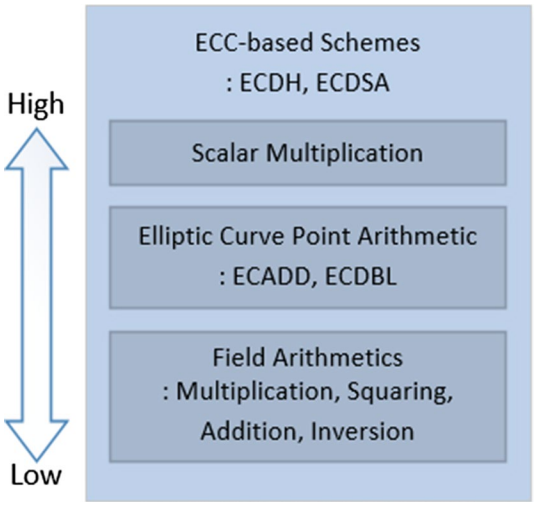


Fig. 1 Hierarchy of ECC-based scheme

performance-critical operation is multi-precision multiplication. Hence, the majority researches of ECC implementation have been focused on improving the performance of multi-precision multiplication for constrained sensor nodes.

1.1 Related work and motivation

After the first ECC implementation by Gura et al. [1], there have been a variety of approaches to optimize ECC implementation for constrained devices. Many studies have focused on improving the performance of multi-precision multiplication which is the most critical factor for an efficient implementation of scalar multiplication.

In 1994, Comba described an efficient column-wise approach of multi-precision multiplication referred as the product scanning method on Intel processor [5]. Until 2004, this method had been known as the fastest multiplication with quadratic complexity on AVR processor. However, this is changed for integers with size larger than 96 bits.

In CHES 2004 [1], Gura et al. presented the hybrid method which combines the advantage of conventional byte-wise multiplication techniques such as the operand scanning and product scanning methods. The hybrid method aims at minimizing the number of load instruction on processor with a large register file by processing four bytes for each iteration of the inner loop in the calculation. Such a significant reduction in load instruction in the hybrid method introduced a speed improvement of up to 25% compared to the product scanning method. Their 160-bit multiplication requires 3106 clock cycles on 8-bit ATmega128 processor. After that, several authors applied this method to accelerate the scalar multiplication of ECC implementation. Most of them focused on optimizing the performance of the hybrid method and proposed some variants that reported between 2593 and 2881 clock cycles on 8-bit ATmega128 processor [6–9].

The next milestone belongs to Hutter and Wenger who proposed the operand caching method [10]. Their technique increases performance of multiplication by caching the operands in the general-purpose registers to reduce the number of load instructions. The operand caching method is slightly improved in WISA 2012, where Seo and Kim introduce an advanced consecutive operand caching method [11, 12].

In 2015, the subtractive Karatsuba method was carefully revisited in [13] by Hutter and Schwabe. This method makes further improvement for the implementation of subtractive Karatsuba method which costs only 1969 clock cycles for 160-bit operands and sets the speed record of multi-precision multiplication on ATmega processor. In [3], it is also proved that Karatsuba method is fastest approach for modular multiplication on constrained devices.

From the point of view of implementation on constrained devices, load and store instructions have a huge influence on the performance of multi-precision multiplication. Hence, the main concern of various multiplication methods is reducing the memory access for operands or intermediate accumulated results during the multiplication. Recently, the operand caching method [10] and Karatsuba multiplication [13] show that careful scheduling of memory access can lead to best performance by maximizing the use of available registers.

Until now, the reduction is treated as a separate part of multiplication process. Most studies do not concentrate on optimizing the reduction operation despite that it always follows multiplication, and consequently, can cause huge memory access overhead by recalling the previous results. In this paper, we focus on finding an effective way of reducing unnecessary memory access by considering multiplications and reductions as a whole.

1.2 Contributions

In this paper, we propose a new method for a fast modular multiplication over 192-bit prime recommended by the US National Institute of Standards and Technology (NIST). The result of our work sets a new speed record on an 8-bit AVR ATmega processor. The following list details the contributions of our work.

- We propose a new integer representation to optimize the implementation of modular multiplication using the characteristic of modulo prime which has the term "$-$ 1." In this regard, we choose the 192-bit NIST standard prime, which has such characteristic and suitable for constrained devices.
- On the basis of the new integer representation, we present a novel approach for the 192-bit modular multiplication over the 192-bit NIST prime for 8-bit architectures. By merging the reduction operations into the subtractive Karatsuba multiplication on the new integer representation, we optimize the intermediate accumulation in the modular multiplication. Our merged modular multiplication has two duplicated groups of 96-bit intermediate results during accumulation. Hence, only one accumulation of the group is required and the result can be used twice. Consequently, we significantly reduce the number of load/store instructions as well as that of addition instructions.

- We present the implementation result of our proposed 192-bit modulo multiplication over the 192-bit NIST prime on an 8-bit AVR ATmega microcontrollers. The result of our work takes only 2888 clock cycles, which is 17% faster than the previous best record of modular multiplication by Liu et al. [3]. In addition, our modular multiplication is even faster than Hutter's subtractive Karatsuba multiplication (without reduction) [13] which achieved a speed record for multiplication on AVR processor.

This paper is organized as follows: In Sect. 2, we give a brief introduction of ECC including NIST curve P-192 and review various multi-precision multiplication techniques. In Sect. 3, we propose the new modular multiplication over the 192-bit NIST prime. Section 4 compares our work with previous works. Finally, we conclude the paper in Sect. 5.

2 Preliminaries

2.1 Elliptic curve cryptography

Elliptic curve cryptography is first introduced by Koblitz and Miller in 1985 [14, 15]. The security of ECC is based on the Elliptic Curve Discrete Logarithm Problem (ECDLP), and there is no general-purpose subexponential algorithms to solve the ECDLP. Let $\mathbb{F}_P$ be a finite field with odd characteristic. An elliptic curve E over $\mathbb{F}_P$ can be defined through a short Weierstraßequation of the form $y^2 = x^3 + ax + b$, where $a, b \in \mathbb{F}_P$ and $4a^3 + 27b^2 \neq 0$. It is preferred that the curve parameter a is fixed to -3 to optimize the point arithmetic in scalar multiplication.

