applied sciences

MDPI

Article

Fast Implementation of NIST P-256 Elliptic Curve Cryptography on 8-Bit AVR Processor

Dong-won Park [1], Nam Su Chang [2], Sangyub Lee [3] and Seokhie Hong [1,*]

1 Graduate School of Information Security, Institute of Cyber Security and Privacy (ICSP), Korea University, Seoul 02841, Korea; wony86a@gmail.com
2 Department of Information Security, Graduate School of Information Security, Sejong Cyber University, Seoul 05000, Korea; nschang@sjcu.ac.kr
3 National Institute for Mathematical Sciences, Daejeon 34047, Korea; sylee@nims.re.kr
* Correspondence: shhong@korea.ac.kr; Tel.: +82-10-6201-6348

Received: 26 October 2020; Accepted: 6 December 2020; Published: 9 December 2020

Abstract: In this paper, we present a highly optimized implementation of elliptic curve cryptography (ECC) over NIST P-256 curve for an 8-bit AVR microcontroller. For improving the performance of ECC implementation, we focus on optimizing field arithmetics. In particular, we optimize the modular multiplication and squaring method exploiting the state-of-the-art optimization technique, namely range shifted representation (RSR). With optimized field arithmetics, we significantly improve the performance of scalar multiplication and set the speed record for execution time of variable base scalar multiplication over NIST P-256 curve. When compared with previous works, we achieve a performance gain of 17.3% over the best previous result on the same platform. Moreover, the execution time of our result is even faster than that over the NIST P-192 curve of the well-known TinyECC library. Our result shows that RSR can be applied to all field arithmetics and evaluate the impact of the adoption of RSR over the performance of scalar multiplication. Additionally, our implementation provides a high degree of regularity to withstand side-channel attacks.

Keywords: elliptic curve cryptography; NIST curves; software implementation; 8-bit AVR microcontroller

1. Introduction

Wireless sensor networks (WSNs) that consist of a numerous number of resource-constrained sensor devices have attracted substantial attraction due to the rapid advancement of Internet of Things (IoT). In various IoT applications, such as monitoring physical and environmental conditions (temperature, sound, and pollution levels), battlefield reconnaissance, home automation, etc., many constrained devices are employed as wireless sensor nodes for their low cost and energy efficiency. When compared with traditional cable networks, it is difficult to ensure secure and reliable communications in WSNs, since those wireless sensor nodes are often deployed in unattended environments. Hence, it can be easily accessed and manipulated by malicious adversaries. Thus, the cryptographic mechanism is required in order to provide sufficient security in WSNs. However, it is hard to deploy cryptographic schemes (especially of public key cryptography) on wireless sensor nodes due to their construction in resources, such as computation power, memory, energy, and even storage space, since they usually assumed to be operated with battery-power. For example, MICAz mote, which is recognized as one of most widely used constrained 8-bit devices, is equipped with an AVR ATmega128 processor that has 128 KB of programmable flash memory and 4 KB of RAM with a clock frequency of 7.3728 MHz.

In early days, it is considered that Public-Key Cryptosystems (PKCs) are infeasible in resource-constrained devices due to its significant amount of computation. In order to overcome the restrictions of resource-constrained device, a lot of cryptograpy techniques have been proposed to apply PKCs for securing communication in WSNs [1–4].

When compared to conventional PKCs, Elliptic curve cryptography (ECC) has a smaller key size providing equivalent security. Hence, ECC is regarded as a better choice for WSNs than other PKCs, such as Rivest-Shamir-Adleman (RSA) and Digital Signature Algorithm (DSA). For example, the RSA scheme with 1024-bit key provides the same level of security in ECC with the 160-bit key. In addition, it is considered that a scalar multiplication, whic is the most expensive operation in ECC, only requires 5% of the execution time of main operation in RSA [1,3,5].

A scalar multiplication requires elliptic curve point arithmetic operations, such as point addition and doubling. These point arithmetic operations require field operations, including multiplication, squaring, reduction, addition, and inversion. In particular, it is considered that the most time-consuming operation in field operation is multiplication accounting for almost 80% of execution time of scalar multiplication. Hence, many researches have naturally focused on optimizing the multiplication technique in order to improve the performance of scalar multiplication.

1.1. Related Work

In [1], Gura et al. presented the first ECC implementation on an 8-bit AVR processor. Before this result, it is believed that ECC is infeasible to be implemented for constrained devices, since it requires a significant amount of computation. This work focuses on numerous data transport to and from memory during multi-precision multiplication of large integers in limited register space. For minimizing the number of load instruction on a processor with a large register file, they proposed the new hybrid method, which combines the advantage of conventional byte-wise multiplication techniques, such as the row-wise and the column-wise methods. They could compute a scalar multiplication in 2.19 s for NIST P-224 curve at a frequency of 8 MHz, i.e., 17.52×10^6 clock cycles on AVR processor. In 2008, Liu et al. presented TinyECC [2], which is a first well-known and widely used cryptographic library containing various platforms, such as 8-bit AVR-based, 16-bit MSP-based, and 32-bit ARM-based devices. TinyECC includes a set of optimization switches for flexible configuration by developer's needs, which gives different execution time and memory consumption. TinyECC supports all 128-bit, 160-bit, and 192-bit NIST-recommended elliptic curves. TinyECC is evaluated on MICAz, TelosB, Tmote Sky, and Imote2, which provide the measurement of performance and resource consumption from the low-end devices to high-end devices. They also used the hybrid method for optimizations for the multiplication and squaring operation.

Recently, researches on ECC implementations over NIST prime curves on an 8-bit AVR processor were conducted in [3,6]. Liu et al. implemented ECC over NIST P-192 curve on 8-bit AVR processor [3] and achieved an execution time of 8.62×10^6 clock cycles for scalar multiplication, which is the best result. With respect to NIST P-256 curve, Zhou et al. [6] achieved the fastest performance, with an execution time of 25.38×10^6 clock cycles. Both of the works adopted the Karatsuba method that was proposed by Hutter and Schwabe [7] for the optimization of multiplication and squaring operations, which leads to the best performance with respect to scalar multiplication. Hence, this Karatsuba method is considered to be the best way for the implementation of multiplication.

In summary, many researches showed that ECC implementations can be feasible for resource-constrained devices and their main purpose is to optimize the performance of a scalar multiplication by using fast multiplication techniques. However, they did not consider modular reduction operation as a crucial part for improving performance, despite it always following every multiplication and squaring. The reduction operation introduces huge memory access by recalling the previous results of multiplication and squaring. In the context of modular multiplication, Park et al. [8] introduced new Karatsuba technique while using new integer representation, namely, Range Shifted Representation (RSR). Their work achieved the best performance for 192-bit modular multiplication over NIST P-192 prime. Because their optimization technique is highly prime-dependent, different optimization approaches are required to adopt their technique to modular multiplication over other primes. Moreover, in order to apply RSR to other field operations i.e., squaring, addition, and inversion, new attempts need to be considered for efficient implementation.

In addition, resistance against side channel attacks is required to ensure secure ECC implementation. Traditional ECC implementations did not consider any countermeasures for preventing side-channel attacks, such as Simple Power Analysis (SPA), which might cause security threats on embedded devices [9–11]. SPA attacks usually exploit the conditional statements in ECC implementation, which leak secret information. In practice, SPA attacks use conditional subtraction in field operation and irregular execution pattern of scalar multiplication, which reveal key related information [12–16]. Hence, constant field operation and regular scalar multiplication algorithms are required for developing resistance to SPA attack.

