Memory-Efficient Implementation of Elliptic Curve Cryptography for the Internet-of-Things

Zhe Liu, *Senior Member, IEEE*, Hwajeong Seo, Aniello Castiglione, *Senior Member, IEEE*, Kim-Kwang Raymond Choo, *Senior Member, IEEE*, and Howon Kim, *Member, IEEE*

Abstract—In this paper, we present memory-efficient and scalable implementations of NIST standardized elliptic curves P-256, P-384 and P-521 on three ARMv6-M processors (i.e. Cortex-M0, M0+, and M1). Specifically, we propose a refined approach to perform the Multiply-ACcumulate (MAC) operation using hardware multiplier provided by ARMv6-M processor, and a compact doubling routine for multi-precision squaring that executes both doubling and partial product operations in an efficient way. We demonstrate that the proposed squaring implementation achieves a speed up of 28 percent compared to the same operation employed in Micro-ECC. Then, we reduce one modular reduction in co-Z conjugate point addition by using lazy reduction and special form representation (CD-AB, EF-AB), which further reduces the execution time of both P-256 and P-384 implementations. Finally, we propose scalable implementations of ECC scalar multiplication on ARMv6-M processors that are widely used for Internet of Things applications.

Index Terms—Elliptic curve cryptography, multi-precision arithmetic, lazy reduction, ARMv6–M architecture, Internet of Things

1 Introduction

A large body of research has been devoted to improving the performance of cryptographic algorithms on embedded devices, and this research topic has received increasing attention in recent years. A seminal work belongs to Gura et al. [8], who proposed the now-classic hybrid multiplication and optimized RSA and Elliptic Curve Cryptography (ECC) on 8-bit AVR and CC1010 microcontrollers. Since then, researchers have focused on further reducing the execution time of software and hardware ECC implementation, including TinyECC [12], NaCl [10], and MoTE-ECC [16] on 8-bit AVR, the work presented in [15] on MSP430 processor as well as the hardware implementation proposed in [14].

The low-cost 32-bit ARMv6-M processors (Cortex-M0, M0+, and M1) are popular processors released by the ARM company.[1] Compared to old series, the ARMv6-M architecture has some application-friendly features, including low cost, increased connectivity, better code reuse, smaller silicon area, and improved energy efficiency. These features allow developers to achieve 32-bit performance at an 8-bit price point for applications, such as in smart grids and smart homes [2]. According to the comparative summary presented by De Clercq et al., cryptography implementations on 32-bit ARMv6-M processors could achieve better performance with a reduced energy requirement than on 8-bit AVR processors [5, Table 4].

Nevertheless, there exist few implementations of cryptography algorithm on ARMv6-M in the published literature. Wenger and Unterluggauer were one of the first to implement ECC over Cortex-M0+ in 2013. They proposed the basic MAC operation for fixed length multiplication [26]. Later in CHES'14, they revisited the previous work and improved the performance by introducing advanced MAC hardware extension and drop-in module. Their findings demonstrated the potential for designing small and fast hardware implementations of pairing-based cryptography [25]. De Clercq et al. optimized López-Dahab (LD) [17] polynomial multiplication on Cortex-M0+ processor and implemented ECC over binary fields. Their implementation outperformed previous software implementations by a factor of at least 3.3 in execution time [5]. In 2015, Düll et al. resulted in new performance/speed records for 128-bit secure elliptic-curve Diffie-Hellman key-exchange software on Cortex-M0. Their implementation only required 3.6×10^6 clock cycles and outperformed Wenger and Unterluggauer's work by a factor of 3 [6]. In

- *Z. Liu is with the College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 210016, China and also with the Interdisciplinary Centre for Security, Reliability and Trust (SNT), University of Luxembourg, Esch-sur-Alzette 4365, Luxembourg. E-mail: sduliuzhe@gmail.com.*
- *H. Seo is with the Collage of IT Engineering, Hansung University, Seoul 136-792, Republic of Korea. E-mail: hwajeong84@gmail.com.*
- *A. Castiglione is with University of Salerno, Fisciano 84084, Italy. E-mail: castiglione@ieee.org.*
- *K.-K. R. Choo is with the Department of Information Systems and Cyber Security and Department of Electrical and Computer Engineering, University of Texas at San Antonio, San Antonio, TX 78249. E-mail: raymond.choo@fulbrightmail.org.*
- *H. Kim is with the College of Computer Engineering, Pusan National University, Busan 609-735, Republic of Korea. E-mail: howonkim@pusan.ac.kr.*

Manuscript received 30 June 2017; revised 12 Feb. 2018; accepted 25 Mar. 2018. Date of publication 11 Apr. 2018; date of current version 10 May 2019. (Corresponding author: Z. Liu.)
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TDSC.2018.2825449

1. https://armkeil.blob.core.windows.net/developer/Files/pdf/Arm-Cortex-processors_Public_August-2017.pdf, last accessed February 5th, 2018.

CHES'16, Renes et al. described the first implementation of hyper elliptic cryptography on ARM Cortex-M0, and the key-exchange scalar multiplication runs in under 2.6×10^6 clock cycles [20].

Previous software implementations on ARMv6-M architecture mainly focused on improving the execution time, while the code size and the scalability of the implementations received less attention. The general ARMv6-M boards (NXP LPC800, Silicon Labs EFM32, STMicroelectronics STM32 L0) only support small flash memory, ranging from 4, 8, 16, 32 to 64 KB. In the flash memory, a large portion will be allocated for the operating system. In general, the operating system does not support any security services other than a simple link-layer encryption. Therefore, application developers have to include auxiliary libraries for security protocols (e.g., Datagram TLS) and the underpinning cryptographic primitives. Furthermore, IoT devices are communicated through a number of heterogeneous network protocols, which support different cryptography suites. Thus, the increasing popularity of IoT devices, including in military and adversarial settings (e.g., Internet of Battlefield Things, and Internet of Military Things) and existing challenges necessitate the design of memory-efficient and scalable implementations of cryptography suites on IoT devices.