NIST first proposed five prime-field curves in 1999 [16] for standardization. The so-called NIST curves E can be defined through a short Weierstraßequation of the following form:

$$E : y^2 = x^3 - 3x + b. \tag{1}$$

From the point of view of implementation in resource-constrained devices, the NIST curve P-192 has a better position than other NIST curves because it provides an appropriate security level and proper computational cost on small device [3]. This curve uses prime field $\mathbb{F}_{P_{192}}$, defined by prime $P_{192} = 2^{192} - 2^{64} - 1$. This prime has the special characteristic that it can be expressed as the sum or difference of a small number of powers of 2. In addition, the powers are all multiples of 8, 16, or 32. The reduction algorithm for $\mathbb{F}_{P_{192}}$ is especially fast and suitable on machines having word size of 8, 16, or 32. For example, the result of multiplication can be reduced via three additions modulo P_{192} using the congruence $2^{192} \equiv 2^{64} + 1 \pmod{P_{192}}$.

2.2 Multi-precision multiplication techniques

In this section, we briefly review the multi-precision multiplication techniques for fast execution on constrained device. Throughout this section, we represent X and

Y by n-word integers as $X = x_0 + x_1 W + \cdots + x_n W^n$ and $Y = y_0 + y_1 W + \cdots + y_n W^n$ where $W = 2^8$.

2.2.1 Operand scanning method

The operand scanning method is the most simplest approach to implement multi-precision multiplication. This method is also referred as schoolbook method or row-wise method. The multiplication consists of two parts, i.e., inner loop and outer loop. In the outer loop, the operand x_i is loaded and held in working register during the inner loop. Within the inner loop, the multiplicand y_i is loaded one by one and the partial product is computed by multiplying with x_i. Once the inner loop is completed, the next operand y_{i+1} is loaded and the inner loop is iterated again.

2.2.2 Product scanning method

The product scanning method accumulates partial products in the different way. This method computes partial product column by column where the intermediate result in the same column accumulated immediately in working register without storing and loading. Once the accumulation for a column is completed, the part of final multiplication result is obtained. This consecutive approach makes easy to handle carry propagation. In addition, the product scanning method is very suitable for constrained device, since a few number of registers are needed to compute partial products and accumulation.

2.2.3 Hybrid scanning method

Another way to compute a multi-precision multiplication is the hybrid scanning method [1] which combines the advantages of the operand scanning and the product scanning. The hybrid scanning method consists of two nested loop structures where the inner loop follows the operand scanning method and the outer loop accumulates the result of the inner loop, similar to the product scanning method. The outer loop can be implemented by processing the inner loop as a sequence of partial product blocks. This method can save the number of load instructions by sharing the operands within the block. To maximize the shared operands, it is possible to make full use of available register. However, since the outer loop follows a column-wise approach, there is no shared operand between two consecutive blocks. Hence, all operands need to be reloaded again.

2.2.4 Operand caching method

In [10], Hutter and Wenger proposed the operand caching method. This method is based on the product scanning method, but it separates the computation into several rows. All rows can be further divided into four parts. In the first part, all operands for the first and second part are loaded. In the second part, all operands are kept constant and reused. Only one word of the multiplicand is loaded between consecutive two columns. The third part follows the opposite process of previous part. That is, all

multiplicand are kept constant and reused. Only one word of the operands is loaded for each column. In the last part, no loading of the operand is required, since the working registers hold the operands. It is an efficient way to reduce a significant amount of load operations in the computation of the row by reusing operands already loaded from the previous part. But whenever a row is changed, reload of operand is required since there is no shared operand between the rows. To overcome this disadvantage, Seo and Kim proposed the consecutive operand caching method [11, 12] which re-schedules the rows in order to share the operands when a row is changed.

2.2.5 Subtractive Karatsuba method

In the early 1960s, Karatsuba proposed the notable multiplication technique with sub-quadratic complexity [17]. This Karatsuba method can effectively reduce a multiplication of two n-word operands to three multiplication of two $k(=n/2)$-word operands. Any multiplication method mentioned above can be applied to compute the reduced half-size multiplication. In [13], Hutter and Schwabe highly optimized implementation of the subtractive Karatsuba method for various ranges of operands on AVR processor. We can explain the subtractive Karatsuba multiplication on the 8-bit platform as follows:

Let $X = X_A + X_B \cdot W^k$ and $Y = Y_A + Y_B \cdot W^k$. Then,

$$L = X_A \cdot Y_A = L_A + L_B \cdot W^k, \tag{2}$$

$$H = X_B \cdot Y_B = H_A + H_B \cdot W^k. \tag{3}$$

We can compute $X \cdot Y$ as

$$X \cdot Y = L + (L + H - (X_A - X_B) \cdot (Y_A - Y_B)) \cdot 2^{8k} + H \cdot 2^{8k}. \tag{4}$$

The main idea of optimization technique in [13] is to reduce memory access by using duplicated computation of $L_B + H_A$ occurred twice in $X \cdot Y$. In addition, this trick saves k addition operations. The subtractive Karatsuba method in [13] shows the best performance for multi-precision multiplication on an 8-bit processor.

3 Proposed modular multiplication

3.1 Range shifted representation

Generally, we can represent 192-bit integers X, Y and their multiplication $Z = X \cdot Y$ based on 8-bit word size ($W = 2^8$) as follows:

$$X = \sum_{i=0}^{i=23} x_i W^i = x_0 + x_1 W + \cdots + x_{23} W^{23}, \tag{5}$$

$$Y = \sum_{i=0}^{i=23} y_i W^i = y_0 + y_1 W + \cdots + y_{23} W^{23}, \tag{6}$$

$$Z = X \cdot Y = \sum_{i=0}^{i=47} z_i W^i = z_0 + z_1 W + \cdots + z_{47} W^{47}, \tag{7}$$

where $x_i, y_i, z_i \in [0, 2^8 - 1]$.

For simplicity, we can rewrite Z as presented in (10).

$$Z_A = z_0 + z_1 W + \cdots + z_{23} W^{23}, \tag{8}$$

$$Z_B = z_{24} + z_{25} W + \cdots + z_{47} W^{23}, \tag{9}$$

$$Z = Z_A + Z_B \cdot W^{24}. \tag{10}$$

For modular reduction, NIST prime $P_{192} = 2^{192} - 2^{64} - 1$ can be used. We can use the equation $W^{24} \equiv W^8 + 1 \pmod{P_{192}}$ for modulo P_{192} reduction. Then, we have

$$Z \pmod{P_{192}} \equiv Z_A + Z_B + Z_B \cdot W^8. \tag{11}$$

This is not complete reduction. We need to reduce the part $(z_{40} W^{24} + z_{41} W^{25} + \cdots + z_{47} W^{31})$ of $Z_B \cdot W^8$ that is not in the range of the 192-bit element. Here we omit the complete reduction step for simplicity.