1.2. Our Contributions

The contributions of this paper are summarized, as follows:

- We present an efficient algorithm for 256-bit modular multiplication over NIST P-256 prime for 8-bit AVR processor taking advantage of the optimization technique of modular multiplication on RSR in [8]. Because the technique presented in [8] is highly prime-dependent, it cannot be directly adopted to other prime field multiplications and it requires different approaches to other bigger operand lengths considering the limited number of registers. Because the 256-bit intermediate result of multiplication occupies 32 registers, it cannot be held in working registers at a time during the modular multiplication. Hence, careful scheduling of accumulation for processing the intermediate result is required in order to avoid unnecessary memory access, i.e., load/store instructions, which is known to be the most time-consuming instructions for modular multiplication on constrained devices. In order to extend the optimization technique on RSR in [8] for the 256-bit modular multiplication, we propose a new algorithm, which divides the reduction process into two 128-bit parts and carefully schedules the accumulation of the intermediate results of multiplication. Consequently, we significantly reduce the number of load/store instructions during modular multiplication.
- In [8], they only introduce modular multiplication on RSR not including other field operations, such as squaring, addition, and inversion. We focus on applying all field operations on RSR in order to improve the performance of scalar multiplication. Firstly, we propose a new modular squaring over NIST P-256 prime, which is based on the optimization of our 256-bit modular multiplication resulting in a dramatic optimization. This modular squaring set the speed record for 256-bit modular squaring over NIST P-256 prime for 8-bit AVR processor, providing about 24% of improved performance as compared with the state-of-the-art method. Moreover, we propose a modular addition on RSR providing a constant execution time in order to protect from SPA attacks. Finally, we present conversion algorithms from the original integer representation to RSR, and vice versa, which are required for applying RSR to field operations.
- We present a highly optimized ECC implementation over NIST P-256 curve providing 128-bit security on an 8-bit AVR ATmega128 processor. The result of our work only requires 20.98×10^6 cycles, which is faster than previous work in [6]. Our proposed implementation achieves the best performance for P-256 curve on an AVR processor.

The rest of this paper is organized, as follows. In Section 2, we briefly review the basic ECC, including NIST curve P-256, and also describe the main features of the 8-bit AVR ATmega128 microcontroller. In Section 3, we overview RSR for modular multiplication applied to the NIST P-256 prime. In Section 4, we describe the optimization technique for field operations on RSR. In particular, we propose modular multiplication and squaring method that is optimized for 8-bit AVR processor. Section 5 compares our implementation of scalar multiplication with other previous works. Finally, we conclude this work in Section 6.

2. Preliminaries

2.1. Elliptic Curve Cryptography

ECC has been the most popular public key cryptosystem since Neal Koblitz [17] and S. Miller [18] first introduced EEC in 1985, due to the short key sizes and low computational cost as compared to RSA, DSA, or Diffie-Hellman. Its security is based on Elliptic Curve Discrete Logarithm Problem (ECDLP), which is hard to be solved by general purpose subexponential algorithms. An elliptic curve E over finite field $\mathbb{F}_P$ can be defined by a short Weierstraß equation, as follows:

$$y^2 = x^3 + ax + b,$$

where $a, b \in \mathbb{F}_P$ and $4a^3 + 27b^2 \neq 0$. It is common practice to choose the curve parameter a as -3 in order to improve the performance of point arithmetic in scalar multiplication.

In 1999, NIST proposed five prime-field curves for standardization [19], which adopts this approach. Hence, the NIST curves E can be defined by a short Weierstraß equation as $y^2 = x^3 - 3x + b$. The NIST P-256 curve use prime field $\mathbb{F}_{P_{256}}$, defined by prime $P_{256} = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$. This prime has a specific form, such that the prime is represented with addition or subtraction of several number of power of 2 and all of the exponents are multiples of 8, which are the advantages of manipulating on 8-bit devices. With this special form, modular reduction operation can be easily conducted by some additions while using the congruence $2^{256} \equiv 2^{224} - 2^{192} - 2^{96} + 1 \pmod{P_{256}}$.

2.2. 8-Bit AVR ATmega Microcontroller

We used the ATmega128 as our target platform, which is a representative of the AVR family based on a modern highly structured RISC design. The ATmega128 features 128 KB of programmable flash memory and 4 KB SRAM, a 4 KB EEPROM, a 9-channel 10-bit A/D converter, and a JTAG interface for on-chip debugging. This AVR microcontroller provides 133 various instructions; most of them take a single clock cycle, but multiplication and memory access (Load, Save) instructions require two clock cycles. There is a set of 32 × 8-bit working registers, where six registers (R26-R31) are for addressing pointers X, Y, and Z, and the remaining 26 registers (R0-R25) are for general purposes. Especially, the least significant registers (R0-R1) holds the result of 8 × 8-bit multiplication.

There are many development and compile tools that are available for the ATmel family. The ATmel corporation offers a free AVR Studio development environment, which includes a compiler, an assembler, and a convenient graphical simulator for Visual Studio environments.

3. Range Shifted Representation for Modular Multiplication

In this section, we introduce the RSR proposed in [8] and apply it to 256-bit integer representation. We assume that input operand size is 256-bit for the simple explanation of RSR.

3.1. Range Shifted Representation

We can apply RSR to represent 256-bit integers X, Y range from 2^{-128} to 2^{128-1}, and their multiplication $Z = X \cdot Y$ with 8-bit word size ($W = 2^8$), as the following:

$$X = \sum_{i=0}^{i=31} x_i W^{i-16} = x_0 W^{-16} + \cdots + x_{31} W^{15}, \tag{1}$$

$$Y = \sum_{i=0}^{i=31} y_i W^{i-16} = y_0 W^{-16} + \cdots + y_{31} W^{15}, \tag{2}$$

$$Z = X \cdot Y = \sum_{i=0}^{i=63} z_i W^{i-32} = z_0 W^{-32} + \cdots + z_{63} W^{31}, \tag{3}$$

where $x_i, y_i, z_i \in [0, 2^8 - 1]$. The result Z of multiplication $X \cdot Y$ is expanded to both sides, where the shape of it is symmetric with respect to W^0. In order to reduce the result Z into the range from 2^{-128} to $2^{128} - 1$, we use shifted prime $P_{256} \cdot 2^{-128} = 2^{128} - 2^{96} + 2^{64} + 2^{-32} - 2^{-128}$. Subsequently, we can reduce it at both sides, where $z_0 W^{-32} + z_1 W^{-31} + \cdots + z_{15} W^{-15}$ and $z_{48} W^{12} + z_{37} W^{13} + \cdots + z_{47} W^{23}$ in Equation (3) is reduced while using the congruence relations

$$\begin{aligned} W^{16} &\equiv W^{12} - W^8 - W^{-4} + W^{-16} \pmod{P_{256} \cdot W^{-16}}, \\ W^{-32} &\equiv 1 - W^{-4} + W^{-8} + W^{-20} \pmod{P_{256} \cdot W^{-16}}. \end{aligned} \tag{4}$$

For example, let

$$\begin{aligned} Z_A &= z_0 + z_1 W + \cdots + z_{15} W^{15}, \\ Z_B &= z_{16} + z_{17} W + \cdots + z_{47} W^{31}, \\ Z_C &= z_{48} + z_{49} W + \cdots + z_{63} W^{15}, \\ Z &= Z_A \cdot W^{-32} + Z_B \cdot W^{-16} + Z_C \cdot W^{16}, \end{aligned}$$

where $z_i \in [0, 2^8 - 1]$. Subsequently, we can reduce Z, as follows:

$$\begin{aligned} Z \equiv & Z_A - Z_A \cdot W^{-4} + Z_A \cdot W^{-8} + Z_A \cdot W^{-20} \\ & + Z_B \cdot W^{-16} \\ & + Z_C \cdot W^{12} - Z_C \cdot W^8 - Z_C \cdot W^{-4} + Z_C \cdot W^{-16} \\ & \pmod{P_{256} \cdot W^{-16}}. \end{aligned} \tag{5}$$

Note that this is not a complete reduction. The part of result in (5) is not included in the range from 2^{-128} to 2^{128-1}. Here, we omit the complete step for simplicity.