Recently in 2017, the Micro-ECC project successfully evaluated memory-efficient implementations of NIST curves [18]. An interesting perspective from this project is to point out that fast execution time with small code size can be achieved in a better trade-off way in practice. The current implementation of Micro-ECC only supports one NIST curve (i.e., P-256) for the TLS 1.3 protocol. However, some interesting speed and size optimization points, including squaring operation, ECC-friendly Montgomery reduction and lazy reduction, have not been duely investigated and the current version of Micro-ECC has non constant-timing, which is vulnerable to some basic side-channel attacks (e.g., timing attacks, simple power analysis attacks).

In this paper, we present a compact and scalable implementation of NIST P-256, P-384 and P-521 on ARMv6-M microprocessors. Specifically, the contributions of this paper are threefold. First, we propose a refined MAC operation that is used in multi-precision multiplication operation. The proposed MAC operation saves one `MOV` operation while reducing the code size. Second, we propose 2-way carry-catcher method to speed up the multi-precision squaring operation. Both of the multi-precision multiplication and squaring are faster than the best previous work. The third contribution is to propose a generic ECC-friendly Montgomery method to optimize the code size of scalar multiplication of NIST P-256, P-384 and P-521. Our proposed implementation of NIST P-256 scalar multiplication on ARMv6-M processors requires an execution time of 16.9K clock cycles while consuming 2.9K bytes for memory footprint, which provides a better trade-off between execution time and memory consumption.

2 ARMv6-M Processors

In this section, we will briefly describe the ARMv6-M processors. The 32-bit ARMv6-M embedded microprocessor (Cortex-M0, M0+, and M1) is the smallest and most energy-efficient processor built by ARM. It is of little surprise that the major organizations have also started to supply these microprocessors. Of the sixteen 32-bit registers, only 13 general purpose registers are directly utilized by the processor. ARMv6-M supports 56 instructions, but only the lower 8 registers (`R0-R7`) are accessible, with the exception of a few operations. The Cortex–M0 and M0+ also support fast 1-cycle ($32 \times 32 \rightarrow 32$-bit) multiplier. The processor only supports Thumb instruction set, which ensures small code size without impacting on ARM performance. Different from ARM instructions, Thumb instruction set has a number of different features, we summarize these as follows [1]:

(1) Branch instructions: program-relative branches are more limited in range than ARM, and unconditional branch is only available in subroutines.
(2) Data processing instructions: the result of the operation must be placed in one of the operand registers. The data processing instructions with limited access to registers (`R8-R15`) cannot update the flags.
(3) Register load and store instructions: in Thumb instruction set, only these registers (`R0-R7`) are accessible.

3 Implementation of Multi-Precision Arithmetic

3.1 Refined MAC

The multiple precision multiplication is one of the most performance-critical operations of public key cryptography in terms of "computational complexity". During the multiple precision multiplication over Cortex-M0+, the word is usually set to 32-bit long and a 16-bit wise multiplication is executed due to the limitation of embedded multiplier ($32 \times 32 \rightarrow 32$-bit).

Previous implementations of multi-precision multiplication on ARMv6-M [6], [18], [26] employed either product-scanning method or subtractive Karatsuba algorithm. For example, Wenger and Unterluggauer [26] introduced the product-scanning with MAC operation. This is also the first attempt to implement 32-bit wise MAC operation on Cortex-M0+. Later, the implementor of Micro-ECC improved Wenger and Unterluggauer's MAC operation by re-ordering the accumulation step, which results in a savings of two addition operations [18]. In another separate and related work, Düll et al. [6] used an optimized subtractive Karatsuba multiplication to achieve even better performance. Specifically, the combination of subtractive Karatsuba algorithm and manual optimization outperforms previous work by a factor of 3. In this paper, we follow a looped fashion of the MAC routine for generic multiplication and squaring implementations, which ensures higher scalability and smaller code size than the unrolled approach.

Given two 32-bit digits a and b, we represent a as $a_H \| a_L$ and b as $b_H \| b_L$, here, a_H and b_H represent the higher 16 bits of a and b while a_L and b_L represent the lower 16 bits of a and b. We use r to represent the intermediate result.

Recall that Wenger and Unterluggauer's MAC operation [26] performs the 32-bit partial product as follows:

$$r \leftarrow ((r + a_L \times b_L + (a_H \times b_H) \ll 32) + (a_L \times b_H) \ll 16) + (a_H \times b_L) \ll 16.$$

As shown in above equation, the MAC step accumulates both lower and higher partial products $a_L \times b_L$ and $a_H \times b_H$ to intermediate results r. Then, both middle partial products $a_L \times b_H$ and $a_H \times b_L$ are sequentially added to the intermediate results r.

Recall that the MAC routine of Micro-ECC is performed as follows [18].

$$r \leftarrow r + (a_L \times b_L + (a_H \times b_H) \ll 32 + ((a_L \times b_H) \ll 16 + (a_H \times b_L) \ll 16)).$$

As shown in above equation, the MAC routine first calculates both the lower and higher partial products $a_L \times b_L$ and $a_H \times b_H$, respectively. Then, the remaining partial products $a_L \times b_H$ and $a_H \times b_L$ are added together and shifted at once. Thereafter, the result is added to both lower and higher partial products. Finally, the sum of all the partial products are added to the intermediate results r. Compared to the MAC operation performed in [26], the reversed accumulation in Micro-ECC saves two addition instructions.