In the following, we propose a new integer representation for 192-bit integer which ranges from 2^{-96} to $2^{96} - 1$. We call it Range Shifted Representation (RSR). We can represent 192-bit integers X, Y and their multiplication $Z = X \cdot Y$ with RSR as follows:

$$X = \sum_{i=0}^{i=23} x_i W^{i-12} = x_0 W^{-12} + x_1 W^{-11} + \cdots + x_{23} W^{11}, \tag{12}$$

$$Y = \sum_{i=0}^{i=23} y_i W^{i-12} = y_0 W^{-12} + y_1 W^{-11} + \cdots + y_{23} W^{11}, \tag{13}$$

$$Z = X \cdot Y = \sum_{i=0}^{i=47} z_i W^{i-24} = z_0 W^{-24} + z_1 W^{-23} + \cdots + z_{47} W^{23}, \tag{14}$$

where $x_i, y_i, z_i \in [0, 2^8 - 1]$. An interesting thing about the RSR is that the result of multiplication is expanded to both sides. The shape of result is symmetric with respect to W^0. Because we want to represent integers in the range of $[2^{-96}, 2^{96} - 1]$, we have to transform P_{192} into the range shifted form for modular reduction. We can use range shifted prime $P_{192} \cdot 2^{-96} = 2^{96} - 2^{-32} - 2^{-96}$ for modular reduction. We have to reduce the result at both sides such that $z_0 W^{-24} + z_1 W^{-23} + \cdots + z_{11} W^{-13}$

and $z_{36}W^{12} + z_{37}W^{13} + \cdots + z_{47}W^{23}$ are reduced by modulo $P_{192} \cdot 2^{-96}$. Let $X, Y, Z \in \mathbb{F}_{P_{192}}$ be represented with RSR where $Z = X \cdot Y$. Then, we can reduce Z using the equation $W^{12} \equiv W^{-12} + W^{-4}$ or $W^{-24} \equiv 1 - W^{-16} \pmod{P_{192} \cdot W^{-12}}$.

Let

$$Z_A = z_0 + z_1 W + \cdots + z_{11} W^{11}, \tag{15}$$

$$Z_B = z_{12} + z_{13} W + \cdots + z_{35} W^{23}, \tag{16}$$

$$Z_C = z_{36} + z_{37} W + \cdots + z_{47} W^{11}, \tag{17}$$

$$Z = Z_A \cdot W^{-24} + Z_B \cdot W^{-12} + Z_C \cdot W^{12}, \tag{18}$$

where $z_i \in [0, 2^8 - 1]$.

Then, we can reduce Z as follows:

$$\begin{aligned} Z \equiv Z_A - Z_A \cdot W^{-16} + Z_B \cdot W^{-12} + Z_C \cdot W^{-12} + Z_C \cdot W^{-4} \\ \pmod{P_{192} \cdot W^{-12}}. \end{aligned} \tag{19}$$

Note that, for complete reduction, we need to reduce the part $(-z_0 - z_1 W - z_2 W^2 - z_3 W^3)$ of $-Z_A \cdot W^{-16}$ that is not in the range of RSR. Here we omit the complete reduction step for simplicity.

To utilize RSR in elliptic curve protocol like ECDH or ECDSA scheme, conversions from the original integer representation to RSR and vice versa are required. For example, let X, Y are coordinates of input point for scalar multiplication, then conversion from $X, Y \in [0, 2^{192} - 1]$ in Eqs. (5, 6) to $X, Y \in [2^{-96}, 2^{96} - 1]$ in Eqs. (12, 13) is required before conducting scalar multiplication. This conversion can be simply done by applying modulo $P_{192} \cdot W^{-12}$ for each coordinate. For the output of the scalar multiplication, conversion from the RSR to original integer representation is required. However, compared to computational cost of scalar multiplication, these conversions require a negligible cycle counts and are needed only once. In regard of computation process of other field arithmetic operations on RSR like addition, subtraction, multiplication, and squaring, it is equal to that on original representation where $P_{192} \cdot W^{-12}$ is used for reduction.

3.2 Modular multiplication with RSR

We can use Karatsuba method for multiplication with RSR. Let $X, Y \in \mathbb{F}_{P_{192}}$ be represented with RSR and $Z = X \cdot Y$.

Let

$$X_A = x_0 + x_1 W + \cdots + x_{11} W^{11}, \tag{20}$$

$$X_B = x_{12} + x_{13} W + \cdots + x_{23} W^{11}, \tag{21}$$

$$Y_A = y_0 + y_1 W + \cdots + y_{11} W^{11}, \tag{22}$$

$$Y_B = y_{12} + y_{13} W + \cdots + y_{23} W^{11}, \tag{23}$$

where $x_i, y_i \in [0, 2^8 - 1]$.

Then, X, Y, Z can be represented as

$$X = X_A \cdot W^{-12} + X_B, \tag{24}$$

$$Y = Y_A \cdot W^{-12} + Y_B, \tag{25}$$

$$\begin{aligned} Z = X \cdot Y &= (X_A \cdot W^{-12} + X_B) \cdot (Y_A \cdot W^{-12} + Y_B) \\ &= X_A Y_A \cdot W^{-24} + X_B Y_B + (X_A Y_B + X_B Y_A) W^{-12} \\ &= X_A Y_A \cdot W^{-24} + X_B Y_B \\ &\quad + (X_A Y_A + X_B Y_B - (X_A - X_B) \cdot (Y_A - Y_B)) W^{-12}. \end{aligned} \tag{26}$$

Let low(L), high(H), middle(M) denote $X_A Y_A, X_B Y_B, (X_A - X_B) \cdot (Y_A - Y_B)$ as follows:

$$\begin{aligned} L = X_A Y_A &= l_0 + l_1 W + \cdots + l_{23} W^{23} = L_A + L_B \cdot W^{12}, \\ (L_A &= l_0 + l_1 W + \cdots + l_{11} W^{11}, L_B = l_{12} + l_{13} W + \cdots + l_{23} W^{11}) \end{aligned} \tag{27}$$

$$\begin{aligned} H = X_B Y_B &= h_0 + h_1 W + \cdots + h_{23} W^{23} = H_A + H_B \cdot W^{12}, \\ (H_A &= h_0 + h_1 W + \cdots + h_{11} W^{11}, H_B = h_{12} + h_{13} W + \cdots + h_{23} W^{11}) \end{aligned} \tag{28}$$

$$M = (X_A - X_B) \cdot (Y_A - Y_B) = m_0 + m_1 W + \cdots + m_{23} W^{23}. \tag{29}$$

We can simply denote Z by L, H, M.

$$\begin{aligned} Z &\equiv L \cdot W^{-24} + H + (L + H - M) W^{-12} \\ &\equiv (L_A + L_B \cdot W^{12}) W^{-24} + H_A + H_B \cdot W^{12} + (H_A + H_B \cdot W^{12} \\ &\quad + L_A + L_B \cdot W^{12} - M) W^{-12} \pmod{P_{192} \cdot W^{-12}} \end{aligned} \tag{30}$$

Then, the result of Karatsuba multiplication can be reduced by $P_{192} \cdot W^{-12}$. We do not need to reduce all part of the result. Because $(L + H - M) W^{-12}$ of Eq. (30) just fits in the 192-bit range of RSR, we need to reduce only two parts $L_A \cdot W^{-24}$ and $H_B \cdot W^{12}$ which overflow on both sides of the RSR range. We can compute Z modulo $P_{192} \cdot W^{-12}$ using the equation $W^{12} \equiv W^{-12} + W^{-4}$ or $W^{-24} \equiv 1 - W^{-16} \pmod{P_{192} \cdot W^{-12}}$ as follows:

$$\begin{aligned} Z \equiv & L_A - L_A \cdot W^{-16} + L_B \cdot W^{-12} + H_A + H_B \cdot W^{-12} + H_B \cdot W^{-4} \\ & + (H_A + H_B \cdot W^{12} + L_A + L_B \cdot W^{12} - M)W^{-12} \\ \equiv & - L_A \cdot W^{-16} + H_B \cdot W^{-4} + (L_A + L_B + H_A + H_B) \\ & + (L_A + L_B + H_A + H_B - M)W^{-12} \pmod{P_{192} \cdot W^{-12}}. \end{aligned} \tag{31}$$

The interesting thing in the above equations is that $(L_A + L_B + H_A + H_B)$ is expressed exactly twice. We can make use of this duplicated intermediate result to reduce memory access and accumulate operations for the efficient implementation of modular multiplication.

3.3 Implementation of modular multiplication with RSR

We used 2-level Karatsuba recursion for implementation of the 192-bit multiplication which is composed of three 96-bit 1-level Karatsuba multiplication, L, H, and M, as represented in Eqs. (27), (28) and (29). Let $L^{(1)}, H^{(1)}$ and $M^{(1)}$ be the 48-bit small multi-precision multiplications for 96-bit 1-level Karatsuba multiplications L, H, and M, respectively. Similarly, let $L^{(2)}, H^{(2)}$ and $M^{(2)}$ be the 96-bit 1-level Karatsuba multiplications for 192-bit 2-level Karatsuba multiplications L, H, and M, respectively.

3.3.1 96-Bit 1-level Karatsuba multiplication

Implementation of 96-bit 1-level Karatsuba multiplication $L^{(2)}, H^{(2)}$ and $M^{(2)}$ follows basically the same scheduling as 96-bit multiplication in [13]. Algorithm 1 is a basic implementation of 96-bit 1-level Karatsuba multiplication presented in [13]. Algorithm 1 is composed of three 48-bit small multi-precision multiplications $L^{(1)}, H^{(1)}$ and $M^{(1)}$ that did not include any load or store instructions, and the result is kept in 11 registers.

Let

$$L^{(1)} = L_A^{(1)} + L_B^{(1)} \cdot W^6, \tag{32}$$

$$H^{(1)} = H_A^{(1)} + H_B^{(1)} \cdot W^6, \tag{33}$$

$$M^{(1)} = M_A^{(1)} + M_B^{(1)} \cdot W^6, \tag{34}$$

where $L_A^{(1)}, L_B^{(1)}, H_A^{(1)}, H_B^{(1)}, M_A^{(1)}$ and $M_B^{(1)}$ are 6-bytes integers. As described in Algorithm 1, we can obtain the result of 96-bit 1-level Karatsuba multiplications $L^{(2)}, H^{(2)}$, and $M^{(2)}$ through the computation of $L^{(1)} + (L^{(1)} + H^{(1)} - M^{(1)})W^6 + H^{(1)} \cdot W^{12}$. We can express this computation in detail as follows:

$$L_A^{(1)} + (L_B^{(1)} + L_A^{(1)} + H_A^{(1)} - M_A^{(1)}) \cdot W^6 + (L_B^{(1)} + H_B^{(1)} + H_A^{(1)} - M_B^{(1)}) \cdot W^{12} + H_B^{(1)} \cdot W^{18}. \tag{35}$$

In Eq. (35), the computation of $L_B^{(1)} + H_A^{(1)}$ is appeared twice. This duplicated computation can be utilized in Algorithm 1 to minimize the register allocation and reduce additional load and store instructions for accumulation process. Let us assume that the result of $L_B^{(1)} + H_A^{(1)}$ in Step 5 is not reused at Step 9, then $L_B^{(1)}$ and $H_A^{(1)}$ should be held in registers before the computation of $L_B^{(1)} + H_A^{(1)}$ in Step 5 and saved in memory with store instructions. Moreover, the result of $L_B^{(1)} + H_A^{(1)}$ is kept in registers for next accumulation. In Step 9, because of the calculation of $L_B^{(1)} + H_A^{(1)}$ the loading of $L_B^{(1)}$ and $H_A^{(1)}$ which are stored in the memory after Step 5 is required. In Algorithm 1, however, the store/load instructions for each $L_B^{(1)}$ and $H_A^{(1)}$ actually are not necessary; only the result of $L_B^{(1)} + H_A^{(1)}$ needs to be kept in registers for reusing at Step 9. Furthermore, six addition instructions for $L_B^{(1)} + H_A^{(1)}$ can be saved.

Algorithm 1 Basic 96-bit×96-bit 1-level Karatsuba multiplication in [14]

Require: $X = (x_0, ..., x_{11})$, $Y = (y_0, ..., y_{11})$
Ensure: $Z = X \cdot Y = (z_0, ..., z_{23})$
1: Load $X_A = (x_0, ..., x_5)$ and $Y_A = (y_0, ..., y_5)$
2: Compute $L^{(1)} \leftarrow X_A \cdot Y_A = (l_0, ..., l_{11})$
3: Store $L_A^{(1)} = (l_0, ..., l_5)$ to $(z_0, ..., z_5)$
4: Load $X_B = (x_6, ..., x_{11})$ and $Y_B = (y_6, ..., y_{11})$
5: Compute $\bar{H}^{(1)} \leftarrow (X_B \cdot Y_B) + L_B = (h_0, ..., h_{11}) + (l_6, ..., l_{11}) = (\bar{h}_0, ..., \bar{h}_{11})$
6: Compute $(X_A - X_B)$ and $(Y_A - Y_B)$
7: Compute $M^{(1)} \leftarrow (X_A - X_B) \cdot (Y_A - Y_B)$
8: Load $L_A^{(1)} = (l_0, ..., l_5)$
9: Compute $T \leftarrow (l_0, ..., l_5, h_6, ..., h_{11}) + \bar{H}^{(1)}$
10: Compute $T \leftarrow T - M^{(1)}$
11: Store T to $(z_6, ..., z_{17})$
12: Ripple carry/borrow from Steps 9+10 through $(\bar{h}_6, ..., \bar{h}_{11})$
13: Store $(\bar{h}_6, ..., \bar{h}_{11})$ to $(z_{18}, ..., z_{23})$

3.3.2 Modified 96-bit 1-level Karatsuba multiplication for $L^{(2)}$

We can represent $L^{(2)}$ as

$$L^{(2)} = L_A^{(2)} + L_B^{(2)} \cdot W^{12}, \tag{36}$$

where $L_A^{(2)}, L_B^{(2)}$ are 12-byte integers. We want to compute $L_A^{(2)} + L_B^{(2)}$ during the computation of 96-bit 1-level Karatsuba multiplication $L^{(2)} = X_A Y_A$ and reload it to build the complete duplicated intermediate result $(L_A^{(2)} + L_B^{(2)} + H_A^{(2)} + H_B^{(2)})$ in 192-bit 2-level Karatsuba multiplication with reduction. Through this process, we can reduce redundant memory access for $L_A^{(2)}$ and $L_B^{(2)}$ in 2-level Karatsuba multiplication. In Algorithm 2, we modified 96-bit 1-level Karatsuba multiplication for $L^{(2)}$ by inserting the computation of $L_A^{(2)} + L_B^{(2)}$ into Algorithm 1.

We can represent $L^{(2)}$ by the 48-bit small multi-precision multiplication $L^{(1)}, H^{(1)}$ and $M^{(1)}$ as follows:

$$\begin{aligned} L^{(2)} = & L_A^{(1)} + (L_A^{(1)} + L_B^{(1)} + H_A^{(1)} - M_A^{(1)}) \cdot W^6 \\ & + (L_B^{(1)} + H_A^{(1)} + H_B^{(1)} - M_B^{(1)}) \cdot W^{12} + H_B^{(1)} \cdot W^{18}. \end{aligned} \tag{37}$$

Then

$$L_A^{(1)} + (L_A^{(1)} + L_B^{(1)} + H_A^{(1)} - M_A^{(1)}) \cdot W^6 = L_A^{(2)} + c \cdot W^{12} \tag{38}$$

where c is 1-byte carry. We can represent $L_A^{(2)}$ and $L_B^{(2)}$ as

$$L_A^{(2)} = L_A^{(1)} + (L_A^{(1)} + L_B^{(1)} + H_A^{(1)} - M_A^{(1)}) \cdot W^6 - c \cdot W^{12}, \tag{39}$$

$$L_B^{(2)} = (L_B^{(1)} + H_A^{(1)} + H_B^{(1)} + c - M_B^{(1)}) + H_B^{(1)} \cdot W^6. \tag{40}$$

We can get $L_A^{(2)}$ easily by taking only 12-byte without carry byte c from Eq. (38). $L_A^{(2)} + L_B^{(2)}$ can be represented as

$$\begin{aligned} L_A^{(2)} + L_B^{(2)} = & (L_A^{(1)} + L_B^{(1)} + H_A^{(1)} + H_B^{(1)} + c - M_B^{(1)}) \\ & + (L_A^{(1)} + L_B^{(1)} + H_A^{(1)} + H_B^{(1)} - M_A^{(1)}) \cdot W^6 - c \cdot W^{12}. \end{aligned} \tag{41}$$

To compute $L_A^{(2)} + L_B^{(2)}$, we first compute $(L_A^{(1)} + L_B^{(1)} + H_A^{(1)} - M_A^{(1)}) \cdot W^6 + (L_A^{(1)} + L_B^{(1)} + H_A^{(1)} + H_B^{(1)} - M_B^{(1)}) \cdot W^{12} + H_B^{(1)} \cdot W^{18}$. Then, upper 6-byte of the first computation, which is $(H_B^{(1)} + c)$, is added to upper 6-byte of $L_A^{(2)}$.

Algorithm 2 96-bit×96-bit 1-level Karatsuba multiplication for the computation of $L^{(2)}$

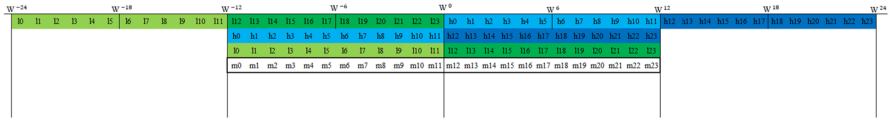
Require: $X = (x_0, ..., x_{11})$, $Y = (y_0, ..., y_{11})$
Ensure: $Z = X \cdot Y = (z_0, ..., z_{23}, carry)$
1: $X_A = (x_0, ..., x_5)$ and $Y_A = (y_0, ..., y_5)$
2: Compute $L^{(1)} \leftarrow X_A \cdot Y_A = (l_0, ..., l_{11})$
3: Compute $L_B^{(1)'} \leftarrow L_B^{(1)} + L_A^{(1)} = (l_6, ..., l_{11}) + (l_0, ..., l_5) = (l'_0, ..., l'_5)$
4: Store $L_A^{(1)} = (l_0, ..., l_5)$ to $(z_0, ..., z_5)$
5: Load $X_B = (x_6, ..., x_{11})$ and $Y_B = (y_6, ..., y_{11})$
6: Compute $\bar{H}^{(1)} \leftarrow (X_B \cdot Y_B) + L_B^{(1)'} = (h_0, ..., h_{11}) + (l'_0, ..., l'_5, 0, 0, 0, 0, 0, 0) = (\bar{h}_0, ..., \bar{h}_{11})$
7: Compute $(X_A - X_B)$ and $(Y_A - Y_B)$
8: Compute $M^{(1)} \leftarrow (X_A - X_B) \cdot (Y_A - Y_B)$
9: Compute $(T, carry) \leftarrow (\bar{h}_0, ..., \bar{h}_5, \bar{h}_0, ..., \bar{h}_5) + (0, 0, 0, 0, 0, 0, \bar{h}_6, ..., \bar{h}_{11}) = (t_0, ..., t_{11}, carry)$
10: Compute $T \leftarrow T - M^{(1)}$
11: Store T to $(z_6, ..., z_{17})$
12: Ripple carry/borrow from Steps 9+10 through $(\bar{h}_6, ..., \bar{h}_{11}, carry)$
13: Compute $(\bar{h}'_6, ..., \bar{h}'_{11}, carry) \leftarrow (\bar{h}_6, ..., \bar{h}_{11}, carry) + (t_0, ..., t_5)$
14: Store $(\bar{h}'_6, ..., \bar{h}'_{11}, carry)$ to $(z_{18}, ..., z_{23}, carry)$

In Algorithm 2, $L_A^{(1)}$ is added to $L_B^{(1)}$ at Step 3. In Step 6, $(L_A^{(1)} + L_B^{(1)} + H_A^{(1)})$ is computed and carry c', which is different from c of (38), is propagated through $(\bar{h}_6, \ldots, \bar{h}_{11})$. In Step 9, $(\bar{h}_0, \ldots, \bar{h}_5)$ is copied to represent the duplicate partial result $(L_A^{(1)} + L_B^{(1)} + H_A^{(1)})$ of Eq. (41) such that $(\bar{h}_0, \ldots, \bar{h}_5, \bar{h}_0, \ldots, \bar{h}_5)$. On the right half of it, $(H_B^{(1)} + c')$ is added. In Step 10, $M^{(1)}$ is subtracted. $(L_A^{(1)} + L_B^{(1)} + H_A^{(1)} - M_A^{(1)}) = (t_0, \ldots, t_5)$ is added to $(H_B^{(1)} + c)$ in Step 13. Then, we can store $L_A^{(2)} + L_B^{(2)}$ in $(z_{12}, \ldots, z_{23}, carry)$. In comparison with Algorithm 1, we can save 6 load instructions for $L_A^{(1)}$ and compute $L_A^{(2)} + L_B^{(2)}$ in Algorithm 2 through this process.

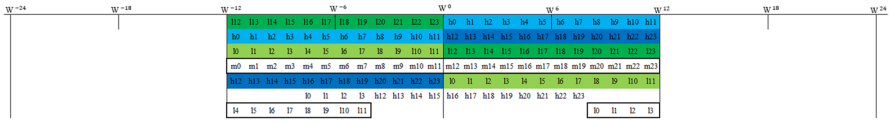
3.3.3 192-Bit 2-level Karatsuba multiplication with reduction

We combined Karatsuba multiplication with reduction on RSR to generate more duplicated intermediate results. The graphical illustrations of 192-bit 2-level Karatsuba multiplication with reduction on RSR are shown in Fig. 2. Figure 2a shows that $L_A^{(2)} = l_0 + \cdots + l_{11}W^{11}$ and $H_B^{(2)} = h_{12} + \cdots + h_{11}W^{23}$ need to be reduced for modular reduction. Figure 2b shows the reduced result of $L_A^{(2)}$ and $H_B^{(2)}$ by $P_{192} \cdot W^{-12}$. Now, we can visualize which one is accumulated for computing the final result of Eq. (31). As mentioned earlier, $(L_A^{(2)} + L_B^{(2)} + H_A^{(2)} + H_B^{(2)})$ is duplicated so that we can use it for reducing memory access and optimize the register usage by inserting accumulated value of the duplicated intermediate results into Karatsuba multiplication with reduction.

Algorithm 3 shows the implementation of 192-bit×192-bit 2-level Karatsuba multiplication with reduction over $\mathbb{F}_{P_{192} \cdot W^{-12}}$. For computing $(L_A^{(2)} + L_B^{(2)} + H_A^{(2)} + H_B^{(2)})$, at first $L_A^{(2)} + L_B^{(2)}$ is computed during the evaluation of $L^{(2)}$ through Algorithm 2 and saved. After the multiplication of $X_B \cdot Y_B$ in Step 4, we get the result $H^{(2)} = H_A^{(2)} + H_B^{(2)} \cdot W^{12}$ and compute $H_A^{(2)} + H_B^{(2)}$ in Step 5. In the next step, we load $L_A^{(2)} + L_B^{(2)}$ and accumulate it to $H_A^{(2)} + H_B^{(2)}$. The accumulated result requires an additional register for a carry byte. Therefore, we can hold the complete duplicated intermediate result $(L_A + L_B + H_A + H_B)$ in 13 registers which is represented by $(T, carry_2) = (t_0, \ldots, t_{11}, carry_2)$. In Step 7, we can represent the other half side of the intermediate result in Fig. 2b by just copying T of duplicated intermediate



(a) Result of Karatsuba multiplication with RSR



(b) Reduced result of Karatsuba multiplication with RSR

Fig. 2 Process of modular multiplication with RSR

results without $carry_2$. This is a very efficient way to decrease the number of load and save operations for previous computation results. Moreover, the number of addition operation is reduced. These advantages save clock cycle counts significantly. In Step 10, $carry_2$ is added for complete accumulation.

Because we cannot always hold the 192-bit result of 1-level Karatsuba multiplication, careful handling of the 32 registers is required to minimize the memory access between 96-bit Karatsuba multiplication $L^{(2)}, H^{(2)}$, and $M^{(2)}$. We reordered the order of computation from $L^{(2)} \rightarrow H^{(2)} \rightarrow M^{(2)}$ in [13] to $M^{(2)} \rightarrow L^{(2)} \rightarrow H^{(2)}$. Since $H_B^{(2)}$ is kept in registers after Step 4, we can directly reduce $H_B^{(2)}$ without any memory access at Step 7. This generates $carry_3$ at which carries from Step 8, Step 9, and Step 10 are accumulated for reducing all carries together at Step 11.

Algorithm 3 192-bit×192-bit 2-level Karatsuba multiplication with reduction over $\mathbb{F}_{P_{192} \cdot W^{-12}}$

Require: $X = (x_0, ..., x_{23})$, $Y = (y_0, ..., y_{23})$
Ensure: $Z = X \cdot Y \pmod{P_{192} \cdot W^{-12}} = (z_0, ..., z_{23})$
1: Load X and Y and compute $(X_A - X_B)$ and $(Y_A - Y_B)$
2: Compute $M^{(2)} \leftarrow (X_A - X_B) \cdot (Y_A - Y_B)$ using Algorithm 1 (except loading Steps)
3: Compute $(L^{(2)}, carry_1) \leftarrow X_A \cdot Y_A = (l_0, ..., l_{23}, carry_1)$ using Algorithm 2
4: Compute $H^{(2)} \leftarrow X_B \cdot Y_B = (h_0, ..., h_{23})$
5: Compute $(T, carry_2) \leftarrow (h_0, ..., h_{11}) + (h_{12}, ..., h_{23}) = (t_0, ..., t_{11}, carry_2)$
6: Load $(l_{12}, ..., l_{23}, carry_1)$ and compute $(T, carry_2) \leftarrow (t_0, ..., t_{11}, carry_2) + (l_{12}, ..., l_{23}, carry_1) = (t_0, ..., t_{11}, carry_2)$
7: Compute $(T', carry_3) \leftarrow (t_0, ..., t_{11}, t_0, ..., t_{11}, carry_2) + (0, 0, 0, 0, 0, 0, 0, 0, h_{12}, ..., h_{23}, 0, 0, 0, 0) = (t'_0, ..., t'_{23}, carry_3)$
8: Load $M^{(2)}$ and compute $T' \leftarrow T' - M^{(2)}$
9: Load (l_0, l_1, l_2, l_3) and compute $(T', carry_3) \leftarrow (t'_0, ..., t'_{23}, carry_3) + (0, 0, 0, 0, l_0, l_1, l_2, l_3, 0, ..., 0)$
10: Ripple $carry_2$ from Steps 6 through $(T', carry_3)$
11: Reduce $carry_3$ from Steps 10 through T'
12: Load $(l_4, ..., l_{11})$ and compute $(T', carry_4) \leftarrow (t'_0, ..., t'_{23}) - (l_4, ..., l_{11}, 0, ..., 0, l_0, l_1, l_2, l_3)$
13: Reduce $carry_4$ from Steps 12 through T'
14: Store T' to $(z_0, ..., z_{23})$

4 Result

In this section, we present the implementation result of our 192-bit modular multiplication on 8-bit AVR ATmega128 processors providing the execution time (cycle counts). The timing of our work is obtained by simulation with Atmel studio 7.0. We refer the cycle counts represented in [18] to compare with various multiplications.

Table 1 shows the execution time of previous works for 192-bit multiplication (only) and 192-bit modular multiplication over NIST P_{192}. The results for multiplication cover various multiplication methods including operand scanning, product scanning, hybrid scanning, operand caching, consecutive operand caching, and Karatsuba method. Among them, the implementation of Karatsuba method by Hutter and Schwabe [13] sets the speed record for 192-bit multiplication. In [3], it is also verified that modular

Table 1 Cycle counts of multiplication and modular multiplication for 192-bit operands on 8-bit ATmega128 processor

Approach	Including reduction	Cycle counts
Operand scanning[a]	X	7760
Product scanning[a]	X	5614
Hybrid scanning[a] [1]	X	4133
Operand caching[a] [10]	X	3470
Consecutive operand caching[a] [11]	X	3437
Subtractive Karatsuba[a] [13]	X	2987
Consecutive operand caching[b] [3]	O	4042
Subtractive Karatsuba[b] [3]	O	3597
This paper[b]	O	2958

[a]Multiplication (only

[b]Modular multiplication over NIST P_{192}

multiplication using the Karatsuba method achieves better performance than other methods for 192-bit modular multiplication over NIST P_{192}.

The Karatsuba multiplication (only) [13] needs 241 LD/LDD instructions, 108 ST/STD instructions, 46 PUSH instructions, and 21 POP instructions. Our modular multiplication requires 212 LD/LDD instructions, 104 ST/STD instructions, 20 PUSH instructions, and 20 POP instructions. Even though our implementation includes a reduction step, it requires fewer LDD/STD instructions and PUSH instructions. This is due to the fact that we can reduce the redundant memory access effectively using duplicate intermediate results of multiplication which are generated from combining Karatsuba multiplication with reduction on RSR.

In [3], Liu et al. present two types of implementation for modular multiplication over NIST P_{192} using consecutive operand caching and Karatsuba method. Bu comparison, our work is about 26% faster than the one using consecutive operand caching method which requires 4042 cycles. The other one applies Karatsuba method of [13] for modular multiplication and requires 3597 cycles which is the previous best result. Our work saves 17% cycles than that and even faster than the multiplication (only) in [13]. Our modular multiplication achieves the best speed record for 192-bit modular multiplication over NIST prime P_{192} on the 8-bit AVR ATmega microcontroller.

In Table 2, we also compare the performance of the modular multiplications in PKCs on 8-bit AVR processor. The basic operation underlying RSA is modular exponentiation where the complexity of the exponentiation is decided by the size of modulus and the exponent. Chinese Remainder Theorem (CRT) can be utilized to reduce the

Table 2 Comparison of modular multiplications in PKCs on 8-bit ATmega128 processor

Literature	Input size	PKCs	Cycle counts
In [21]	512-bit	RSA-1024	65,649
In [20]	160-bit	ECC (OPFs)	3237
This paper	192-bit	ECC (NIST P_{192})	2958

size of both modulus and the exponent. For example, the exponentiation of RSA-1024 can be decomposed into two 512-bit modular exponentiations by applying CRT where 512-bit modular multiplication can be used instead of 1024-bit modular multiplication to speed up by a factor of four. The 512-bit modular multiplication is most time-consuming operation in RSA-1024 where Montgomery reduction [19] is commonly used to avoid trial division by using simple shift instruction which accelerates reduction operation. For comparison between RSA and ECC, we choose 160-bit key size of ECC system to achieve comparable security level to RSA-1024. The 160-bit ECC implementation in [20] uses Optimal Prime Fields(OPFs) which are represented by low-weight primes. This specific primes allow for simplification of the modular arithmetic. The result of 160-bit modular multiplication makes a big difference with the result of 512-bit modular multiplication used in RSA-1024 [21]. This difference shows why ECC is better choice for the implementation of PKCs on constrained devices. Our 192-bit modular multiplication is even faster than the 160-bit modular multiplication which uses also Montgomery method to perform reduction efficiently. In our work, instead of using Montgomery reduction, we focused on merging reduction operation into Karatsuba multiplication having two duplicated groups of intermediate results which result in reduction in the memory access.

5 Conclusion

Many studies focus on improving the performance of multi-precision multiplication, which is the most critical factor for an efficient ECC implementation on constrained devices. Among various methods for multi-precision multiplications, the Karatsuba multiplication of Hutter and Schwabe in [13] is to be considered the best choice for an efficient implementation on the 8-bit AVR ATmega family of microcontrollers. However, these studies do not consider the reduction operation followed by multiplication thoroughly although this process introduces significant amount of memory access for recalling the multiplication result.

In this paper, we concentrated on reducing unnecessary memory access related to accumulation of intermediate results by merging reduction process into multiplication. In this context, we proposed a new integer representation named range shifted representation and optimized the modular multiplication over 192-bit NIST prime P_{192}. Our work shows that Karatsuba multiplication with reduction on RSR generates duplicated intermediate results during accumulation which have many advantages for an efficient implementation of modular multiplication. Careful ordering of computation routines also saves load/save instructions. Our proposed modular multiplication surpasses the multiplication (only) in [13] and achieved a new speed record for 192-bit modulo multiplication over NIST prime P_{192} on an 8-bit AVR ATmega processor.

Acknowledgements This work was supported by Institute for Information and communications Technology Planning and Evaluation (IITP) grant funded by the Korea government (MSIT). (No. 2019-0-00033, Study on Quantum Security Evaluation of Cryptography based on Computational Quantum Complexity).

References

1. Gura N, Patel A, Wander A, Eberle H, Shantz SC (2004) Comparing elliptic curve cryptography and RSA on 8-bit CPUs. In: Joye M, Quisquater JJ (eds) Cryptographic hardware and embedded systems (lecture notes in computer science), vol 3156. Springer, Berlin, pp 119–132
2. Liu A, Ning P (2008) TinyECC: a configurable library for elliptic curve cryptography in wireless sensor networks. In: Proceedings of the 7th International Conference on Information Processing in Sensor Networks (IPSN), pp 245–256
3. Liu Z, Seo H, Großschädl J, Kim H (2016) Efficient implementation of NIST-compliant elliptic curve cryptography for 8-bit AVR-based sensor nodes. IEEE Trans Inf Forensics Secur 11(7):1385–1397
4. Seo SC, Seo H (2018) Highly efficient implementation of NIST-compliant Koblitz curve for 8-bit AVR-based sensor nodes. IEEE Access 6:67637–67652
5. Comba PG (1990) Exponentiation cryptosystems on the IBM PC. IBM Syst J 29(4):526–538
6. Scott M, Szczechowiak P (2007) Optimizing multiprecision multiplication for public key cryptography. Cryptology ePrint archive, report 2007/299
7. Szczechowiak P, Oliveira LB, Scott M, Collier M, Dahab R (2008) NanoECC: testing the limits of elliptic curve cryptography in sensor networks. In: Proceedings of the International Conference on Wireless Sensor Networks'08). Springer, Berlin, pp 305–320
8. Uhsadel L, Poschmann A, Paar C (2007) Enabling full-size public-key algorithms on 8-bit sensor nodes. In: Proceedings of the International Conference on Security and Privacy in Ad-Hoc and Sensor Networks (ESAS'07). Springer, Berlin, pp 73–86
9. Yang Z, Johann G (2011) Efficient prime-field arithmetic for elliptic curve cryptography on wireless sensor nodes. In: Proceedings of the International Conference on Computer Science and Network Technology, pp 459–466
10. Hutter M, Wenger E (2011) Fast multi-precision multiplication for publickey cryptography on embedded microprocessors. In: Preneel B, Takagi T (eds) Cryptographic hardware and embedded systems (lecture notes in computer science), vol 6917. Springer, Berlin, pp 459–474
11. Seo H, Kim H (2012) Multi-precision multiplication for public-key cryptography on embedded microprocessors. In: MotiYung DHL (ed) Information security applications, vol 7690. Lecture notes in computer science. Springer, Berlin, pp 55–67
12. Seo H, Kim H (2013) Optimized multi-precision multiplication for public-key cryptography on embedded microprocessors. Int J Comput Commun Eng 2(3):255
13. Hutter M, Schwabe P (2015) Multiprecision multiplication on AVR revisited. J Cryptogr Eng 5(3):201–214
14. Miller VS (1985) Use of elliptic curves in cryptography. In: Proceedings of the Conference on the Theory and Application of Cryptographic Techniques, Santa Barbara, CA, USA. Springer, Berlin, pp 417–426 (1985)
15. Koblitz N (1987) Elliptic curve cryptosystems. Math Comput 48(177):203–209
16. National Institute of Standards and Technology (1999) Recommended elliptic curves for federal government use. http://csrc.nist.gov/encryption/dss/ecdsa/NISTReCur.pdf
17. Karatsuba AA, Ofman YP (1963) Multiplication of multidigit numbers on automata. Sov Phys Dokl 7(7):595–596
18. Liu Z, Seo H, Kim H (2016) A synthesis of multi-precision multiplication and squaring techniques for 8-bit sensor nodes: state of-the-art research and future challenges. J Comput Sci Technol 31(2):284–299
19. Montgomery PL (1985) Modular multiplication without trial division. Math Comput 44(170):519–521
20. Liu Z, Großschädl J, Wong DS (2014) Low-weight primes for lightweight elliptic curve cryptography on 8-bit AVR processors. In: Information Security and Cryptology—INSCRYPT 2013. LNCS (2014)

21. Liu Z, Großschädl J, Kizhvatov I (2010) *Efficient and Side-Channel Resistant RSA Implementation for 8-bit AVR Microcontrollers.* In: Workshop on the Security of the Internet of Things—SOCIOT 2010, 1st International Workshop, Tokyo, Japan, November 29. IEEE Computer Society, Los Alamitos

Affiliations

Dong-won Park[1] · Seokhie Hong[1] · Nam Su Chang[2] · Sung Min Cho[3]

Dong-won Park
wony86a@gmail.com

Nam Su Chang
nschang@sjcu.ac.kr

Sung Min Cho
muji0828@korea.ac.kr

[1] Center for Information Security Technologies (CIST), Korea University, Seoul 02841, South Korea

[2] Sejong Cyber University, Seoul 05000, South Korea

[3] Crypt & Tech, Seoul 02841, South Korea