3.2. Modular Multiplication on RSR

We perform modular multiplication on RSR using the subtractive Karatsuba method, as in [8]. Let $X, Y \in \mathbb{F}_{P_{256}}$ be represented with RSR, and $Z = X \cdot Y$.

Let

$$X_A = x_0 + x_1 W + \cdots + x_{15} W^{15}, \tag{6}$$

$$X_B = x_{16} + x_{17} W + \cdots + x_{31} W^{15}, \tag{7}$$

$$Y_A = y_0 + y_1 W + \cdots + y_{15} W^{15}, \tag{8}$$

$$Y_B = y_{16} + y_{17} W + \cdots + y_{31} W^{15}, \tag{9}$$

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

Subsequently, X, Y, Z can be represented as

$$X = X_A \cdot W^{-16} + X_B, \tag{10}$$

$$Y = Y_A \cdot W^{-16} + Y_B, \tag{11}$$

$$\begin{aligned} Z = X \cdot Y &= (X_A \cdot W^{-16} + X_B) \cdot (Y_A \cdot W^{-16} + Y_B) \\ &= X_A Y_A \cdot W^{-32} + X_B Y_B + (X_A Y_B + X_B Y_A) W^{-16} \\ &= X_A Y_A \cdot W^{-32} + X_B Y_B \\ &+ (X_A Y_A + X_B Y_B - (X_A - X_B) \cdot (Y_A - Y_B)) W^{-16}. \end{aligned} \tag{12}$$

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

$$\begin{aligned} L =& X_A Y_A = l_0 + \cdots + l_{31} W^{31} = L_A + L_B \cdot W^{16}, \\ &(L_A = l_0 + \cdots + l_{15} W^{15}, L_B = l_{16} + \cdots + l_{31} W^{15}) \end{aligned} \tag{13}$$

$$\begin{aligned} H =& X_B Y_B = h_0 + \cdots + h_{31} W^{31} = H_A + H_B \cdot W^{16}, \\ &(H_A = h_0 + \cdots + h_{15} W^{15}, H_B = h_{16} + \cdots + h_{31} W^{15}) \end{aligned} \tag{14}$$

$$M = (X_A - X_B) \cdot (Y_A - Y_B) = m_0 + \cdots + m_{31} W^{31}. \tag{15}$$

Afterwards, Z can be represented by L, H, M as

$$\begin{aligned} Z &\equiv L \cdot W^{-32} + H + (L + H - M) W^{-16} \\ &\equiv (L_A + L_B \cdot W^{16}) W^{-32} + H_A + H_B \cdot W^{16} + (H_A \\ &+ H_B \cdot W^{16} + L_A + L_B \cdot W^{16} - M) W^{-16} \\ &\pmod{P_{256} \cdot W^{-16}}. \end{aligned} \tag{16}$$

In Equation (16), only two parts, $L_A \cdot W^{-32}$ and $H_B \cdot W^{16}$, are overflow on both sides of our range. By reducing them, we can compute $Z \pmod{P_{256} \cdot W^{-16}}$, as follows:

$$\begin{aligned} Z &\equiv L_A - L_A \cdot W^{-4} + L_A \cdot W^{-8} + L_A \cdot W^{-20} \\ &+ L_B \cdot W^{-16} + H_A + H_B \cdot W^{12} - H_B \cdot W^{8} \\ &- H_B \cdot W^{-4} + H_B \cdot W^{-16} + (H_A + H_B \cdot W^{16} \\ &+ L_A + L_B \cdot W^{16} - M) W^{-16} \\ &\equiv -L_A \cdot W^{-4} + L_A \cdot W^{-8} + L_A \cdot W^{-20} \\ &+ H_B \cdot W^{12} - H_B \cdot W^{8} - H_B \cdot W^{-4} \\ &+ (L_A + L_B + H_A + H_B) \\ &+ (L_A + L_B + H_A + H_B - M) W^{-16} \\ &\pmod{P_{256} \cdot W^{-16}}. \end{aligned} \tag{17}$$

In Equation (17), $(L_A + L_B + H_A + H_B)$ is computed twice. Hence, only one computation is required for these duplicated accumulations and the result can be used twice.

4. Optimization of Finite Field Operation

4.1. Modular Multiplication

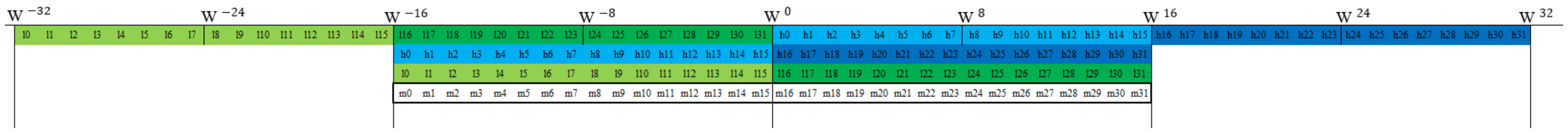
We propose a 256-bit modular multiplication over $\mathbb{F}_{P_{256} \cdot W^{-16}}$, as shown in Algorithm 1, which is composed of a *product part*, an *accumulation part*, and two *reduction parts*. At first, three 128-bit multiplications are computed for L, H, and M, as represented in Equations (13)–(15), which basically follows the same scheduling of 128-bit Karatsuba multiplication, as in [7]. Thanks to the proposed technique presented in [8], in *accumulation part*, $(L_A + L_B + H_A + H_B)$ is only computed once for the duplicated accumulations in Equation (17) and saved in 17 registers, which are represented by $(T, carry_2) = (t_0, \ldots, t_{15}, carry_2)$. The result can be used twice in each *reduction part*. We cannot hold the intermediate result of 256-bit multiplication for reduction operation due to the limited number of registers. Thus, the reduction process is divided into two 128-bit size of *reduction parts*. In *reduction part* 1, the left half side of intermediate result is computed for a reduction of L_A and H_B, whereas *reduction part* 2 computes the right half side, as shown in Figure 1b.

We rearrange the order of computation from $L \rightarrow H \rightarrow M$ in Algorithm 2 to $M \rightarrow L \rightarrow H$ as in [8]. Subsequently, we can directly use H_B without any memory access for the computation of $H_A + H_B$ at Step 7, because it is kept in registers after the previous computation of H at Step 5.

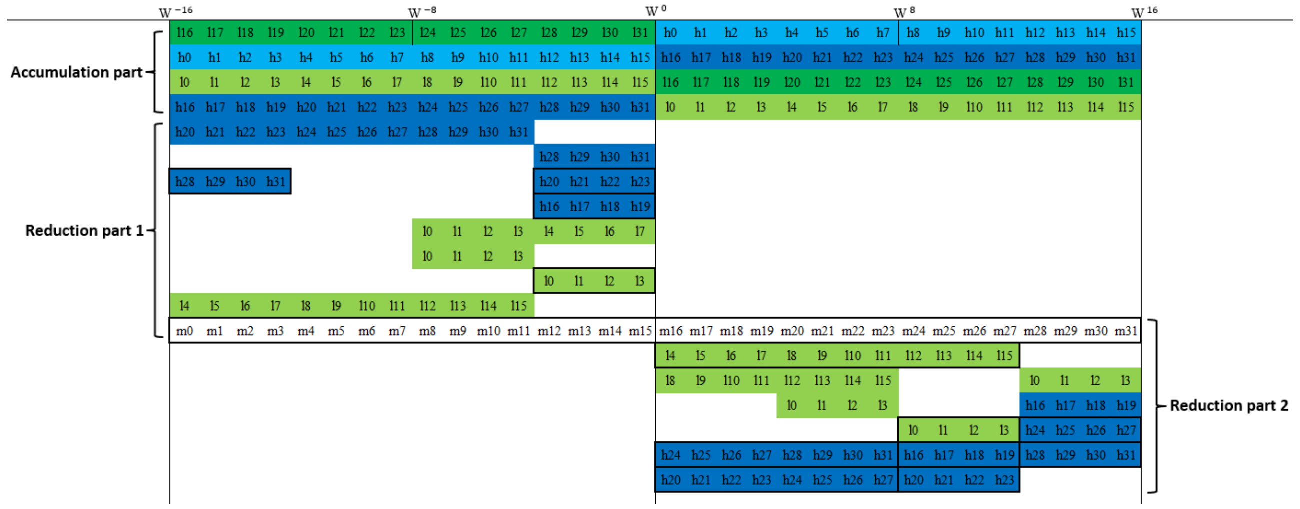
We can find the duplicated computations of $L_A + L_B + H_A + H_B$ in Figure 1b, which can be computed in the *accumulation part* in Algorithm 1, as mentioned in Section 3. We can avoid unnecessary memory access that is related with these computations by reusing the result of first computation in *reduction part* 2. After the *accumulation part*, the result $(t_0, \ldots, t_{15}, carry_2)$ of duplicated computation is stored at Step 9 and loaded in *reduction part* 2. This *accumulation part* corresponds to the Step 4, 9, 10, 11, and 12 in Algorithm 2, except the subtraction with M_A and M_B in Step 9 and 11, respectively. When comparing the number of load and store instructions for manipulating L_A, L_B, H_A, H_B, and T during each accumulation part in Algorithms 1 and 2 the former requires 80 load and 48 store instructions, whereas the latter only requires 48 load and 16 store instructions even when considering additional computation for a reduction of L_A and H_B.

Algorithm 1 256-bit $\times$ 256-bit Karatsuba multiplication with reduction over $\mathbb{F}_{P_{256} \cdot W^{-16}}$

```
Require: $X = (x_0, \ldots, x_{31}), Y = (y_0, \ldots, y_{31})$
Ensure: $Z = X \cdot Y \pmod{P_{256} \cdot W^{-16}} = (z_0, \ldots, z_{31})$
 1: Product part
 2: $|X_A - X_B|$ and $|Y_A - Y_B|$ {Load: $X$,$Y$ ,Store:$|X_A - X_B|$,$|Y_A - Y_B|$}
 3: $M \leftarrow |X_A - X_B| \cdot |Y_A - Y_B| = (m_0, \ldots, m_{31})$ {Load: $|X_A - X_B|$,$|Y_A - Y_B|$ ,Store:$M$}
 4: $L \leftarrow X_A \cdot Y_A = (l_0, \ldots, l_{31})$ {Load: $X_A, Y_A$ ,Store: $L$}
 5: $H \leftarrow X_B \cdot Y_B = (h_0, \ldots, h_{31})$ {Load: $X_B$,$Y_B$ ,Store: $H$}
 6: Accumulation part
 7: $(t_0, \ldots, t_{15}, carry_1) \leftarrow H_A + H_B$ {Load: $H_A$}
 8: $(t_0, \ldots, t_{15}, carry_2) \leftarrow (t_0, \ldots, t_{15}, carry_1) + L_A + L_B$ {Load: $L_A, L_B$}
 9: Store $(T_R, carry_2) = (t_0, \ldots, t_{15}, carry_2)$ {Store: $T_R, carry_2$}
10: Reduction part 1
11: $(T_L, carry_3) \leftarrow (t_0, \ldots, t_{15}, carry_2) + (h_{20}, h_{21}, \ldots, h_{31}, h_{28}, h_{29}, h_{30}, h_{31})$
12: $(T_L, carry_4) \leftarrow T_L - (h_{28}, h_{29}, h_{30}, h_{31}, 0, \ldots, 0, h_{20}, h_{21}, h_{22}, h_{23})$
13: $(T_L, carry_5) \leftarrow (T_L, carry_4) - (0, \ldots, 0 , h_{16}, h_{17}, h_{18}, h_{19})$ {Load: $(h_{16}, h_{17}, h_{18}, h_{19})$}
14: $(T_L, carry_6) \leftarrow (T_L, carry_3) + (0, \ldots, 0, l_0, l_1, \ldots, l_7) + (0,0,0,0,0,0,0,0, l_0, l_1, l_2, l_3, 0,0,0,0)$
    {Load:$(l_0, \ldots, l_7)$}
15: $(T_L, carry_7) \leftarrow (T_L, carry_5) - (0, \ldots, 0, l_0, l_1, l_2, l_3)$
16: $(T_L, carry_8) \leftarrow (T_L, carry_6) + (l_4, l_5, \ldots, l_{15}, 0,0,0,0)$ {Load:$(l_8, l_9, \ldots, l_{15})$}
17: $T_L \leftarrow T_L - M_A$ {Load:$M_A$, Store: $(T_L, carry_7, carry_8)$}
18: Reduction part 2
19: $(T_R, carry_9) \leftarrow T_R - M_B$ {Load:$T_R, M_B$}
20: $(T_R, carry_{10}) \leftarrow (T_R, carry_9) - (l_4, l_5, \ldots, l_{15}, 0,0,0,0)$
21: $(T_R, carry_{11}) \leftarrow T_R + (l_8, l_9, \ldots, l_{15}, 0,0,0,0, l_0, l_1, l_2, l_3)$ {Load:$(l_0, l_1, l_2, l_3)$}
22: $(T_R, carry_{12}) \leftarrow (T_R, carry_{11}) + (carry_8, 0,0,0, l_0, l_1, l_2, l_3, 0,0,0,0, h_{16}, h_{17}, h_{18}, h_{19}, carry_2)$
    {Load:$(h_{16}, h_{17}, h_{18}, h_{19}, carry_2, carry_8)$}
23: $(T_R, carry_{13}) \leftarrow (T_R, carry_{10}) - (carry_7, 0,0,0,0,0,0,0, l_0, l_1, l_2, l_3, h_{24}, h_{25}, h_{26}, h_{27})$
    {Load:$(h_{24}, \ldots, h_{31}, carry_7)$}
24: $(T_R, carry_{14}) \leftarrow (T_R, carry_{13}) - (h_{24}, \ldots, h_{31}, h_{16}, h_{17}, h_{18}, h_{19}, h_{28}, h_{29}, h_{30}, h_{31})$
25: $(T_R, carry_{15}) \leftarrow (T_R, carry_{14}) - (h_{20}, \ldots, h_{27}, h_{20}, h_{21}, h_{22}, h_{23}, 0,0,0,0)$ {Load:$(h_{20}, h_{21}, h_{22}, h_{23})$}
26: Reduce $carry_{12}, carry_{15}$ through $T = (T_L, T_R)$ {Load:$T_L$}
27: Store $T$ to $(z_0, \ldots, z_{31})$ {Store: $T_L, T_R$}
```



(**a**) Result of 256-bit Karatsuba multiplication with Range Shifted Representation (RSR).



(**b**) Reduction scheduling of 256-bit Karatsuba multiplication with RSR.

Figure 1. Process of 256-bit Modular multiplication with RSR.

Algorithm 2 256-bit × 256-bit Karatsuba multiplication in [7]

```
Require: $X = (x_0, \ldots, x_{31}), Y = (y_0, \ldots, y_{31})$
Ensure: $Z = X \cdot Y = (z_0, \ldots, z_{63})$
 1: Product part
 2: $L \leftarrow X_A \cdot Y_A = (l_0, \ldots, l_{31})$ {Load: $X_A, Y_A$ ,Store: $L((z_0, \ldots, z_{15}) = L_A)$}
 3: $H \leftarrow X_B \cdot Y_B = (h_0, \ldots, h_{31})$ {Load: $X_B$,$Y_B$ ,Store: $H$}
 4: $(T, carry_1) \leftarrow L_B + H_A$ {Load: $L_B$,$H_A$ ,Store: $(t_0, \ldots, t_{15})$}
 5: Propagate $carry_1$ to $H_B$ {Load: $H_B$ ,Store: $H_B$}
 6: $|X_A - X_B|$ and $|Y_A - Y_B|$ {Load: $X$,$Y$ ,Store:$|X_A - X_B|$,$|Y_A - Y_B|$}
 7: $M \leftarrow |X_A - X_B| \cdot |Y_A - Y_B| = (m_0, \ldots, m_{31})$ {Load: $|X_A - X_B|$,$|Y_A - Y_B|$ ,Store:$M$}
 8: Combination of $L, H, M$ and $T$
 9: $(z_{16}, \ldots, z_{31}, carry_2) \leftarrow L_A - M_A$ {Load: $L_A$,$M_A$}
10: $(z_{16}, \ldots, z_{31}, carry_3) \leftarrow (z_{16}, \ldots, z_{31}) + T$ {Load: $T$ ,Store: $(z_{16}, \ldots, z_{31})$}
11: $(t_0, \ldots, t_{15}, carry_4) \leftarrow T - M_B$ {Load: $M_B$}
12: $(z_{32}, \ldots, z_{47}, carry_5) \leftarrow (t_0, \ldots, t_{15}) + (h_{16}, \ldots, h_{31})$ {Load: $H_B$ ,Store: $(z_{32}, \ldots, z_{47})$}
13: Propagate $carry_5$ to $H_B$
14: Store $H_B$ to $(z_{48}, \ldots, z_{63})$ {Store: $(z_{48}, \ldots, z_{63})$}
```

Generally, for a reduction operation, the result of multiplication should be first computed. Subsequently, reduction is proceeded with handling of double-sized result, such as 512-bit result $(z_0, \ldots, z_{63})$ in Algorithm 2, which causes huge memory access, since the number of registers is limited. However, for the case of Algorithm 1, we only need to reduce L_A and H_B at both sides, as shown in Figure 1a. In other words, we can easily merge reduction operation into multiplication without requiring the complete result of multiplication, as shown in Figure 1b.

In *reduction part* 1 and 2, we cannot hold L_A and H_B altogether in registers, because 16 working registers are always occupied by the intermediate results T_L or T_R. Hence, repeated load instructions for them are required during *reduction part* 1 and 2. In order to minimize the number of load instructions, we carefully schedule the order of load instructions for L_A and H_B, as shown in Figure 1b. In *reduction part* 1, we can avoid load instructions that are related to H_B, because some of portion of the required result, i.e., $(h_{20}, h_{21}, \ldots, h_{31})$, computed in Step 5 is kept in working registers. Hence, 20 load instructions are required for the remaining portion $(h_{16}, h_{17}, h_{18}, h_{19})$ of H_B and $(l_0, \ldots, l_{15})$. In the starting point of *reduction part* 2, we can avoid load instructions for L_A, since $(l_0, \ldots, l_{15})$ from the end point of *reduction part* 1 can be continuously used. Eventually, an additional 20 load instructions for (l_0, l_1, l_2, l_3) and $(h_{16}, \ldots, h_{31})$ are required in order to complete *reduction part* 2.

At the end of Algorithm 1, the positive carry $carry_{12}$ and the negative carry $carry_{15}$ are reduced to the intermediate result $T = (T_L, T_R)$.

4.2. Modular Squaring

We propose an optimized modular squaring on RSR. Modular squaring consists of a *product part*, an *accumulation part*, and two *reduction parts*, as shown in Algorithm 3. We adopt 2-level subtractive Karatsuba method for implementation of modular squaring. For three 128-bit squaring operations L, H, and M in *product part* in Algorithm 3, the 128-bit Karatsuba technique can be applied, which make use of left shift instruction for the processing of equal cross-product terms. Note that the absolute difference of M in the subtractive Karatsuba multiplication is always positive. Thus, no calculation for the sign of M is required in *product part*. Through combining squaring operation with reduction, we can get the duplicated accumulations as same as $(L_A + L_B + H_A + H_B)$ in modular multiplication (17). This can be used in order to avoid redundant memory access during modular squaring by reusing it in *reduction part* 2 in Algorithm 3. Hence, an *accumulation part* and two *reduction parts* in Algorithm 3 follow the same process of Algorithm 1.

Algorithm 3 256-bit $\times$ 256-bit modular squaring over $\mathbb{F}_{P_{256} \cdot W^{-16}}$

```
Require: $X = (x_0, \ldots, x_{31})$
Ensure: $Z = X^2 \pmod{P_{256} \cdot W^{-16}} = (z_0, \ldots, z_{31})$
1: Product part
2: $X_A - X_B$ {Load: $X$ ,Store:$(X_A - X_B)$}
3: $M \leftarrow (X_A - X_B)^2 = (m_0, \ldots, m_{31})$ {Load: $(X_A - X_B)$ ,Store:$M$}
4: $L \leftarrow X_A^2 = (l_0, \ldots, l_{31})$ {Load: $X_A$ ,Store: $L$}
5: $H \leftarrow X_B^2 = (h_0, \ldots, h_{31})$ {Load: $X_B$ ,Store: $H$}
6: Accumulation part and Reduction part 1,2 as in the Algorithm 1
```

4.3. Modular Addition

Because we use operands $X, Y \in [0, 2^{128} - 1]$ for modular addition, $X + Y = Z \in [0, 2^{129} - 2]$ needs to be subtracted by $P_{256} \cdot W^{-16}$, at most, twice in order to ensure Z becoming less than 2^{128}. If $Z \in [0, 2^{128} - 1]$, there is nothing to do. If $Z \in [2^{128}, 2^{128} + P_{256} \cdot W^{-16} - 1]$, only one subtraction of $P_{256} \cdot W^{-16}$ is needed in order to ensure that the final result is smaller than 2^{128}. If $Z \in [2^{128} + P_{256} \cdot W^{-16}, 2^{129} - 2]$, two subtractions of $P_{256} \cdot W^{-16}$ are necessary. These conditional subtraction statements, depending on the range of Z, cause observable differences in the execution time that can be used to find a secret value by a side-channel attacker [20]. For the constant execution time implementation, we always execute two final subtractions after the addition $X + Y$. The two subtractions may be $-P_{256} \cdot W^{-16}$ or -0.

Because the 256-bit addition result cannot be held in 32 working registers, some parts of it have to be stored and reloaded repeatedly through the following two subtractions. We can reduce memory access by minimizing unnecessary register uses. For example, the addition only consists of 32-byte additions without an additional register for the *carry*-byte that is produced from the last byte addition. The *carry* that is held in carry flag decides whether the next subtraction is $-P_{256} \cdot W^{-16}$ or -0. We only use three registers, $R0, R1$, and $R2$, in order to hold $P_{256} \cdot W^{-16}$, because it can only be expressed with three bytes 0x00, 0x01, and 0xFF in hexadecimal.

Let $R0, R1$, and $R2$ be zeros. Through the following instructions after the addition, we can hold $P_{256} \cdot W^{-16}$ or 0 in $R0, R1$, and $R2$, depending on the carry flag.

$$SBC\ R1\ R0$$
$$ADC\ R2\ R0$$

After the 32-byte additions for $Z = X + Y$, if there is no *carry*, then all three registers are set to zero. Subsequently, the following two subtractions have no effect on the result. If *carry* from the addition exists, the carry flag is set. Subsequently, the subtract-with-borrow instruction (SBC) sets $R1$ as 0xFF and the carry flag is set again. Afterwards, $R2$ is set to 0x01 by the following the add-with-carry (ADC) instruction. Through these instructions, we can hold $P_{256} \cdot W^{-16}$ in $R0, R1$, and $R2$.

After the first 32-byte subtractions for $Z' = Z - P_{256} \cdot W^{-16}$, the carry flag may be set by the *borrow* originating from the last byte subtraction. Note that this new *borrow* means that Z' is smaller than 2^{128}. In other words, no *borrow* means that Z' is bigger than or equal to 2^{128}. Through the following instructions after the first subtraction, we can hold $P_{256} \cdot W^{-16}$ or 0 in $R0, R1$, and $R2$.

$$ADC\ R1\ R0$$
$$SBC\ R2\ R0$$

If *borrow* from the first subtraction exists, then ADC instruction sets $R1$ as 0x00 and carry flag is set again, because $R1$ was set to 0xFF from the previous step. Subsequently, $R2$ is set to 0x00 through SBC instruction and we can hold zeros in $R0, R1$, and $R2$. If there was no *borrow*, $P_{256} \cdot W^{-16}$ remains on the three registers.

Through these simple carry processes, we can always execute one addition and two final subtraction for a constant time of modular addition. Table 1 represents all of the cases of the *carry* and the *borrow*, which decide whether the next subtraction is $-P_{256} \cdot W^{-16}$ or -0.

Table 1. Number of cases of carry and borrow in modular addition.

Range of $Z = X + Y$	*Carry*	First Sub	*Borrow*	Second Sub
$[0, 2^{128} - 1]$	0	-0	0	-0
$[2^{128}, 2^{128} + P_{256} \cdot W^{-16} - 1]$	1	$-P_{256} \cdot W^{-16}$	1	-0
$[2^{128} + P_{256} \cdot W^{-16}, 2^{129} - 2]$	1	$-P_{256} \cdot W^{-16}$	0	$-P_{256} \cdot W^{-16}$

4.4. Modular Inversion

Extended Euclidean Algorithm (EEA) is widely known as an efficient method for modular inversion. However, EEA has a non-constant execution time, depending on input, which can be exploited by side channel attack, such as timing attack. In order to prevent side channel leakage, we implement Fermat's Little Theorem-based inversion, which has a constant execution time. For $a \in \mathbb{F}_{P_{256} \cdot W^{-16}}$, the inversion of a can be computed via $a^{-1} \equiv a^{P_{256}-2} \pmod{P_{256} \cdot W^{-16}}$, which is significantly slower than EEA.

4.5. Conversion of Representation

In order to apply RSR to field operations, conversions from the original representation to RSR and vice versa are required to shift the range of integer representation. Algorithm 4 describes the conversion from the original representation to RSR, where the range is changed from $[0, P_{256}]$ to $[2^{-128}, P_{256} \cdot W^{-16}]$. This conversion consists of one 256-bit addition, two 256-bit subtractions, and final subtraction of $P_{256} \cdot W^{-16}$. This conversion has a negligible amount of clock cycles when compared to a modular multiplication or squaring and is only required for coordinates x and y for the input point of scalar multiplication. Reversely, the conversion from RSR to the original representation is required for the output point of scalar multiplication, where the range is restored from $[2^{-128}, P_{256} \cdot W^{-16}]$ to $[0, P_{256}]$. Three 256-bit addition, two 256-bit subtractions, and final subtraction of P_{256} are required, as shown in Algorithm 5. This conversion also costs negligible clock cycles, since it is only required for coordinates X and Y of the output point.

Algorithm 4 Conversion from original representation to RSR

```
Require: $X = (x_0, \ldots, x_{31}) \in \mathbb{F}_{P_{256}}$
Ensure: $X' = (x'_0, \ldots, x'_{31}) \in \mathbb{F}_{P_{256} \cdot W^{-16}}$
 1: Define 256-bit integers
 2: $S_1 = (x_{16}, x_{17}, \ldots, x_{31}, x_0, x_1, \ldots, x_{15})$
 3: $S_2 = (x_{20}, x_{21}, \ldots, x_{31}, x_{28}, x_{29}, x_{30}, x_{31}, 0, 0, \ldots, 0, x_{16}, x_{17}, x_{18}, x_{19})$
 4: $S_3 = (x_{28}, x_{29}, x_{30}, x_{31}, 0, 0, \ldots, 0, x_{20}, x_{21}, \ldots, x_{31}, x_{20}, x_{21}, \ldots, x_{27})$
 5: $S_4 = (0, 0, \ldots, 0, x_{16}, x_{17}, \ldots, x_{27}, x_{16}, x_{17}, x_{18}, x_{19}, x_{28}, x_{29}, x_{30}, x_{31})$
 6: Return $X' = S_1 + S_2 - S_3 - S_4 \pmod{P_{256} \cdot W^{-16}}$
```

Algorithm 5 Conversion from RSR to original representation

```
Require: $X' = (x'_0, \ldots, x'_{31}) \in \mathbb{F}_{P_{256} \cdot W^{-16}}$
Ensure: $X = (x_0, \ldots, x_{31}) \in \mathbb{F}_{P_{256}}$
 1: Define 256-bit integers
 2: $S_1 = (x'_{16}, x'_{17}, \ldots, x'_{31}, x'_0, x'_1, \ldots, x'_{15})$
 3: $S_2 = (x'_4, x'_5, \ldots, x'_{15}, x'_4, x'_5, \ldots, x'_{15}, 0, 0, 0, 0, x'_0, x'_1, x'_2, x'_3)$
 4: $S_3 = (0, 0, 0, 0, 0, 0, 0, 0, x'_0, x'_1, x'_2, x'_3, 0, 0, 0, 0, 0, 0, 0, 0, x'_0, x'_1, x'_2, x'_3, 0, 0, \ldots, 0)$
 5: $S_4 = (0, 0, 0, 0, 0, 0, 0, 0, x'_0, x'_1, x'_2, x'_3, 0, 0, \ldots, 0)$
 6: $S_5 = (0, 0, \ldots, 0, x'_0, x'_1, x'_2, \ldots, x'_{15}, 0, 0, 0, 0)$
 7: $S_6 = (0, 0, \ldots, 0, x'_0, x'_1, x'_2, x'_3, 0, 0, 0, 0)$
 8: Return $X' = S_1 + S_2 + S_3 + S_4 - S_5 - S_6 \pmod{P_{256}}$
```

5. Optimization of Elliptic Curve Group Arithmetic

5.1. Coordinate System for Point Arithmetic

A scalar multiplication is composed of a two point arithmetic operation, i.e., point addition and doubling. There are several ways to represent the points. Affine coordinate system uses two coordinates x, y and it requires the computation of field inversion in point arithmetic. Because field inversion is a relatively expensive operation, most previous works represent the points in a projective coordinate system. There are plenty of researches that proposed an efficient projective coordinate system, such as Jacobian, Chudnovsky, and mixed coordinate system [21–24]. When comparing these projective coordinates, it is preferable to choose Jacobian coordinates for the fastest pointer arithmetic operations. While using Jacobian coordinates, the point addition requires $4M + 4S + 9A$ (M = modular multiplication, S = modular squaring, A = modular addition) and point doubling requires $8M + 3S + 7A$.

5.2. Co-Z Arithmetic

We choose the co-Z arithmetic, which was proposed by Meloni in [25], for point arithmetic. It provides a very efficient point addition in Jacobian coordinates, where the two involved points share the same Z-coordinate. The co-Z point addition is faster than a point addition in Jacobian coordinates and even faster than a point doubling. Let $P = (X_1, Y_1, Z)$ and $Q = (X_1, Y_1, Z)$, the point addition of P and Q is defined by $P + Q = (X_3, Y_3, Z_3)$, which is computed, as follows:

$$\begin{aligned} A &= (X_2 - X_1)^2,\ B = X_1 A, \\ C &= X_2 A,\ D = (Y_2 - Y_1)^2, \\ E &= Y_1(C - B), \\ X_3 &= D - B - C, \\ Y_3 &= (Y_2 - Y_1)(B - X_3) - E, \\ Z_3 &= Z(X_2 - X_1). \end{aligned}$$

The co-Z point addition has a computational cost of $5M + 2S + 7A$. Moreover, the Z-coordinate of P is updated to be equal with the Z-coordinate of $P + Q$ for free. Through the expression of B and E, we can find out that $B = X_1(X_2 - X_1)^2 = x_1 Z_3^2$ and $E = Y_1(X_2 - X_1)^3 = y_1 Z_3^3$, where $(x_1, y_1) = (X_1/Z^2, Y_1/Z^3)$ represents the affine coordinates of P. Thus, we have $P \equiv (B : E : Z_3)$. Such a free update allows for the subsequent use of co-Z point addition between $P + Q$ and P during scalar multiplication. In [25], it was shown that the conjugate $P - Q$ can be obtained with a small additional cost, sharing the same Z coordinate as $P + Q$. The conjugate $P - Q = (X_3', Y_3', Z_3)$ can be computed, as follows:

$$\begin{aligned} F &= (Y_1 + Y_2)^2, \\ X_3' &= F - (B + C), \\ Y_3' &= (Y_1 + Y_2)(X_3' - B) - E. \end{aligned}$$

With computation of two co-Z points $P + Q$ and $P - Q$, we can compute the co-Z point addition with the cost of $6M + 3S + 16A$.

5.3. Montgomery Ladder for Regular Scalar Multiplication

For resistance against SPA attack, we consider a regular scalar multiplication algorithm, which always performs the same operations in each iteration, regardless of input scalar. For regular scalar multiplication, we choose the Montgomery ladder proposed by *Rivain* in [26] for the short Weierstraß-form elliptic curves over large prime field. This Montgomery algorithm is based on the co-Z arithmetic, which only uses (X, Y) coordinates without the computation of Z coordinate in a

Jacobian coordinate system. The co-Z point addition takes the (X, Y) coordinates of two co-Z points P and Q and computes the (X, Y) coordinates of $P + Q$ and P at the cost of $4M + 2S + 7A$. On the other hand, the co-Z conjugate point addition computes the (X, Y) coordinates of $P + Q$ and of $P - Q$ at the cost of $5M + 3S + 11A$. The Montgomery ladder algorithm computes scalar multiplication with a regular pattern, which always performs the co-Z point addition and the co-Z conjugate point addition in each iteration. Hence, our scalar multiplication provides resistance against SPA attack.

6. Performance Analysis and Comparison

We implement scalar multiplication over NIST P-256 curve on 8-bit AVR ATmega128 processor. More precisely, we optimize field level operations on RSR, including modular multiplication, modular squaring, and modular addition by assembly language. We implement the elliptic curve group operations and scalar multiplication by C language. We use the computer algebra system *Magma* in order to validate the results of scalar multiplication. We simulate all test programs with AVR studio 7.0 in order to obtain execution time in clock cycles.

Table 2 compares the execution time of the proposed field arithmetic operations with the best result previously published. A modular addition and subtraction require 390 clock cycles, while a modular multiplication and squaring need 5355 and 3691 clock cycles, respectively. When compared with the work conducted by Zhou et al. [6], our implementation of modular addition and subtraction is slightly faster, while modular multiplication and squaring provide about 20% and 24% of improved performance, respectively.

Table 2. Comparison of field arithmetic on ATmega128 processor in clock cycles.

Implementation	ADD	SUB	MUL	SQL	INV
Zhou et al. [6]	418	424	6609	4919	1,347,009
This paper	390	390	5355	3691	1,031,682

Table 3 compares our work with other prime field ECC implementations over different NIST curves with respect to execution time, SPA resistance, and code size. Those implementations use NIST primes that range from 192-bit to 256-bit, which provide more than 80-bit security. Our ECC implementation over NIST P-256 curve only requires 20.98×10^6 cycles which is faster than previous work in [6]. Our proposed implementation achieves the best performance for the P-256 curve. The work of Zhou et al. [6] on NIST P-256 curve adopted Montgomery ladder with (x, y)-only co-Z addition for implementation of scalar multiplication as our work. Hence, the gap of performance between the work in [6] and our work comes from our optimization of field level operations, especially modular multiplication and modular squaring.

Table 3. A comparison of various implementations of scalar multiplication over NIST curves on ATmega128 processor in 10^6 clock cycles

Implementation	Curve	SPA Resistance	Clock Cycles	Code Size (bytes)
Liu A. et al.[2]	NIST P-192	No	21.38	19,000 [a]
Liu Z. et al. [3]	NIST P-192	Yes	8.62	26,000 [a]
Gura et al. [1]	NIST P-224	No	17.52	4812
Wenger et al. [27]	NIST P-256	Yes	34.93	16,112
Zhou et al. [6]	NIST P-256	Yes	25.38	13,980
This paper	NIST P-256	Yes	20.98	17,776

[a] roughly measured.

7. Conclusions

In this paper, we introduce the fast ECC implementation over NIST P-256 curve on an 8-bit AVR ATmega128 processor. We focus on applying RSR to all of the field operations in order to improve the performance of ECC. Firstly, we propose 256-bit modular multiplication over $\mathbb{F}_{P_{256} \cdot W^{-16}}$, while taking advantage of the optimization technique in [8]. Secondly, we propose a new modular squaring over NIST P-256 prime. This modular squaring achieves the speed record providing about 24% of improved performance when compared with the state-of-the-art. Besides, we also propose an efficient masking method for modular addition, which ensures constant execution time. Finally, we evaluate the impact of RSR-optimized field operations on the performance of scalar multiplication. When compared to the state-of-the-art, our ECC implementation achieves the best performance of scalar multiplication over NIST P-256 curve with 17.3% improvement.

We focus on eliminating the conditional statements that SPA attack usually exploits in order to resist against SPA attack. Our implementation does not use any conditional statements throughout all field and group operations. Especially, we implement constant modular addition and regular scalar multiplication to prevent SPA attacks.

Our optimized scheduling can be similarly applied to modular multiplication and squaring over other NIST primes. In the future, we will implement ECC over other NIST curves, such as P-384 and P-512.

Author Contributions: Conceptualization, S.H.; Formal analysis, N.S.C.; Project administration, D.-w.P.; Software, S.L.; Writing—original draft, D.-w.P.; Writing—review & editing, S.L. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. NRF-2020R1A2C1011769).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Gura, N.; Patel, A.; Wander, A.; Eberle, H.; Shantz, S.C. Comparing elliptic curve cryptography and RSA on 8-bit CPUs. In *Lecture Notes in Computer Science, Proceedings of the Cryptographic Hardware and Embedded Systems, Cambridge, MA, USA, 11–13 August 2004*; Springer: Berlin/Heidelberg, Germany, 2004; Volume 3156, pp. 119–132.
2. Liu, A.; Ning, P. TinyECC: A configurable library for elliptic curve cryptography in wireless sensor networks. In Proceedings of the 2008 International Conference on Information Processing in Sensor Networks (IPSN 2008), St. Louis, MO, USA, 22–24 April 2008; pp. 245–256.
3. Liu, Z.; Seo, H.; Großschädl, J.; Kim, H. Efficient Implementation of NIST-Compliant Elliptic Curve Cryptography for 8-bit AVR-Based Sensor Nodes. *IEEE Trans. Inf. Forensics Secur.* **2016**, *11*, 1385–1397. [CrossRef]
4. Seo, S.C.; Seo, H. Highly Efficient Implementation of NIST-Compliant Koblitz Curve for 8-bit AVR-Based Sensor Nodes. *IEEE Access* **2018**, *6*, 67637–67652. [CrossRef]
5. Lederer, C.; Mader, R.; Koschuch, M.; Großschädl, J.; Szekely, A.; Tillich, S. Energy-efficient implementation of ECDH key exchange for wireless sensor networks. In *Lecture Notes in Computer Science, Proceedings of the Information Security Theory and Practice. Smart Devices, Pervasive Systems, and Ubiquitous Networks, Brussels, Belgium, 1–4 September 2009*; Springer: Berlin/Heidelberg, Germany, 2009; Volume 5746, pp. 112–127.
6. Zhou, L.; Su, C.; Hu, Z.; Lee, S.; Seo, H. Lightweight implementations of NIST P-256 and SM2 ECC on 8-bit resource-constraint embedded device. *ACM Trans. Embed. Comput. Syst. (TECS)* **2019**, *18*, 1–13. [CrossRef]
7. Hutter, M.; Schwabe, P. Multiprecision multiplication on AVR revisited. *J. Cryptogr. Eng.* **2015**, *5*, 201–214. [CrossRef]
8. Park, D.; Hong, S.; Chang, N.S.; Cho, S.M. Efficient Implementation of Modular Multiplication over 192-bit NIST Prime for 8-bit AVR-Based Sensor Node. *J. Supercomput.* **2020**, 1–9. [CrossRef]

9. Oswald, E. Enhancing simple power-analysis attacks on elliptic curve cryptosystems. In *Lecture Notes in Computer Science, Proceedings of the International Conference on Cryptographic Hardware and Embedded Systems (CHES'02), Redwood Shores, CA, USA, 13–15 August 2002*; Springer: Berlin/Heidelberg, Germany, 2002; pp. 82–97.
10. De Meulenaer, G.; Standaert, F.-X. Stealthy compromise of wireless sensor nodes with power analysis attacks. In *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, Proceedings of the Mobile Lightweight Wireless Systems (MOBILIGHT 2010), Barcelona, Spain, 10–12 May 2010*; Springer: Berlin/Heidelberg, Germany, 2010; pp. 229–242.
11. Mangard, S.; Oswald, E.; Popp, T. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*; Springer: Berlin/Heidelberg, Germany, 2007.
12. Sakai, Y.; Sakurai, K. Simple power analysis on fast modular reduction with NIST recommended elliptic curves. In *Lecture Notes in Computer Science, Proceedings of the International Conference on Information and Communications Security (ICICS'05), Beijing, China, 10–13 December 2005*; Springer: Berlin/Heidelberg, Germany, 2005; pp. 169–180.
13. Stebila, D.; Thériault, N. Unified point addition formulæ and side-channel attacks. In *Lecture Notes in Computer Science, Proceedings of the International Conference on Cryptographic Hardware and Embedded Systems (CHES'06), Yokohama, Japan, 10–13 October 2006*; Springer: Berlin/Heidelberg, Germany, 2006; pp. 354–368.
14. Walter, C.D. Simple power analysis of unified code for ECC double and add. In *Lecture Notes in Computer Science, Proceedings of the International Conference on Cryptographic Hardware and Embedded Systems (CHES'04), Cambridge, MA, USA, 11–13 August 2004*; Springer: Berlin/Heidelberg, Germany, 2004; pp. 191–204.
15. Coron, J.-S. Resistance against differential power analysis for elliptic curve cryptosystems. In *Lecture Notes in Computer Science, Proceedings of the Cryptographic Hardware and Embedded Systems, Worcester, MA, USA, 12–13 August 1999*; Koç, Ç.K., Paar, C., Eds.; Springer: Berlin/Heidelberg, Germany, 1999; Volume 1717, pp. 292–302.
16. Danger, J.-L.; Guilley, S.; Hoogvorst, P.; Murdica, C.; Naccache, D. A synthesis of side-channel attacks on elliptic curve cryptography in smart-cards. *J. Cryptogr. Eng.* **2013**, *3*, 241–265. [CrossRef]
17. Koblitz, N. Elliptic curve cryptosystems. *Math. Comput.* **1987**, *48*, 203–209. [CrossRef]
18. Miller, V.S. Use of elliptic curves in cryptography. In *Lecture Notes in Computer Science, Proceedings of the Conference on the Theory and Application of Cryptographic Techniques, Santa Barbara, CA, USA, 19–22 August 1985*; Springer: Berlin/Heidelberg, Germany, 1985; pp. 417–426.
19. National Institute of Standards and Technology. Recommended Elliptic Curves for Federal Government Use. July 1999. Available online: https://csrc.nist.gov/publications/detail/fips/186/4/final (accessed on 1 October 2020)
20. Walter, C.D.; Thompson, S. Distinguishing exponent digits by observing modular subtractions. In *Lecture Notes in Computer Science, Proceedings of the Topics in Cryptology, San Francisco, CA, USA, 8–12 April 2001*; Springer: Berlin/Heidelberg, Germany, 2001; Volume 2020, pp. 192–207.
21. Hankerson, D.; Menezes, A.J.; Vanstone, S. *Guide to Elliptic Curve Cryptography*; Springer: Berlin/Heidelberg, Germany, 2006.
22. López, J.; Dahab, R. Improved algorithms for elliptic curve arithmetic in GF(2n). In *Lecture Notes in Computer Science, Proceedings of the Selected Areas in Cryptography—SAC '98 (LNCS 1556) [457], Kingston, ON, Canada, 17–18 August 1998*; Springer: Berlin/Heidelberg, Germany, 1999; pp. 201–212.
23. Chudnovsky, D.; Chudnovsky, G. Sequences of numbers generated by addition in formal groups and new primality and factoring tests. *Adv. Appl. Math.* **1987**, *7*, 385–434. [CrossRef]
24. Cohen, H.; Miyaji, A.; Ono, T. Efficient elliptic curve exponentiation using mixed coordinates. In *Lecture Notes in Computer Science, Proceedings of the International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, 18–22 October 1998*; Springer: Berlin/Heidelberg, Germany, 1998; pp. 51–65.
25. Meloni, N. New point addition formulae for ECC applications. In *Lecture Notes in Computer Science, Proceedings of the Arithmetic of Finite Fields, Madrid, Spain, 21–22 June 2007*; Carlet, C., Sunar, B., Eds.; Springer: Berlin/Heidelberg, Germany, 2007; Volume 4547, pp. 189–201.
26. Rivain, M. Fast and regular algorithms for scalar multiplication over elliptic curves. *IACR Cryptol. Eprint Arch.* **2011**, *2011*, 338.

27. Wenger, E.; Unterluggauer, T.; Werner, M. 8/16/32 shades of elliptic curve cryptography on embedded processors. In *Lecture Notes in Computer Science, Proceedings of the International Conference on Cryptology in India, Mumbai, India, 7–10 December 2013*; Springer: Berlin/Heidelberg, Germany, 2013; pp. 244–261.