In this paper, we further optimize the MAC of Micro-ECC at the instruction set level. The detailed descriptions are given in Algorithm 1. The main part of MAC routine is identical to that of Micro-ECC but our approach does not initialize the R7 register (see step 22), as only the higher 16-bit value of R7 register is utilized and the lower 16-bit value is always set to 0 in step 18 (0xnnnn0000 where 'n' is unknown values). After step 25, the register (R5) keeps the carry bit in the lower 16-bit part and unknown value in the higher 16-bit (i.e., half word trick). In order to extract the correct carry bit (lower 16-bit) from the register (R5), we use the optimized shifting technique for the intermediate result by 32-bit. Different from the approach adopted in Micro-ECC, we move the higher register to the lower register as follows:

$$\texttt{MOV R3, R4} \rightarrow \texttt{MOV R4, R5} \rightarrow \texttt{MOV R5, \#0}.$$

In our approach, we employ UXTH instruction to extract the correct intermediate results.

$$\texttt{MOV R3, R4} \rightarrow \texttt{UXTH R4, R5} \rightarrow \texttt{MOV R5, \#0}.$$

The UXTH R4, R5 instruction only moves the lower 16-bit part of R5 to the register R4, which efficiently removes the unknown value (higher 16-bit) from register R5. The approach does not address the over-flow condition during general cryptography computations, since the overflow only happens when the length of integer is over 2,097,152-bit (i.e., $2^{16} \times 32$-bit).

Table 1 lists a comparison between our optimized method and previous MAC implementations in [18], [26]. As shown in the table, our proposed MAC operation optimizes both the code size (2 bytes) and execution time (1 clock cycle), which achieves a good trade-off between speed and memory consumption. Note that we have also tried to explore the 64-bit wise MAC operation or inner Karatsuba routine in our implementation. However, both options are not able to achieve better trade-off between execution time and memory consumption. On one hand, 64-bit wise MAC cannot be implemented without using a large number of memory access operations. The 64-bit wise MAC approach requires two times general purpose registers of proposed approach to keep the operands and intermediate results in the general-purpose registers, however, the number of general purpose registers on ARMv6-M platform is very limited. On the other hand, Karatsuba method can improve the performance but the code size is much larger than the proposed method. Based on the above reasons, we decide to adopt the 32-bit wise MAC routine in our implementation to ensure a better trade-off between compact code size and the relatively fast execution time.

TABLE 1
C.S.: Code Size (Bytes), E.T.: Execution Time (Clock Cycles)

Implementation	C.S.	E.T.	ADD	MUL	LSL	UXTH	MOV	LDR
Wenger et al. [26]	54	29	9	4	6	2	4	2
Micro–ECC [18]	50	27	7	4	6	2	4	2
This work	48	26	7	4	6	2	3	2

ADD: addition, subtraction, exclusive-or; MUL: multiplication; LSL: shifting to left, right; UXTH: extracting lower 16-bit; MOV: moving variable, not variable; LDR: loading, storing.

Algorithm 1. MAC Operation for Fast 1-Cycle Hardware Multiplier

Input: Operand stored in pointers R8, R9
Output: Result stored in {R3, R4, R5}

```
 1: MOV R1, R8
 2: LDR R1, [R1, #offset1]
 3: MOV R2, R9
 4: LDR R2, [R2, #offset2]

 5: UXTH R6, R1
 6: UXTH R7, R2
 7: LSR R1, R1, #16
 8: LSR R2, R2, #16

 9: MOV R0, R6
10: MUL R0, R0, R7                    {Low-Low}
11: MUL R6, R6, R2                    {Low-High}
12: MUL R2, R2, R1                    {High-High}
13: MUL R1, R1, R7                    {High-Low}

14: LSL R7, R6, #16                   {Low-High}
15: LSR R6, R6, #16
16: ADD R0, R7
17: ADC R2, R6

18: LSL R7, R1, #16                   {High-Low}
19: LSR R6, R1, #16
20: ADD R0, R7
21: ADC R2, R6

22: //MOV R7, #0                  {Optimized-Away}
23: ADD R3, R3, R0                        {Low-Low}
24: ADC R4, R4, R2                       {High-High}
25: ADC R5, R5, R7                {Half Word Trick}
```

3.2 Multi-precision Multiplication and Squaring

Our implementation of multi-precision multiplication follows the structure of product-scanning for the outer loop algorithm, but the intermediate results are efficiently handled with the proposed MAC operation in each iteration of the inner loop. On the other hand, the implementation of

multi-precision squaring can be seen as an optimized version of Sliding-Block-Doubling (SBD) [23].

The original SBD method for squaring an n-word operand (i.e., represented by $A[0...n-1]$) is performed in two steps. In the first step, the partial products ($A[i] \times A[j]$, where $i \neq j$) are computed in the product-scanning method. Afterwards, the intermediate results are doubled and the remaining partial products ($A[i] \times A[j]$ where $i = j$) are accumulated. A straightforward implementation of SBD method on ARMv6-M platform requires to perform both doubling on intermediate results and addition of remaining partial products operations. These operations require to use many registers simultaneously to hold the local variables, which is not available for ARMv6-M platform. Based on the observation, we propose a 2-way carry-catcher method to perform the doubling and partial product operations. The core idea is to define the two carry-catcher registers to store the two carry bits for doubling and partial product, respectively. The carry-catcher registers are always set to either 1 or 0, so we can use these registers in two different purposes, one for carry-catcher and the other for zero variable. In order to hide the overheads of carry-propagation operations, the carry bit in carry-catcher register is always added to the intermediate results together with other accumulation operations.

The 2-way carry-catcher method is shown in Algorithm 2. First, the intermediate results are loaded from memory address in R0 and stored in registers R5 and R6. As the destination address in R0 is set before entering loop and step 23, the destination address is always ready in the register R0. From step 3 to 6, the intermediate results in register R5 and R6 are doubled with storing the carry into carry-catcher register R4. The carry flag is restored from the carry-catcher register R4 by using logical shift to right LSR before the doubling step. After the doubling step, new carry flag is updated into carry-catcher register R4. From step 7 to 13, the squaring operation with operand R2 is performed and three intermediate results, including R1 for {LOW-LOW}, R2 for {LOW-HIGH} and R0 for {HIGH-HIGH} are obtained. From step 14 to 22, the second carry-catcher register R7 sets the carry flag and the squaring results R1, R2, R0 are added to intermediate results registers R5 and R6 in an optimized manner. Thereafter, the intermediate results are stored into memory R0. Finally, the destination address R0 is compared with the loop counter R9 and the branch instruction is performed by referring to the conditions.

3.3 Generic Montgomery Reduction

For the general modular reduction operation, we adopt the classic Montgomery reduction algorithm, which performs modular reduction without costly division operations [19]. Montgomery reduction algorithm for an n-word operand requires n multiplications for quotient computation ($Q = T \cdot M' \bmod 2^m$), where T is the intermediate result, M is the modulus and M' is the constant ($M' = -M^{-1} \bmod 2^m$), and n^2 times of multiplications for $Q \cdot M$ computation. We implement the product-scanning variant of Montgomery algorithm [24]. The 32-bit wise quotient computation is performed with single MUL instruction ($32 \times 32 \rightarrow 32$-bit) in 1-cycle hardware multiplier, thus only n clock cycles are needed for the quotient computation.

Algorithm 2. 2-way Carry-Catcher for Doubling and Partial Product

Input: Result pointer R8, operand pointer R3, loop counter R9, carry-catcher {R4, R7}
Output: Results {R5, R6}

```
 1: LOOP:
 2: LDM R0!, {R5-R6}

 3: LSR R4, #1                          {Carry-Catcher #1}
 4: ADC R5, R5                                  {Doubling}
 5: ADC R6, R6
 6: ADC R4, R4

 7: LDM R3!, {R2}
 8: UXTH R1, R2
 9: LSR R0, R2, #16

10: MOV R2, R1
11: MUL R1, R1, R1                              {Low-Low}
12: MUL R2, R2, R0                             {Low-High}
13: MUL R0, R0, R0                            {High-High}
14: LSR R7, #1                          {Carry-Catcher #2}
15: ADC R5, R5, R1                              {Low-Low}
16: ADC R0, R0, R7

17: LSL R1, R2, #17                            {Low-High}
18: LSR R2, R2, #15
19: ADD R5, R5, R1
20: ADC R0, R0, R2

21: ADD R6, R6, R0                            {High-High}
22: ADC R7, R7, R7

23: MOV R0, R8
24: STM R0!, {R5-R6}
25: MOV R8, R0

26: CMP R9, R0
27: BHS LOOP
```

The detailed descriptions are given in Algorithm 3. From step 2 to 8, the MUL instruction and MAC operation are performed in order to generate the quotient variable and to compute the intermediate results. Since the least significant word of accumulation result (AC) is always zero, only shifting operation is required without storing the results in step 8. From step 9 to 17, the remaining partial products are computed in product-scanning with MAC operation, and the results are stored into memory (step 13, 16 and 17). Finally, the final subtraction of the modulus (M) is performed to get a fully reduced result in the $[0, M)$ range. We follow the constant-timing masked final subtraction method proposed in [13]. This approach uses the most significant bit (CY) to mask the modulus and to perform the subtraction. After the final subtraction, the final result is placed in the range of $[0, 2^m)$ [27]. We also integrate the masking and subtraction routines, which saves some memory access instructions compared to performing the masking and subtraction in a separate way.

4 Scalable ECC Implementation

4.1 NIST Curves: P-256, P-384, and P-521

NIST curves (P-256, P-384 and P-521) have curve forms of Weierstrass curve, and provide medium and high security

levels ranging from 128-bit to 256-bit [21]. The Weierstrass equation can be expressed as

$$\mathbb{E} : y^2 = x^3 + ax + b$$

with curve parameter $a = -3$, which is defined over the fields $\mathbb{F}_{2^{256}-2^{224}+2^{192}+2^{96}-1}$, $\mathbb{F}_{2^{384}-2^{128}-2^{96}+2^{32}-1}$ and $\mathbb{F}_{2^{521}-1}$ for P-256, P-384, and P-521, respectively. Both NIST primes P-256 and P-384 have a form of Solinas primes, the coefficients of which are on powers of 32-bit radix, thus can be efficiently implemented on the 32-bit architecture.

Algorithm 3. Generic Montgomery Algorithm with MAC

Input: Modulus M, constant M_I= $(-M^{-1} \bmod R) \bmod 2^{32}$ where R is Montgomery radix, 96-bit accumulated results AC, intermediate result P=A×B where A, B are operands, length LEN, carry CY, borrow BW
Output: Result OT

1: AC← 0 {Initialization}
2: **for** i=0 to LEN-1 by 1 **do**
3: **for** j=0 to i-1 by 1 **do**
4: AC← AC+OT[j]×M[i-j] {MAC}
5: **end for**
6: AC← AC+P[i] {Accumulation}
7: OT[i]← (AC× M_I) $\bmod 2^{32}$ {MUL}
8: AC← AC+OT[i]×M[0] {MAC}
9: AC← AC≫ 32 {Shifting}
10: **end for**
11: **for** i=LEN to 2×LEN-2 by 1 **do**
12: **for** j=i-LEN+1 to LEN-1 by 1 **do**
13: AC← AC+OT[j]×M[i-j] {MAC}
14: **end for**
15: AC← AC+P[i] {Accumulation}
16: OT[i-LEN]← AC $\bmod 2^{32}$ {Saving}
17: AC← AC≫ 2^{32} {Shifting}
18: **end for**
19: AC← AC+P[2× LEN-1] {Accumulation}
20: OT[len-1]← AC $\bmod 2^{32}$ {Saving}
21: OT[len]← (AC div 2^{32}) $\bmod 2^{32}$
22: CY← {(AC div2^{64})$\oplus 2^{33}$ -1}+1
23: BW← 0 {Masked Final Subtraction}
24: **for** i=0 to LEN-1 by 1 **do**
25: BW‖ OT[i]← OT[i]-(M[i]&CY)-BW
26: **end for**
27: **return** OT

4.1.1 Modular Subtraction and Addition

After having a close look at the primes of NIST P-256, P-384 and P-521 curves, we find that these three primes only have four 32-bit word forms when represented into hex form, namely, 0xFFFFFFFF, 0xFFFFFFFE, 0x00000001, and 0x00000000. The masked common modular subtraction is shown in Algorithm 4. First, we perform the generic subtraction, which generates intermediate results (AC) and mask variable (K). The mask variable is always set to either 0 or 0xFFFFFFFF. In step 4 and 5, we only load the 5 words of modulus (M[1], M[3], M[4], M[6], M[7]) and mask them with mask variable (K). In step 6, the 256-bit common modulus (MK) is constructed with both masked modulus and mask variables. For the first word of modulus (0xFFFFFFFF), we directly use the mask variable (K), since it keeps the value (0xFFFFFFFF) when the modular reduction is needed. Otherwise, it keeps the value (0), which does not affect the results. From Step 8 to 9, the addition with the 256-bit common modulus is performed, which outputs the reduced results for the P-256 case. For P-384 and P-521 cases, we need to perform word-wise addition operations four and six times more in step 10 and 11. In particular, the remaining modulus is only a single form (0xFFFFFFFF), which needs word-wise addition with the mask variable (K).

Algorithm 4. Masked Common Modular Subtraction

Input: Operands (A, B), modulus M, length LEN (P256: 8, P384: 12)
Output: Modular subtraction (A-B mod M)

1: K← 0, Q←0
2: **for** i=0 to LEN-1 by 1 **do**
3: K‖ AC[i]← A[i]-B[i]-(K&1)
4: **end for**
5: M1← M[1]&K, M3←M[3]&K, M4←M[4]&K
6: M6← M[6]&K, M7←M[7]&K
7: MK← {M7,M6,M3,M4,M3,M1,M1,K}
8: **for** i=0 to LEN-1 by 1 **do**
9: **if** i < 8 **then**
10: Q‖ AC[i]← AC[i]+MK[i]+Q {P256, P384}
11: **else**
12: Q‖ AC[i]← AC[i]+K+Q {P384}
13: **end if**
14: **end for**
15: **return** AC

For the modular addition, the addition results are reduced by the modulus when the carry bit is set. To avoid having a branch statement caused by the carry bit, the modular reduction is performed in a masked way, which is performed in reverse of the masked common modular subtraction. These three different curves P-256, P-384 and P-521 shares modular subtraction and addition routine, this can help minimize the code size, while achieving high performance.

4.1.2 ECC Friendly Montgomery Algorithm

The implementation of reduction in Micro-ECC is based on a number of generic additions, generic subtractions, and conditional statements. We find the work is non constant-timing and the code-size and execution time can be improved. On the other hand, the fast reduction of P-256 curve is not compatible with the P-384 curve, which somehow increases the code size for implementation of the P-384 curve.

In this work, we follow the ECC-friendly Montgomery reduction suggested by Gueron and Krasnov [7]. Gueron and Krasnov showed that the multiplication of word form (0xFFFFFFFF) is in one addition operation and one subtraction operation. Our generic ECC-friendly Montgomery reduction supports both P-256 and P-384 curves,[2] resulting

2. NIST P-192 is also supported.

in an optimization of the code size by about 50 percent than employing two independent routines. The detailed descriptions of generic ECC-friendly Montgomery reduction are given in Algorithm 5. In step 3 and 4, the partial products of modulus (M[i-j]) and the quotient (OT[j]) are performed. Partial products are efficiently handled with addition and subtraction operations as described in Algorithm 6. To further minimize the number of branch statements, we reorganize the computation of partial product according to the occurrence of the word-wise modulus form. For P-256 and P-384 curves, the word-wise modulus form (0xFFFFFFFF) appears four and nine times in the modulus, respectively. For this reason, in step 1, we first check the value whether 0xFFFFFFFF. Subsequently, the other word forms (0x00000000, 0x00000001, and 0xFFFFFFFE) are ordered from the most to the least occurrences. In step 6 of Algorithm 5, the quotient computation is optimized for the special constant M_T (0x00000001). Afterwards, the partial product with least significant word of modulus (M[0], 0xFFFFFFFF) and quotient is performed. It usually requires one addition and one subtraction but we only perform one addition operation since the least significant word of intermediate result is equal to the quotient. Thus, one subtraction is always equal to the zero value.

$$(\mathtt{AC} + (\mathtt{AC} \bmod 2^{32}) \ll 2^{32}) \gg 32$$
$$\leftarrow (\mathtt{AC} + ((\mathtt{AC} \times \mathtt{M\_I}) \bmod 2^{32}) \times \mathtt{M}[0]) \gg 32.$$

In step $9 \sim 14$, the remaining partial products are performed. In particular, the most significant word of modulus (M[7] for P-256 and M[11] for P-384) is always set to 0xFFFFFFFF; thus, we directly perform the addition and subtraction routine for this special case. Otherwise, we use the Algorithm 6. In step $15 \sim 25$, the intermediate results are accumulated and reduced. For the scalar multiplication, we used a fast common-Z interleaved point addition and doubling formulas in Montgomery ladder algorithm [9], [22]. In particular, we optimized one reduction in (X, Y)-only co-Z conjugate addition by using a lazy reduction technique [3], [4]. This is based on the observation that a special form (CD-AB, EF-AB) in conjugate addition can be performed in only two modular reductions rather than three modular reductions. In Algorithm 7, step 8, 15, 16, 18, and 19 describe the special form (CD-AB, EF-AB). Step 8, 15, and 18 perform the multiplication without the reduction. Instead, we perform the lazy reduction with subtraction in step 16 and 19. The detailed descriptions are given in Algorithm 8. In Steps $1 \sim 3$, $2n$-word subtraction is performed. Afterwards, we perform the masked addition on the higher intermediate results by following steps 4$\sim$12 in Algorithm 4. Finally, ECC-friendly Montgomery reduction is performed on the intermediate results by following Algorithm 5. The technique replaces one $2n$-word reduction in two n-word subtractions, which optimize the scalar multiplication by 2.41 and 2.35 percent for P-256 and P-384, respectively.

P-521 curve is the Mersenne prime, which provides a very efficient reduction routine. For the n-word reduction, one 521-bit addition or one 521-bit subtraction is required. For the $2n$-word reduction, it only requires two 521-bit addition operations. The lazy reduction is not considered because the $2n$-word reduction on P-521 is exactly equal to two n-word addition operations.

Algorithm 5. Generic ECC-Friendly Montgomery Algorithm

Input: Modulus M, constant M_I= (-M^{-1}mod R) mod2^{32} where R is Montgomery radix, 96-bit accumulated results AC, intermediate result P=A×B where A, B are operands, length LEN, carry CY, borrow BW
Output: Result OT

1: AC← 0 {Initialization}
2: **for** i=0 to LEN-1 by 1 **do**
3: **for** j=1 to i-1 by 1 **do**
4: AC← AC+OT[j]×M[i-j] {Algorithm. 6}
5: **end for**
6: AC← AC+P[i] {Accumulation}
7: OT[i]← ACmod2^{32} {Mov}
8: AC← AC≫ 32 {Shifting}
9: AC← AC+OT[i] {Add}
10: **end for**
11: **for** i=LEN to 2×LEN-2 by 1 **do**
12: **for** j=i-LEN+1 to LEN-1 by 1 **do**
13: **if** j = = i-LEN+1 **then**
14: AC← AC+OT[j]·2^{32} -OT {Constant (0xFFFFFFFF)}
15: **else**
16: AC← AC+OT[j]×M[i-j] {Algorithm. 6}
17: **end if**
18: **end for**
19: AC← AC+P[i] {Accumulation}
20: OT[i-LEN]← AC mod 2^{32} {Saving}
21: AC← AC≫ 2^{32} {Shifting}
22: **end for**
23: AC← AC+P[2× LEN-1] {Accumulation}
24: OT[len-1]← AC mod 2^{32} {Saving}
25: OT[len]← (AC div 2^{32}) mod 2^{32}
26: CY← {(AC div2^{64})⊕2^{33} -1}+1
27: BW← 0 {Masked Final Subtraction}
28: **for** i=0 to LEN-1 by 1 **do**
29: BW‖ OT[i]← OT[i]-(M[i]&CY)-BW
30: **end for**
31: **return** OT

Algorithm 6. Partial Product for Special Modulus

Input: Operand OT, modulus M, intermediate result AC
Output: Intermediate result AC

1: **if** M == 0xFFFFFFFF **then**
2: AC← AC+OT·2^{32} -OT {Add, Sub}
3: **else if** M == 0x00000000 **then**
4: no operation {Skip}
5: **else if** M == 0x00000001 **then**
6: AC← AC+OT {Add}
7: **else if** M == 0xFFFFFFFE **then**
8: AC← AC+OT·2^{32} -OT-OT {Add, 2Sub}
9: **end if**
10: **return** AC

TABLE 2
Comparison of Code Size (Bytes) and Execution Time (Clock Cycles) for Different Multi-Precision Multiplication, Squaring and Modular Reduction Implementations on Cortex-M0+

Implementation	Size	256	384	448	512	1024	2048
Multiple precision multiplication (1-cycle multiplier)							
Düll et al. [6]	2,176	1,294	–	–	–	–	–
Wenger et al. [26]	3,456	2,173	–	–	–	–	–
Micro–ECC [18]	154	2,979	6,459	8,703	11,284	44,019	174,003
This work[1]	152	2,915	6,315	8,507	11,028	42,995	169,907
This work[2]	222	2,644	5,647	7,573	9,784	37,696	146,095
This work[2] (K)	–	–	–	–	9,394	31,723	116,698
Multiple precision squaring (1-cycle multiplier)							
Düll et al. [6]	1,394	857	–	–	–	–	–
Micro–ECC [18]	168	2,032	4,202	5,582	7,155	26,812	103,756
This work[1]	164	1,968	4,058	5,386	6,899	25,788	99,660
This work[2]	268	1,531	3,121	4,127	5,270	19,471	74,751
This work[2] (K)	–	–	–	–	5,601	17,589	61,724
Montgomery reduction (1-cycle multiplier)							
This work[2]	364	2,924	6,074	8,081	10,377	39,104	151,856
This work[†] (E)	342	1,713	3,385	–	–	–	–

The operands range from 256 to 2,048 bits (K: Karatsuba; E: ECC-friendly Montgomery reduction). [1]minimum size, [2]partially unrolled; [†]P-192 support: 1,082 clock cycles.

Algorithm 7. Lazy Reduction for a Special form (`CD-AB`, `EF-AB`) of (X, Y)-only co-Z Conjugate Addition (`XYCZ-ADDC`)

Input: (X_1, Y_1) and (X_2, Y_2) where $P \equiv (X_1, Y_1, Z)$ and $Q \equiv (X_2, Y_2, Z)$ for $Z \in \mathbb{F}_q$, $P, Q \in E(\mathbb{F}_q)$
Output: (X_3, Y_3) and (X_3', Y_3') where $P + Q \equiv (X_3, Y_3, Z_3)$ and $P - Q \equiv (X_3', Y_3', Z_3)$ for $Z_3 \in \mathbb{F}_q$

1: $A \leftarrow X_2 - X_1$
2: $A \leftarrow A^2$
3: $B \leftarrow X_1 \times A$
4: $C \leftarrow X_2 \times A$
5: $A \leftarrow Y_2 - Y_1$
6: $D \leftarrow A^2$
7: $E \leftarrow C - B$
8: $E \leftarrow Y_1 \times E$ {No reduction}
9: $C \leftarrow B + C$
10: $X_3 \leftarrow D - C$
11: $D \leftarrow Y_1 + Y_2$
12: $X_3' \leftarrow D^2$
13: $X_3' \leftarrow X_3' - C$
14: $Y_3 \leftarrow B - X_3'$
15: $Y_3 \leftarrow A \times Y_3$ {No reduction}
16: $Y_3 \leftarrow Y_3 - E$ {Algorithm 8}
17: $Y_3' \leftarrow X_3' - B$
18: $Y_3' \leftarrow D \times Y_3'$ {No reduction}
19: $Y_3' \leftarrow Y_3' - E$ {Algorithm 8}
20: **return** $\big((X_3, Y_3), (X_3', Y_3')\big)$

5 Performance Evaluation and Comparison

5.1 Evaluation of Modular Multiplication and Squaring

Findings from the evaluation of multi-precision multiplication, squaring, and Montgomery algorithm on Cortex-M0+ are given in Table 2. For multiplication and squaring operations, we evaluate three implementations. First, a memory-efficient implementation is proposed, which is based on a straightforward optimization of Micro-ECC MAC routine with half word trick (`UXTH` technique). Second, we partially unroll the memory-efficient implementation, which use two MAC routines for lower partial products ($A[i] \times B[j]$ where $i + j < n$) and higher partial products ($A[i] \times B[j]$ where $n - 1 < i + j < 2n - 1$), respectively. Third, we evaluate the impact of subtractive Karatsuba multiplication for long integer, which replaces a multiplication of two n-word multiplications into three $\frac{n}{2}$-word multiplications with several additions [11].

Algorithm 8. Lazy Modular Subtraction

Input: Operands (`A`, `B`), modulus `M`, length `LEN` (P-256: 8, P-384: 12)
Output: Modular subtraction (`A-B mod M`)

1: `K`$\leftarrow$ `0`, `Q`$\leftarrow$`0`
2: **for** `i=0 to 2·LEN-1 by 1` **do**
3: `K`$\|$ `AC[i]`$\leftarrow$ `A[i]-B[i]-(K&1)`
4: **end for**
5: $\texttt{AC}_\texttt{L} \leftarrow \texttt{AC} \bmod 2^{\texttt{LEN}}$
6: $\texttt{AC}_\texttt{H} \leftarrow (\texttt{AC} \gg \texttt{LEN}) \bmod \texttt{(M\&K)}$ {Step 4∼12 in Algorithm 4}
7: $\texttt{AC} \leftarrow \texttt{AC}_\texttt{H} \| \texttt{AC}_\texttt{L} \bmod \texttt{M}$ {Algorithm 5}
8: **return** `AC`

Compared with speed-optimization of Düll et al. and multiplication of Wenger et al. [6], [26], our proposed memory-efficient implementation reduce the execution timing by 51∼56 percent but with a significant reduction on code size (i.e., 90∼96 percent). Compared with Micro-ECC for multiplication, the minimum size implementation results in a smaller code size (i.e., 2 bytes) and faster execution time. The partially unrolled version requires a larger code size (i.e., 68 bytes), but enjoys an improved performance (i.e., increased by 11∼16 percent). The subtractive Karatsuba algorithm shows a higher performance than the partially unrolled version from 512-bit. In particular, the 2048-bit multiplication is accelerated by 20 percent, which is an useful implementation for RSA. For the squaring implementation, Düll et al. showed 44∼56 percent enhancements in execution timing but it increases the code size by $80 \sim 88$ percent. Compared with Micro-ECC, the minimum size implementation has a smaller code size (4 bytes) because Micro-ECC handles the partial products ($A[i] \times A[j]$ where $i \neq j$) with two accumulation steps. The partially unrolled version, which uses both SBD and the proposed 2-way carry-catcher methods, achieves 25∼28 percent performance improvement. The Karatsuba algorithm outperforms the partially unrolled version in 1024-bit and 2048-bit by 9.7 and 17 percent, respectively. For the modular reduction, both the original and the ECC-friendly Montgomery reduction are evaluated. The original Montgomery reduction supports random modulus, which is useful for RSA implementation. The ECC-friendly Montgomery reduction is specialized for NIST P-256 and P-384 curves. These special cases are efficiently handled with the ECC-friendly Montgomery reduction and this shows better performance than original Montgomery reduction by 41 and 44 percent for P-256 and P-384, respectively.

TABLE 3
Comparison of ECC Scalar Multiplication Implementations in Code Size (Bytes) and Execution Time (10^3 Clock Cycles) on Cortex-M0+ (Frequency: 48 MHz)

Implementation		Curve25519	P-256	P-384	P-521
Düll et al. [6]	cycles	3,590	–	–	–
	bytes	7,900	–	–	–
Wenger et al. [26]	cycles	–	10,730	–	–
	bytes	–	7,168	–	–
Micro–ECC† [18]	cycles	–	18,088	–	–
	bytes	–	3,320	–	–
This work	cycles	–	16,900	52,251	84,833
	bytes	–	2,900	2,928	2,524

5.2 Evaluation of ECC

A comparative summary of the ECC implementations in terms of code size (bytes) and execution time (clock cycles) are given in Table 3. The fastest implementations of Curve25519 and P-256 are achieved by Düll et al. [6] and Wenger et al. [26], respectively. Both implementations require large code sizes (7.9KB for Curve25519 and 7.2 KB for P-256). The work of Micro-ECC [18] has the memory-efficient ECC implementation, which requires only 3.3 KB for P-256, in addition to practically fast performance. However, Micro-ECC is implemented in non constant-timing. All modular reduction operations use conditional statements for the final addition or the subtraction operation. For the finite field inversion operation, Micro-ECC adopts the non constant-timing Extended euclidean Algorithm (EEA). Furthermore, the implementations of fast reduction and EEA require independent routines, which increase the code size.

Our implementation supports three NIST curves: P-256, P-384 and P-521. As shown in the table, the implementation of NIST P-256 requires an execution time of 1.69M clock cycles while only consuming a code size of 2.9K bytes, while the P-384 and P-521 have execution time of roughly 5.23M clock cycles and 8.48M clock cycles with the memory footprint of 2.93K bytes and 2.52K bytes. Our implementations can be run in constant-timing with a simple Fermat based inversion and a masked reduction. Together with the compact ECC-friendly Montgomery reduction and lazy reduction, the P-256 implementation reduce the code size by a factor of 13 percent and speed up the execution time by roughly 4.5 percent when comparing with Micro-ECC. There is no doubt that the high security P-521 has the slowest performance among the three curves but it achieves the smallest code size, since the curve is Mersenne prime having the simplest reduction operation.

6 Conclusion

Lightweight implementation of public-key cryptography on embedded microprocessors is an important and active research topic in security and cryptography community. There exist a large body of research on fast implementation of elliptic curve cryptography on 8-bit and 16-bit microcontrollers but very few on 32-bit ARMv6-M series platform. In this paper, we presented the compact and scalable implementation of 3 NIST curves P-256, P-384 and P-521 on 32-bit ARMv6-M processors, while taking both the execution time and memory consumption into account. In particular, we proposed refined MAC and 2-way carry catcher method to improve the performance of multi-precision multiplication and squaring. We also optimized the Montgomery algorithm by using the lazy reduction approach. Our implementation of NIST P-256 scalar multiplication only requires 1.69M clock cycles with the memory footprint of roughly 2.9K bytes. A very interesting work based on our work is to integrate the new IETF standard curves Curve25519 and Curve448 into our ECC library. We hope the work can further promote the deployment of NIST compliant curves on ARM based IoT devices.

Acknowledgments

This work of Hwajeong Seo was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No. NRF-2017R1C1B5075742). This research of Howon Kim was supported by the MSIT (Ministry of Science and ICT), Korea, under the Industry4.0s research and development program(S0604-17-1002) supervised by the NIPA(National IT Industry Promotion Agency).

References

[1] ARM, ARM developer suite version 1.2, 2000. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.dui0068b/DUI0068.pdf
[2] ARM, Cortex–M0 Processor, Feb. 2016. [Online]. Available: http://www.arm.com/products/processors/cortex-m/cortex-m0.php
[3] J.-C. Bajard, S. Duquesne, and M. Ercegovac, "Combining leak–resistant arithmetic for elliptic curves defined over Fp and RNS representation," *Publications Mathématiques de Besançon: Algèbre et Théorie des Nombres*, pp. 67–87, 2013.
[4] K. Bigou and A. Tisserand, "Single base modular multiplication for efficient hardware RNS implementations of ECC," in *Proc. Int. Workshop Cryptographic Hardware Embedded Syst.*, 2015, pp. 123–140.
[5] R. De Clercq, L. Uhsadel, A. Van Herrewege, and I. Verbauwhede, "Ultra low-power implementation of ECC on the ARM Cortex–M0+," in *Proc. 51st Annu. Des. Autom. Conf.*, 2014, pp. 1–6.
[6] M. Düll, B. Haase, G. Hinterwälder, M. Hutter, C. Paar, A. H. Sánchez, and P. Schwabe, "High-speed Curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers," *Des. Codes Cryptography*, vol. 77, no. 2/3, pp. 493–514, 2015.
[7] S. Gueron and V. Krasnov, "Fast prime field elliptic-curve cryptography with 256-bit primes," *J. Cryptographic Eng.*, vol. 5, no. 3, pp. 141–151, 2015.
[8] N. Gura, A. Patel, A. Wander, H. Eberle, and S. C. Shantz, "Comparing elliptic curve cryptography and RSA on 8-bit CPUs," in *Proc. Cryptographic Hardware Embedded Syst.*, 2004, pp. 119–132.
[9] M. Hutter, M. Joye, and Y. Sierra, "Memory-constrained implementations of elliptic curve cryptography in co-Z coordinate representation," in *Proc. Int. Conf. Cryptology Africa*, 2011, pp. 170–187.
[10] M. Hutter and P. Schwabe, "NaCl on 8-bit AVR microcontrollers," in *Proc. Int. