Montgomery Multiplication for Public Key Cryptography on MSP430X

HWAJEONG SEO, KYUHWANG AN, and HYEOKDONG KWON, Hansung University
ZHI HU, Central South University

For traditional public key cryptography and post-quantum cryptography, such as elliptic curve cryptography and supersingular isogeny key encapsulation, modular multiplication is the most performance-critical operation among basic arithmetic of these cryptographic schemes. For this reason, the execution timing of such cryptographic schemes, which may highly determine that the service availability for low-end microprocessors (e.g., 8-bit AVR, 16-bit MSP430X, and 32-bit ARM Cortex-M), mainly relies on the efficiency of modular multiplication on target embedded processors.

In this article, we present new optimal modular multiplication techniques based on the interleaved Montgomery multiplication on 16-bit MSP430X microprocessors, where the multiplication part is performed in a hardware multiplier and the reduction part is performed in a basic arithmetic logic unit (ALU) with the optimal modular multiplication routine, respectively. This two-step approach is effective for the special modulus of NIST curves, SM2 curves, and supersingular isogeny key encapsulation. We further optimized the Montgomery reduction by using techniques for “Montgomery-friendly” prime. This technique significantly reduces the number of partial products. To demonstrate the superiority of the proposed implementation of Montgomery multiplication, we applied the proposed method to the NIST P-256 curve, of which the implementation improves the previous modular multiplication operation by 23.6% on 16-bit MSP430X microprocessors and to the SM2 curve as well (first implementation on 16-bit MSP430X microcontrollers).

Moreover, secure countermeasures against timing attack and simple power analysis are also applied to the scalar multiplication of NIST P-256 and SM2 curves, which achieve the 8,582,338 clock cycles (0.53 seconds@16 MHz) and 10,027,086 clock cycles (0.62 seconds@16 MHz), respectively. The proposed Montgomery multiplication is a generic method that can be applied to other cryptographic schemes and microprocessors with minor modifications.

CCS Concepts: • **Security and privacy → Public key (asymmetric) techniques**; • **Mathematics of computing → Mathematical optimization**;

Additional Key Words and Phrases: Montgomery multiplication, public key cryptography, MSP430X, software implementation

This work was partly supported by an National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (NRF-2017R1C1B5075742), partly supported as part of the Military Crypto Research Center (UD170109ED) funded by the Defense Acquisition Program Administration (DAPA) and Agency for Defense Development (ADD), and partly supported by an Institute for Information and Communications Technology Promotion (IITP) grant funded by the Korean government (MSIT) (2018-0-00264, Research on Blockchain Security Technology for IoT Services).
Z. Hu was supported by the Natural Science Foundation of China under grants 61972420 and 61602526, and the Hunan Provincial Natural Science Foundation of China under grant 2019JJ50827.
Authors’ addresses: H. Seo (corresponding author), K. An, and H. Kwon, Hansung University, Republic of Korea; emails: {hwajeong84, tigerk9212, korlethean}@gmail.com; Z. Hu, Central South University, China; email: 214097@csu.edu.cn.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2020 Association for Computing Machinery.
1539-9087/2020/05-ART20 $15.00
https://doi.org/10.1145/3387919

ACM Reference format:
Hwajeong Seo, Kyuhwang An, Hyeokdong Kwon, and Zhi Hu. 2020. Montgomery Multiplication for Public Key Cryptography on MSP430X. *ACM Trans. Embed. Comput. Syst.* 19, 3, Article 20 (May 2020), 15 pages.
https://doi.org/10.1145/3387919

1 INTRODUCTION

Internet of Things (IoT) technology has been actively studied in academic and industry fields due to its useful applications, ranging from home automation, surveillance system, and healthcare services. Unlike traditional services, IoT applications are able to provide highly customized services for each customer by recognizing a customer's needs or preferences through seamlessly collected data from remotely deployed IoT devices. However, low-end IoT sensors are usually placed in the public space (building, road, and street), which are easily accessible and manipulated by any legitimate or malicious users. If adversaries illegally capture installed IoT devices and perform sophisticated reverse engineering or any effective hacking measures, the secret information can be easily leaked and misused.

To prevent potential threats, the information of IoT devices should be securely encrypted through the cryptography algorithm, namely public key cryptography (PKC). However, PKC requires complicated computations and low-end IoT devices have very limited resources in terms of storage, energy, and computation power. To meet sufficient service availability, careful optimization techniques of implementations should be considered. PKC instantiation, such as elliptic curve cryptography (ECC) in the pre-quantum case or supersingular isogeny key encapsulation (SIKE) [Azarderakhsh et al. 2017, 2019] in the post-quantum case, highly relies on the efficient implementation of modular multiplication, which is the most expensive operation in finite field arithmetic. For this reason, the execution timing of modular multiplication determines the service availability for low-end microprocessors (e.g., 8-bit AVR, 16-bit MSP430X, and 32-bit ARM Cortex-M embedded processors).

In this article, we present new optimal modular multiplication techniques based on interleaved Montgomery multiplication on 16-bit MSP430X microprocessors, which are effective for the special modulus of NIST curves, SM2 curves, and SIKE. In the proposed interleaved Montgomery multiplication, the multiplication part is performed in a hardware multiplier, whereas the reduction part is performed in a basic arithmetic logic unit (ALU) with an optimal routine. This separated approach ensures the efficient multiplication and reduction. We further improved the Montgomery reduction by using the special optimization for "Montgomery-friendly" prime. This significantly reduces the number of partial products. We applied the proposed method to the NIST P-256 curve, of which the implementation improves the previous modular multiplication operation by 23.6% for 16-bit MSP430X microprocessors and to the SM2 curve, respectively (first implementation on 16-bit MSP430X microcontrollers). Optimized implementations of finite field addition, doubling, subtraction, and inversion are also introduced for each curve. Moreover, secure countermeasures against timing attack and simple power analysis are applied to the scalar multiplication on the NIST P-256 curve and SM2 curve, which achieve the 8,582,338 clock cycles (0.53 seconds@16 MHz) and 10,027,086 clock cycles (0.62 seconds@16 MHz), respectively. Our implementations imply that the proposed Montgomery multiplication would have broad applications to more cryptographic schemes (e.g., SIKE) and microprocessors (e.g., 8-bit AVR and 32-bit ARM Cortex-M).

The rest of this article is organized as follows. In Section 2, we explore the previous works on Montgomery multiplication and the target MSP430X processor. In Section 3, we present implementations of Montgomery multiplication, the NIST P-256 curve, and the SM2 curve on the MSP430X processor. In Section 4, we evaluate proposed implementations on target embedded processors. We conclude the article in Section 5.

ALGORITHM 1: Calculation of the Montgomery Reduction

```
Require: An odd $m$-bit modulus $M$, Montgomery radix $R = 2^m$, an operand $T$ where $T = A \cdot B$ or $T = A \cdot A$
    in the range $[0, 2M - 1]$, and pre-computed constant $M' = -M^{-1} \bmod R$
Ensure: Montgomery product $Z = \mathrm{MonRed}(T, R) = T \cdot R^{-1} \bmod M$
  1: $Q \leftarrow T \cdot M' \bmod R$
  2: $Z \leftarrow (T + Q \cdot M)/R$
  3: if $Z \geq M$ then $Z \leftarrow Z - M$ end if
  4: return Z
```

2 PRELIMINARIES AND RELATED WORKS

2.1 Montgomery Multiplication

The modular reduction in a textbook approach requires an expensive division operation, which is high overheads on low-end devices, since, in general, the low-end device does not support the division instruction. For this reason, many implementations find an alternative way to perform the division operation. The division operation can be transformed to the relatively cheap multiplication operation through the Montgomery reduction, of which a detailed description is given in Algorithm 1.

The Montgomery reduction proceeds as follows: given the intermediate result of multiplication $T = A \cdot B$ or $T = A \cdot A$ (where A and B are operands), T is multiplied by the inverse of modulus (M') and then results are reduced by R and stored into Q. Afterward, the equation $((T + Q \times M)/R)$ is performed. Finally, the calculation of the Montgomery multiplication may require a final subtraction of the modulus (M) to get a reduced result in the range of $[0, M)$. Recently, Gueron and Krasnov [2015] presented the implementation of a Montgomery multiplication–friendly modulus. When the modulus has a special pattern (`0xFFFFFFFF` in hexadecimal), this can be performed in addition and subtraction operations rather than multiplication. The simplified multiplication equation is as follows:

```
0xFFFFFFFF = 0x100000000 - 0x1.
```

The approach is widely used in recent ECC and SIKE implementations and shows the highest performance [Koziel et al. 2016; Faz-Hernández et al. 2018; Jalali et al. 2019; Adalier 2015; Liu et al. 2019; Seo 2020; Seo et al. 2019a, 2019b]. In this article, we further optimize the special feature for NIST P-256 and SM2 curves.

2.2 Target Processors

The MSP430 family of microcontrollers is widely used in low-end IoT services, such as small satellite applications [Peters et al. 2009]. The most popular IoT platforms of MSP430 is `TelosB` and `TmoteSky`. The MSP430 microcontrollers have 16-bit instruction sets and 12 general-purpose registers. General-purpose registers are a main concern to achieve high performance by reducing the number of memory accesses. Specifications for clock frequency and ROM/RAM vary for each model. The MSP430 supports several basic instruction sets, including addition, subtraction, move, and clear operations. The basic instruction usually requires one clock cycle. Detailed basic arithmetic is given in Table 1.

In particular, integer multiplication is carried out with a memory-mapped hardware multiplier. The cost of multiplication is the cost of writing operands and reading the result to/from a memory address of the multiplier in MSP430 embedded processors. Operands can be accessed by four different addressing modes, including register direct, indexed, register indirect, and indirect with auto-increment. Usually the indirect mode shows lower latency than other address modes.

Table 1. Instruction Set Summary for MSP

asm	Operands	Description	Operation	#Clock
ADD	Rr, Rd	Add without Carry	Rd ← Rd+Rr	1
ADDC	Rr, Rd	Add with Carry	Rd ← Rd+Rr+C	1
SUB	Rr, Rd	Sub without Borrow	Rd ← Rd-Rr	1
SUBC	Rr, Rd	Sub with Borrow	Rd ← Rd-Rr-B	1
MOV	Rr, Rd	Move	Rd ← Rr	1
CLR	Rd	Clear	Rd ← 0	1

Advanced MSP430X microcontrollers have recently been introduced. The MSP430X supports 20-bit addressing pointers and a new 32-bit hardware multiplier. This advanced 32-bit hardware multiplier significantly improves the performance of traditional MSP430 implementation based on 16-bit hardware multiplier. The advanced hardware multiplier supports both 32-bit multiplication and 32-bit multiplication and accumulation (MAC) modes. To select multiplication modes, 32-bit operands should be written into specific memory addresses (multiplication: MPY32L, MPY32H; MAC: MAC32L, MAC32H). In particular, the MAC mode efficiently accumulates intermediate results into the result memory (RES0, RES1, RES2, RES3) and sets the carry bit into the carry memory (SUMEXT). The multiplier is triggered by writing 32-bit operands into the operand memory (OP2L, OP2H). Afterward, 65-bit results are accessible through result and carry memory addresses (RES0, RES1, RES2, RES3, SUMEXT). In particular, SUMEXT only keeps one bit overflow.

With an advanced multiplier, many previous works used product-scanning multiplication over an MSP430X hardware multiplier since the MAC mode efficiently accumulates intermediate results in a column-wise fashion. The column-wise multiplication usually ensures the small number of memory accesses [Gouvêa et al. 2012; Seo 2018]. In this work, we also adopted the product-scanning method for multiplication and squaring, but we used a basic ALU for reduction over MSP430X microprocessors for the special modulus.

3 PROPOSED MONTGOMERY MULTIPLICATION

In this section, we explore the efficient implementation of Montgomery multiplication for a special modulus. The target modulus consists of special patterns (0x00000000, 0x00000001, 0xFFFFFFFE, and 0xFFFFFFFF in hexadecimal), which can be performed in simple addition and subtraction operations rather than complicated multiplication. Although we target the NIST P-256 and SM2 curves, the proposed method can be applied to other cryptographic algorithms without limitation, such as SIKE, where the schemes also have similar modulus patterns.

3.1 Constant Modular Addition/Subtraction for a Special Modulus

The finite field addition (respectively, subtraction) operation requires the final subtraction (respectively, addition) with the target modulus after addition (respectively, subtraction) to fit intermediate results in the range of the target field. When the data format is unsigned, the reduced result should not generate overflow bits (respectively, underflow bits). If we perform the conditional final subtraction or addition operation, the execution timing or power consumption becomes varied depending on conditional statements. Since program routines (i.e., the number of point addition or doubling) are highly correlated with secret values, the adversary may get the secret information from conditional execution of final subtraction for reduction [Walter and Thompson 2001].

To avoid conditional statements for modular reduction, the constant-time reduction is introduced by by Liu et al. [2016], which utilizes the conditional reduction (i.e., multi-precision subtraction) of field arithmetic with the mask. After executing the first part of modular addition (i.e., $A + B$), it first generates the 2's complement of carry, which is the *mask* value. When the carry bit is set, the *mask* is always set to `0xFF`. Otherwise, the value is set to zero (`0x00`). The masked modulo is then subtracted without the comparison. In the work of Zhou et al. [2019], the optimized reduction technique for a special modulus is introduced. For the NIST P-256 curve, the modulus $p_{256} = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$ can be rewritten as `0xFFFFFFFF00000001000000000000000000000000FFFFFFFFFFFFFFFFFFFFFFFF` in hexadecimal. For the SM2 curve, the modulus $p_{256} = 2^{256} - 2^{224} - 2^{96} + 2^{64} - 1$ can be rewritten as `0xFFFFFFFEFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF00000000FFFFFFFFFFFFFFFF` in hexadecimal. The modulus consists of only four special patterns, `0xFFFF`, `0xFFFE`, `0x0000`, and `0x0001`, in a 16-bit wise hexadecimal way. Among them, three patterns (i.e., `0xFFFF`, `0xFFFE`, and `0x0001`) are only masked and utilized for modular reduction since the `0x0000` pattern does not require the masked reduction. These features are highly utilized in MSP430X microprocessors. The `0x0001` pattern is obtained from carry, and the `0xFFFF` pattern is obtained through one subtraction with `ZERO` register and `CARRY` register (i.e., `0x0000 - 0x0001 = 0xFFFF`). The remaining pattern (`0xFFFE`) is obtained from the `0xFFFF` pattern. We perform the operation with the `0xFFFF` pattern and then perform one more addition or subtraction to obtain the `0xFFFE` pattern. Details of masked subtraction for NIST P-256 are given in Algorithm 2.

ALGORITHM 2: Masked Subtraction for NIST P-256 on MSP430X

```
Input: carry register (CARRY), temporal
       register (MASK)
Output: result pointer (RESULT)
...
 1: CLR MASK
 2: SUB CARRY, MASK

 3: SUB MASK, 2*0(RESULT)
 4: SUBC MASK, 2*1(RESULT)
 5: SUBC MASK, 2*2(RESULT)
 6: SUBC MASK, 2*3(RESULT)
 7: SUBC MASK, 2*4(RESULT)
 8: SUBC MASK, 2*5(RESULT)
 9: SBC 2*6(RESULT)
10: SBC 2*7(RESULT)
11: SBC 2*8(RESULT)
12: SBC 2*9(RESULT)
13: SBC 2*10(RESULT)
14: SBC 2*11(RESULT)

15: SUBC CARRY, 2*12(RESULT)
16: SBC 2*13(RESULT)

17: SUBC MASK, 2*14(RESULT)
18: SUBC MASK, 2*15(RESULT)
...
```

As with the preceding demonstration, the `MASK` register is first set to zero and then subtracted by the `CARRY` register. When the `CARRY` register is set to 1, the `MASK` register is always set to `0xFFFF` (in hexadecimal). Otherwise, both `CARRY` and `MASK` registers are set to zero. By using an efficient memory-based operation of MSP430X microcontrollers, masked values are directly subtracted from intermediate results (i.e., `RESULT`). For the case of modular subtraction, the borrow bit is used for the `MASK` register, and the least significant bit of the `MASK` register is extracted to the `CARRY` register through an `AND` instruction with value (`0x0001`). The case for SM2 is described in Algorithm 3. In steps 19 and 20, to handle the `0xFFFE` pattern, we perform the `0xFFFF` pattern and then add the carry value.

3.2 Interleaved Montgomery Multiplication/Squaring for a Special Modulus

Generic n-word Montgomery multiplication requires $(n^2 + n)$ multiplication. The Montgomery multiplication consists of multiplication (n^2) and reduction $(n^2 + n)$ parts. Both parts can be implemented in an interleaved or separated way. On the one hand, the advantage of a separated

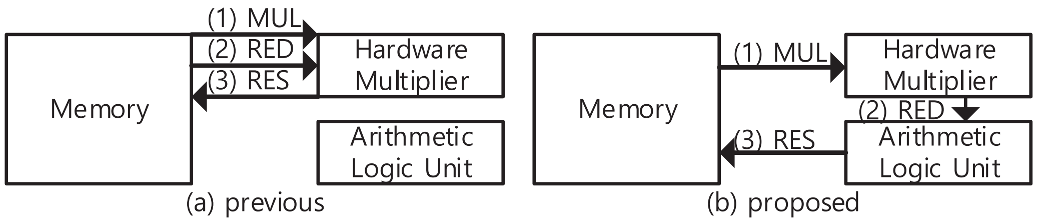

Fig. 1. Procedures for Montgomery multiplication in hardware utilization. MUL, multiplication; RED, reduction; RES, result.

ALGORITHM 3: Masked Subtraction for SM2 on MSP430X

Input: carry register (CARRY), temporal register (MASK)
Output: result pointer (RESULT)

```
...
 1: CLR MASK
 2: SUB CARRY, MASK

 3: SUB MASK, 2*0(RESULT)
 4: SUBC MASK, 2*1(RESULT)
 5: SUBC MASK, 2*2(RESULT)
 6: SUBC MASK, 2*3(RESULT)
 7: SBC 2*4(RESULT)
 8: SBC 2*5(RESULT)
 9: SUBC MASK, 2*6(RESULT)
10: SUBC MASK, 2*7(RESULT)
11: SUBC MASK, 2*8(RESULT)
12: SUBC MASK, 2*9(RESULT)
13: SUBC MASK, 2*10(RESULT)
14: SUBC MASK, 2*11(RESULT)

15: SUBC MASK, 2*12(RESULT)
16: SUBC MASK, 2*13(RESULT)

17: SUBC MASK, 2*14(RESULT)
18: SUBC MASK, 2*15(RESULT)

19: ADD CARRY, 2*14(RESULT)
20: ADC 2*15(RESULT)
...
```

version combines any multiplication and reduction methods without difficulties. The separated mode is usually considered for efficient implementation when the Karatsuba multiplication is used for multiplication. On the other hand, the interleaved version optimizes the number of memory accesses for intermediate results since the reduction is directly performed on intermediate results of multiplication. In this work, the interleaved version is adopted since the hardware multiplier of MSP430X is very efficient to handle the accumulation of intermediate results. For this reason, we decided to use a product-scanning method rather than Karatsuba multiplication. In Figure 1, the procedures for interleaved Montgomery multiplication in hardware utilization are compared.

Note that previous methods only utilized the hardware multiplier for both multiplication and reduction operation. This approach is efficient for the original Montgomery multiplication. However, the proposed method performs the multiplication in the hardware multiplier module and the reduction in the basic ALU. This approach shows better performance than previous works when it comes to the special modulus. The special modulus requires a small number of partial products. Furthermore, partial products can be performed in basic arithmetic.

3.2.1 Register and Memory Utilization. Since performance highly relies on the number of memory accesses, optimized register utilization is very important for high-speed implementations. The MSP430X microprocessor equips only 12 general-purpose registers, among which 5, 2, 1, and 3 registers are assigned for intermediate results, temporal storage, memory address of intermediate results in hardware multiplier, and memory address of operands and results, respectively. Every operand of multiplication is directly assigned to the hardware multiplier, and

96-bit-wise intermediate results are cached in five 16-bit registers, which is used for efficient reduction based on the basic arithmetic. Montgomery multiplication needs to keep Q operands to perform the reduction, which are dynamically loaded/stored from/to the `STACK`. The `STACK` pointer is accessible through the `R1` register.

3.2.2 Modular Reduction. Our modular multiplication combined both hardware-aided multiplication and basic ALU-based modular reduction. At first, we follow the product-scanning multiplication (i.e., column-wise multiplication) routines, which can be implemented with the MAC mode of the hardware multiplier. Afterward, intermediate results are loaded to some 16-bit registers and then reduced. Different from previous Montgomery reduction, which utilizes the product-scanning-based multiplication, in our reduction we exploited properties of the special modulus and thus replaced the expensive multiplication with addition/subtraction operations or no operation.

ALGORITHM 4: Montgomery Multiplication in the Second Column for NIST P-256 on MSP430X

```
Input:  operand pointers (APTR and BPTR), memory
        address of carry bit in hardware multiplier
        (SPTR), temporal registers (T0 and T1)
Output: stack pointer R1, intermediate results
        (C0, C1, C2, C3, CARRY)
...
 1: MOV @APTR+, &MPY32L
 2: MOV @APTR+, &MPY32H
 3: MOV @BPTR+, &OP2L
 4: MOV @BPTR+, &OP2H

 5: MOV @APTR+, &MAC32L
 6: MOV @APTR+, &MAC32H
 7: SUB #2*4, BPTR
 8: MOV @BPTR+, &OP2L
 9: MOV @BPTR+, &OP2H
10: ADD @RL+, C0
11: ADDC @RL+, C1
12: ADDC @RL+, C2
13: ADDC @RL+, C3
14: ADDC @SPTR, CARRY
15: SUB #2*4, RL

16: MOV C0, 2*2(R1)
17: MOV C1, 2*3(R1)
18: MOV C2, C0
19: MOV C3, C1
20: MOV CARRY, C2
21: CLR C3
22: CLR CARRY
...
```

For example, the modulus for the NIST P-256 curve consists of three patterns in a hexadecimal way: `0x00000000`, `0x00000001`, and `0xFFFFFFFF`. Since the `0x00000000` pattern does not require any computations, the routine is optimized away in the reduction routine. The `0x00000001` pattern only requires five 16-bit-wise addition operations for the 64-bit intermediate case, and operands are directly loaded from memory and added to the memory. The `0xFFFFFFFF` pattern requires three 16-bit-wise addition operations and five 16-bit wise subtraction operations, where both operations require identical 32-bit operands. We first load 32-bit operands to two 16-bit registers (temporal storage) and use operands twice for 32-bit addition and 32-bit subtraction, respectively. When the `0xFFFFFFFF` pattern appears before the operand generation, five 16-bit-wise subtraction operations are optimized away because the least significant double word is always set to zero. The `0xFFFFFFFE` pattern for the SM2 curve requires one more subtraction after the `0xFFFFFFFF` pattern operation.

Montgomery reduction can be efficiently exploited and further simplified by taking advantage of the so-called Montgomery-friendly modulus, which admits efficient computations, such as *all-zero* words for the lower part of the modulus. Efficient optimizations for the modulus were first pointed out by Costello et al. [2016] in the setting of SIKE when using a modulus of the form $2^x \cdot 3^y - 1$ (referred to as SIKE-friendly primes) are exploited by the SIKE library [Azarderakhsh et al. 2019]. We further optimized the Montgomery reduction for NIST P-256 (i.e., $M = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$) by using this Montgomery-friendly prime technique. The following are simplified computations for NIST P-256:

$$\begin{aligned} Z &= (T + (T \cdot M' \, mod \, 2^{256}) \cdot M)/2^{256} \\ &= (T + (T \cdot M' \, mod \, 2^{256}) \cdot (2^{256} - 2^{224} + 2^{192} + 2^{96}) - (T \cdot M' \, mod \, 2^{256}))/2^{256} \\ &= (T + (T \cdot M' \, mod \, 2^{256}) \cdot (2^{256} - 2^{224} + 2^{192} + 2^{96}))/2^{256}. \end{aligned}$$

The following are simplified computations for SM2:

$$\begin{aligned} Z &= (T + (T \cdot M' \, mod \, 2^{256}) \cdot M)/2^{256} \\ &= (T + (T \cdot M' \, mod \, 2^{256}) \cdot (2^{256} - 2^{224} - 2^{96} + 2^{64}) - (T \cdot M' \, mod \, 2^{256}))/2^{256} \\ &= (T + (T \cdot M' \, mod \, 2^{256}) \cdot (2^{256} - 2^{224} - 2^{96} + 2^{64}))/2^{256}. \end{aligned}$$

The simplified Montgomery reduction only needs to perform partial products with one `0xFFFFFFFF` and two `0x00000001` words for the NIST P-256 curve. The original Montgomery reduction requires partial products with four `0xFFFFFFFF` and one `0x00000001` words for the NIST P-256 curve. For the SM2 curve, four `0xFFFFFFFF` and one `0xFFFFFFFE` word operations are required. The original Montgomery reduction requires partial products with six `0xFFFFFFFF` and one `0xFFFFFFFE` words for the SM2 curve.

The reduced number of modulus words directly improves performance. This technique is easily applied to other ECC curves, such as NIST P-192. The detailed descriptions of Montgomery multiplication in the second column for NIST P-256 on MSP430X are given in Algorithm 4. It can be seen that from step 1 to step 15, two partial products are obtained in the product-scanning way, whereas from step 16 to step 22, Montgomery reduction with a simplified modulus (`0x00000000`) is optimized away. The multiplication assigned operands to the multiplier in the indirect memory mode, and the multiplication is performed in the MAC mode. After the multiplication, 65-bit results are accumulated to registers (`C0, C1, C2, C3, CARRY`). The reduction part is just result alignment. For better understanding, we used the rhombus form and block form in Figures 2 and 3. Let A and B be operands of length m bits each. Each operand is written as $A = (A[n-1], \ldots, A[1], A[0])$ and $B = (B[n-1], \ldots, B[1], B[0])$, where $n = \lceil m/w \rceil$ is the number of words to represent operands and w is the computer word size (i.e., 32 bits). The result $C = A \cdot B$ is represented as $C = (C[2n-1], \ldots, C[1], C[0])$. In the rhombus form, the lowest index ($i, j = 0$) of the product appears at the rightmost corner, whereas the highest index ($i, j = n-1$) appears at the leftmost corner. A black arrow over a point indicates the processing of a partial product. The lowermost points represent results $C[i]$ from the rightmost corner ($i = 0$) to the leftmost corner ($i = 2n-1$). The black dot describes the partial product. Green, red, and blue dots indicate the `0x00000001`, `0xFFFFFFFF`, and `0xFFFFFFFE` patterns, respectively. In the block form, the multiplication is performed from top right to bottom left. In Figure 2, Montgomery multiplication of NIST P-256 on 16-bit MSP430X microcontrollers is described. The above rhombus form indicates multiplication and the below rhombus form indicates reduction, respectively. From the first to third columns, only the multiplication part is computed. From the fourth column, the `0x00000001` pattern appears and one addition is required for reduction. From the seventh column, two addition operations for reduction are required. From the eighth column, the `0xFFFFFFFF` pattern appears, which requires one addition and one subtraction operation.

In Figure 3, Montgomery multiplication of the SM2 curve on 16-bit MSP430X microcontrollers is described. From the first to second columns, only the multiplication part is computed. From the third column, the `0x00000001` pattern appears and one addition is required for reduction. From

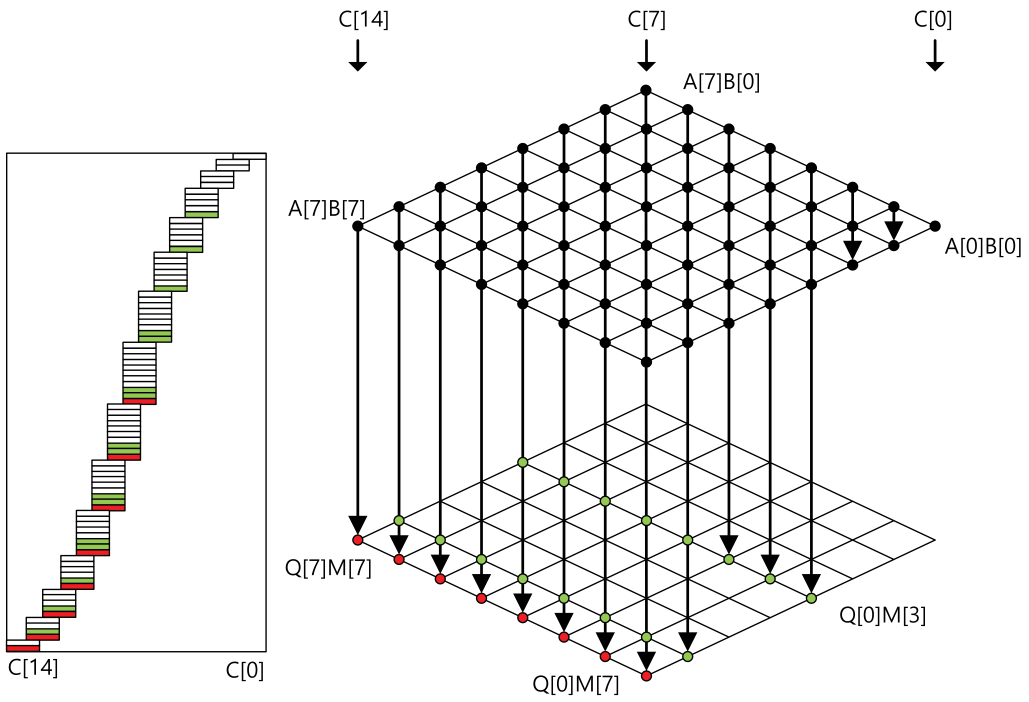

Fig. 2. Montgomery multiplication of NIST P-256 on 16-bit MSP430X microcontrollers

the fifth to the seventh columns, the `0xFFFFFFFF` pattern appears, which requires one addition and subtraction operation. From the eighth column, the `0xFFFFFFFE` pattern appears, which requires one addition and two subtraction operations.

3.2.3 Final Reduction. The last step of Montgomery multiplication may require the final subtraction to get reduced results. We adopted the masked subtraction described in Algorithms 2 and 3 for NIST P-256 and SM2 curves, respectively. Since the final reduction is always performed, the implementation ensures constant timing.

3.2.4 Modular Squaring. The squaring operation is also frequently called in cryptographic implementations. For the straightforward squaring implementation, we can directly use the multiplication implementation for squaring by setting both operands to identical values. However, the multiplication routine does not ensure the highest performance for squaring operation, as some memory accesses/partial products can be optimized by loading/performing once rather than twice. Detailed descriptions are given in Algorithm 5. Note that from steps 1 to 6, the partial product is obtained. When the part of the operand for the partial product is identical, we only need to assign it rather than full operands. This program performs doubling on the partial product.

3.3 Implementation of NIST P-256 and SM2 on Microprocessors

The first implementation of ECC on MSP430X belongs to Gouvêa et al. [2012], where they utilized new 32-bit hardware multiplier instructions of MSP430X. Particularly, the new 32-bit hardware multiplier enhances the previous 16-bit hardware multiplier–based prime field multiplication by about 45%. The combination of optimized algorithms and hardware accelerators shows that ECC at the security level of 128 bits is feasible for MSP430X. Seo and Kim [2014] and Seo et al. [2013,

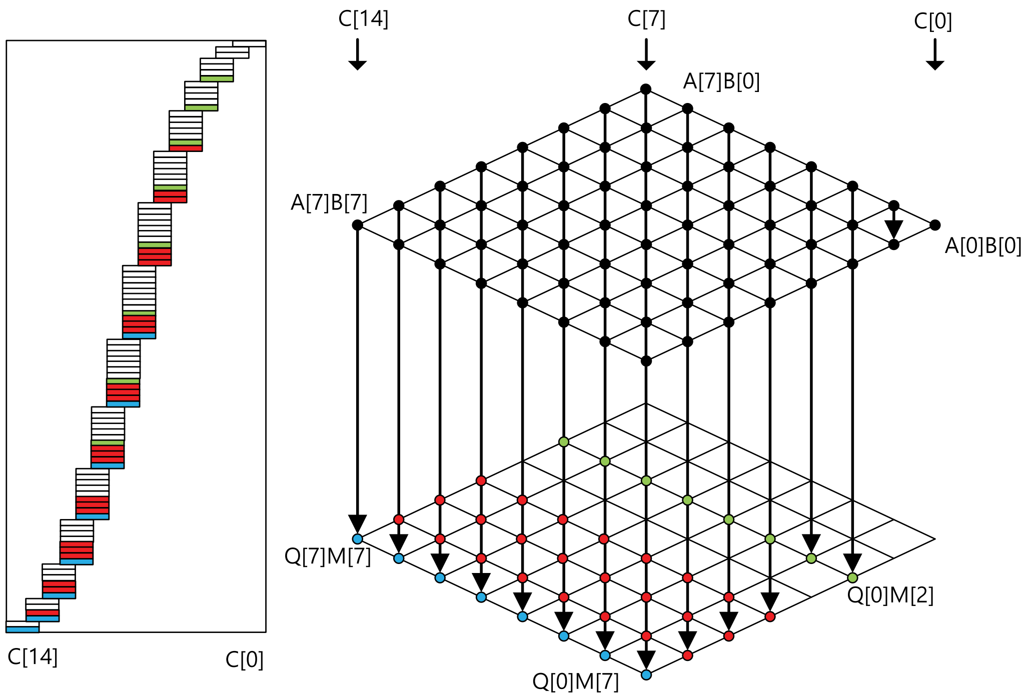

Fig. 3. Montgomery multiplication of SM2 on 16-bit MSP430X microcontrollers.

ALGORITHM 5: Partial Products for Squaring Operations on MSP430X

Input: operand pointers (APTR and BPTR), memory address of carry bit in hardware multiplier (SPTR)
Output: intermediate results (CARRY)

```
...
1: MOV @APTR+, &MAC32L
2: MOV @APTR+, &MAC32H
3: SUB #2*4, BPTR
4: MOV @BPTR+, &OP2L
5: MOV @BPTR+, &OP2H
6: ADD @SPTR, CARRY

7: SUB #2*2, BPTR
8: MOV @BPTR+, &OP2L
9: MOV @BPTR+, &OP2H
10: ADD @SPTR, CARRY
...
```

2014] intensively studied multi-precision multiplication and squaring operations on MSP430 processors, where they optimized register usages by caching operands and memory access through an incremental addressing mode.

In LatinCrypt 2014, Hinterwälder et al. [2014] suggested Curve25519 and NIST P-256 for MSP430 microcontrollers, in which they avoided conditional jumps and loads to prevent timing attacks. Moreover, they provided a comprehensive evaluation of different implementations of the modular multiplication, based on which Curve25519 and NIST P-256 implementations on MSP430X 32-bit hardware multipliers achieved 6.5M and 9.1M cycles, respectively. In particular, they achieved the best performance with one-level Karatsuba multiplication. Düll et al. [2015] optimized the X25519 key exchange protocol for MSP430X 16-bit microcontrollers, and their implementations for MSP430X take 5,301,792 cycles (32-bit multiplier) and 7,933,296 cycles (16-bit multiplier) for the computation of Diffie-Hellman key exchange. The computation is performed in less than a second if clocked at 16 MHz for a security level of 128 bits. Recently, Seo [2018] presented a size-optimized implementation of Curve25519, where he utilized a hardware multiplier and accelerated the performance through the optimized multiplication routines in a product-scanning way. Interestingly, he successfully demonstrated that high performance is possible with a small code size. In a later work of Seo [2019], the optimization of the Curve Ed448 implementation on low-end IoT processors (i.e., 8-bit AVR and 16-bit MSP processors) was presented. In particular, the three-level and two-level subtractive Karatsuba algorithms are adopted for multi-precision multiplication on AVR and MSP processors. The first SM2 implementation on microcontrollers belongs to Zhou et al. [2019], but they do not use Montgomery multiplication and MSP430X microcontrollers to evaluate.

In this work, we targeted the special modulo of NIST P-25 and implemented desired cryptographic primitives. The NIST P-256 elliptic curve is given by

$$E/\mathbb{F}_{p_{256}} : y^2 = x^3 - 3 \cdot x + b,\ p_{256} = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1,$$

and other details can be found in the FIPS 186-2 standard [FIPS 186-2 2000].

The SM2 curve is given by

$$E/\mathbb{F}_{p_{256}} : y^2 = x^3 + a \cdot x + b,\ p_{256} = 2^{256} - 2^{224} - 2^{96} + 2^{64} - 1,$$

and other details can be found in the SM2 standard [Chinese Encryption Administration 2010].

For finite field arithmetic, we mainly follow proposed techniques described in Section 3 to do modular addition/subtraction and modular multiplication/squaring operations. Moreover, we adopted the constant-time finite field inversion of NIST P-256, which is performed by powering $p_{256} - 2$. Such inversion can be computed at a cost of 255 squaring and 13 multiplication by following Algorithm 2 in Zhou et al. [2019]. Similarly, we adopted the Fermat-based inversion for SM2 in Algorithm 3 of Zhou et al. [2019]. For embedded processors, RAM is a very limited resource. To reduce RAM usage, we reused variables for finite field inversion. Only six 32-byte variables are declared and used. For the elliptic curve group arithmetic, we utilized the Montgomery ladder using co-Z Jacobian arithmetic with X and Y coordinates only, which ensures the fast and regular Montgomery ladder algorithm for scalar multiplication [Rivain 2011]. Since the regular Montgomery ladder algorithm does not require conditional statements, the implementation is always constant timing and thus is secure against simple power analysis and timing attacks.

3.4 Double Modular Addition

In point doubling, two operand doubling operations are required. This can be done with ordinary finite field addition, but the dedicated doubling routine can improve performance. Since doubling only requires one operand pointer, we can utilize one more register to store variables. We used nine registers to keep intermediate results, and remaining seven words are stored in the memory directly. Detailed descriptions are given in Algorithm 6. From step 1 to step 17, operands are loaded to registers and memory. The nine 16-bit registers (`Q0-Q8`) store intermediate results, and memory pointed by `RPTR` stores seven 16-bit words. The doubling is performed from step 18 to step 33.

4 EVALUATION

We implemented both NIST P–256 and SM2 curves by using optimized arithmetic operations on 16-bit MSP430X microprocessors (i.e., MSP430F5529) and evaluated the performance of implementations in execution time (clock cycles) and memory (bytes). In Table 2, detailed descriptions of performance evaluation for finite field operations are given. Note that addition and subtraction operations are much cheaper than multiplication and squaring operations (i.e., 8 times faster). The optimized doubling operation is faster than ordinary addition by seven and six clock cycles for NIST P-256 and SM2, respectively. It is also natural that the squaring operation is faster (by 5.2% and 4.3% for NIST P-256 and SM2 curves, respectively) than multiplication through the dedicated

Table 2. Performance Evaluation (Execution Timing in Clock Cycles) of Finite Field Addition, Subtraction, Multiplication, Squaring, and Inversion Operations for NIST P−256 and SM2 on 16-Bit MSP430X Microprocessors

	Timing (Cycles)					
Curve	ADD	D_ADD	SUB	MUL	SQR	INV
NIST P-256	222	215	223	1,899	1,799	487,663
SM2	242	236	243	2,277	2,177	590,234

squaring routine in this article. The inversion is implemented based on Fermat's little theorem, which is a regular fashion and ensures constant timing. Even though NIST P-256 and SM2 curves have the same length as the modulus (256 bits long), the NIST P-256 curve shows better performance than the SM2 curve due to modulus patterns.

ALGORITHM 6: Double Addition for NIST P-256 on MSP430X

Input: operand pointer (APTR))
Output: result pointer (RPTR)

```
...
MOV @APTR+, Q0
MOV @APTR+, Q1
MOV @APTR+, Q2
MOV @APTR+, Q3
MOV @APTR+, Q4
MOV @APTR+, Q5
MOV @APTR+, Q6
MOV @APTR+, Q7
MOV @APTR+, Q8

MOV @APTR+, 2*9(RPTR)
MOV @APTR+, 2*10(RPTR)
MOV @APTR+, 2*11(RPTR)
MOV @APTR+, 2*12(RPTR)
MOV @APTR+, 2*13(RPTR)
MOV @APTR+, 2*14(RPTR)
MOV @APTR+, 2*15(RPTR)
SUB #2*16, APTR

ADD @APTR+, Q0
ADDC @APTR+, Q1
ADDC @APTR+, Q2
ADDC @APTR+, Q3
ADDC @APTR+, Q4
ADDC @APTR+, Q5
ADDC @APTR+, Q6
ADDC @APTR+, Q7
ADDC @APTR+, Q8

ADDC @APTR+, 2*9(RPTR)
ADDC @APTR+, 2*10(RPTR)
ADDC @APTR+, 2*11(RPTR)
ADDC @APTR+, 2*12(RPTR)
ADDC @APTR+, 2*13(RPTR)
ADDC @APTR+, 2*14(RPTR)
ADDC @APTR+, 2*15(RPTR)
...
```

We evaluated the proposed NIST P-256 and SM2 implementations on MSP430X microcontrollers. The scalar multiplication of NIST P-256 requires 8,582,338 clock cycles, and the required memory for Code, Data, and Const are 7,480, 1,700, and 156 bytes, respectively. The scalar multiplication of SM2 requires 10,027,086 clock cycles, and the required memory for Code, Data, and Const are 8,678, 1,704, and 160 bytes, respectively. The implementation of NIST P-256 shows better performance than SM2. The performance difference comes from the difference of finite field arithmetic, which shows that NIST P-256 curve has implementation-friendly parameters than SM2 curve.

We also give the comparison results of NIST P-256 and SM2 from previous work in Table 4. For the most performance-critical operations, our proposed modular multiplication operation improved the performance of those in Hinterwälder et al. [2014] by 23.6%, respectively. Such performance enhancements are achieved through optimized memory access, register utilization, and

Table 3. Performance Evaluation (Execution Timing in Clock Cycles and Memory Usage in Bytes) of Scalar Multiplication for NIST P–256 and SM2 on 16-Bit MSP430X Microprocessors

	Timing (Cycles)	Memory (Bytes)		
Curve	Scalar MUL	Code	Data	Const
NIST P-256	8,582,338	7,480	1,700	156
SM2	10,027,086	8,678	1,704	160

Table 4. Comparison of NIST P-256 and SM2 Implementations on 16-Bit MSP430X Microprocessors

		Timing (Cycles)			Security	
Method	Curve	MUL	SQR	Scalar MUL	Cache Attack	Timing Attack
[Gouvêa et al. 2012]	NIST P-256	3,315	3,064	5,321,776	–	–
[Hinterwälder et al. 2014]	NIST P-256	2,488	–	9,139,739	✔	✔
This work	NIST P-256	1,899	1,799	8,582,338	✔	✔
This work	SM2	2,277	2,177	10,027,086	✔	✔

Note: Timing is measured in terms of clock cycles.

efficient modular reduction techniques. Moreover, this performance improvement directly influences the performance of scalar multiplication.

Previous implementations of scalar multiplication required 5,321,776 clock cycles [Gouvêa et al. 2012], which is faster than ours. However, this is mainly because those implementations utilized the irregular NAF method for scalar multiplication, which requires a pre-computed look-up table (LUT) to accelerate the performance. Yet the frequent LUT access increases cache hit rates and may cause cache attack. It should be noted that in the work of Gouvêa et al. [2012], the point addition and doubling chain is not a regular fashion, which would also be vulnerable to timing attack.

To avoid potential side channel attacks, NIST P-256 and SM2 curves should be implemented in a regular fashion as with the Montgomery ladder algorithm. Hinterwälder et al. [2014] first implemented the Montgomery ladder algorithm for NIST P-256. We also used the Montgomery ladder for scalar multiplication on the NIST P-256 curve. Thus, constant timing finite field arithmetic and regular elliptic curve group arithmetic result in constant timing scalar multiplication implementation. Even though this sacrifices performance, the implementation is much more secure than in previous work by Gouvêa et al. [2012]. Compared with the previous secure NIST P-256 implementation by Hinterwälder et al. [2014], the proposed implementation enhances the performance by 6.1% through the improved modular multiplication. Since the proposed work is, to the best of our knowledge, the first SM2 implementation on MSP430X microcontrollers, we cannot compare results with previous works. The SM2 implementation shows performance similar to that of NIST P-256.

5 CONCLUSION

In this article, we presented new optimal modular multiplication techniques for a special modulus based on the interleaved Montgomery multiplication on 16-bit MSP430X microprocessors. The multiplication part of Montgomery multiplication is performed in the hardware multiplier, whereas the reduction operation is performed in the basic ALU with an optimal routine. The Montgomery reduction is optimized by using the Montgomery-friendly prime technique. Furthermore, the final subtraction is efficiently handled through the masked subtraction for the target embedded processors.

The proposed implementation improved the previous modular multiplication operation for the NIST P-256 curve by 23.6% for 16-bit MSP430X microprocessors. Based on the improved Montgomery multiplication, the scalar multiplication of NIST P-256 and SM2 curves is efficiently constructed. The implementation utilized the Co-Z representation and security countermeasures against timing attack and simple power analysis. The proposed implementation of scalar multiplication of NIST P-256 and SM2 curvesachieves 8,582,338 clock cycles (0.53 seconds@16 MHz) and 10,027,086 clock cycles (0.62 seconds@16 MHz), respectively. It is our hope that such techniques for modular multiplication with a special modulus on a MSP430X microprocessor would improve performance (as well as implementation security) of cryptographic primitives, which are thus applicable for more cryptographic schemes (e.g., NIST ECC and SIKE) and more platforms (e.g., 8-bit AVR and 32-bit ARM Cortex-M).

REFERENCES

Mehmet Adalier. 2015. Efficient and secure elliptic curve cryptography implementation of Curve P-256. In *Proceedings of the Workshop on Elliptic Curve Cryptography Standards*.

Reza Azarderakhsh, Matthew Campagna, Craig Costello, Luca De Feo, Basil Hess, Amir Jalali, David Jao, et al. 2017. Supersingular Isogeny Key Encapsulation—Submission to the NIST's Post-Quantum Cryptography Standardization Process. Retrieved April 19, 2020 from https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-1/submissions/SIKE.zip.

Reza Azarderakhsh, Matthew Campagna, Craig Costello, Luca De Feo, Basil Hess, Amir Jalali, David Jao, et al. 2019. Supersingular Isogeny Key Encapsulation—Submission to the NIST's Post-Quantum Cryptography Standardization Process, Round 2. Retrieved April 19, 2020 from https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-2/submissions/SIKE-Round2.zip.

Craig Costello, Patrick Longa, and Michael Naehrig. 2016. Efficient algorithms for supersingular isogeny Diffie-Hellman. In *Advances in Cryptology—CRYPTO 2016*. Lecture Notes in Computer Science, Vol. 9814. Springer, 572–601.

Michael Düll, Björn Haase, Gesine Hinterwälder, Michael Hutter, Christof Paar, Ana Helena Sánchez, and Peter Schwabe. 2015. High-speed Curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers. *Designs, Codes and Cryptography* 77, 2–3 (2015), 493–514.

FIPS 186-2. 2000. Digital Signature Standard (DSS). *Federal Information Processing Standards Publication 186-2. National Institute of Standards and Technology.*

Armando Faz-Hernández, Julio López, Eduardo Ochoa-Jiménez, and Francisco Rodríguez-Henríquez. 2018. A faster software implementation of the supersingular isogeny Diffie-Hellman key exchange protocol. *IEEE Transactions on Computers* 67, 11 (2018), 1622–1636.

Conrado P. L. Gouvêa, Leonardo B. Oliveira, and Julio López. 2012. Efficient software implementation of public-key cryptography on sensor networks using the MSP430X microcontroller. *Journal of Cryptographic Engineering* 2, 1 (2012), 19–29.

Shay Gueron and Vlad Krasnov. 2015. Fast prime field elliptic-curve cryptography with 256-bit primes. *Journal of Cryptographic Engineering* 5, 2 (2015), 141–151.

Gesine Hinterwälder, Amir Moradi, Michael Hutter, Peter Schwabe, and Christof Paar. 2014. Full-size high-security ECC implementation on MSP430 microcontrollers. In *Proceedings of the International Conference on Cryptology and Information Security in Latin America*. 31–47.

Amir Jalali, Reza Azarderakhsh, Mehran Mozaffari Kermani, and David Jao. 2019. Supersingular isogeny Diffie-Hellman key exchange on 64-bit ARM. *IEEE Transactions on Dependable and Secure Computing* 15, 5 (2019), 902–912.

Brian Koziel, Amir Jalali, Reza Azarderakhsh, David Jao, and Mehran Mozaffari-Kermani. 2016. NEON-SIDH: Efficient implementation of supersingular isogeny Diffie-Hellman key exchange protocol on ARM. In *Proceedings of the International Conference on Cryptology and Network Security*. 88–103.

Zhe Liu, Hwajeong Seo, Aniello Castiglione, Kim-Kwang Raymond Choo, and Howon Kim. 2019. Memory-efficient implementation of elliptic curve cryptography for the Internet-of-Things. *IEEE Transactions on Dependable and Secure Computing* 16, 3 (2019), 521–529.

Zhe Liu, Hwajeong Seo, Johann Großschädl, and Howon Kim. 2016. Efficient implementation of NIST-compliant elliptic curve cryptography for 8-bit AVR-based sensor nodes. *IEEE Transactions on Information Forensics and Security* 11, 7 (2016), 1385–1397.

Daniel Peters, Dejan Raskovic, and Denise Thorsen. 2009. An energy efficient parallel embedded system for small satellite applications. *ISAST Transactions on Computers and Intelligent Systems* 1, 2 (2009), 8–16.

Matthieu Rivain. 2011. Fast and regular algorithms for scalar multiplication over elliptic curves. *IACR Cryptology ePrint Archive* 338.

Hwajeong Seo. 2018. Compact software implementation of public-key cryptography on MSP430X. *ACM Transactions on Embedded Computing Systems* 17, 3 (2018), 66.

Hwajeong Seo. 2019. Compact implementations of curve Ed448 on low-end IoT platforms. *ETRI Journal* 41, 6 (2019), 863–872.

Hwajeong Seo. 2020. Memory efficient implementation of modular multiplication for 32-bit ARM Cortex-M4. *Applied Sciences* 10, 4 (2020), 1539.

Hwajeong Seo, Amir Jalali, and Reza Azarderakhsh. 2019a. *Optimized SIKE Round 2 on 64-Bit ARM*. Technical Report. *IACR Cryptology ePrint Archive*. 721.

Hwajeong Seo, Amir Jalali, and Reza Azarderakhsh. 2019b. SIKE round 2 speed record on ARM Cortex-M4. In *Proceedings of the International Conference on Cryptology and Network Security*. 39–60.

Hwajeong Seo and Howon Kim. 2014. Multi-precision squaring on MSP and ARM processors. In *Proceedings of the 2014 International Conference on Information and Communication Technology Convergence (ICTC'14)*. IEEE, Los Alamitos, CA, 356–361.

Hwajeong Seo, Yeoncheol Lee, Hyunjin Kim, Taehwan Park, and Howon Kim. 2014. Binary and prime field multiplication for public key cryptography on embedded microprocessors. *Security and Communication Networks* 7, 4 (2014), 774–787.

Hwajeong Seo, Kyung-Ah Shim, and Howon Kim. 2013. Performance enhancement of TinyECC based on multiplication optimizations. *Security and Communication Networks* 6, 2 (2013), 151–160.

Sean Shen and Xiaodong Lee. 2014. *SM2 Digital Signature Algorithm*. Retrieved on April 30, 2020 from https://tools.ietf.org/html/draft-shen-sm2-ecdsa-02.

Colin D. Walter and Susan Thompson. 2001. Distinguishing exponent digits by observing modular subtractions. In *Proceedings of the Cryptographers' Track at the RSA Conference*. 192–207.

Lu Zhou, Chunhua Su, Zhi Hu, Sokjoon Lee, and Hwajeong Seo. 2019. Lightweight implementations of NIST P-256 and SM2 ECC on 8-bit resource-constraint embedded device. *ACM Transactions on Embedded Computing Systems* 18, 3 (2019), Article 23.

Received July 2019; revised December 2019; accepted March 2020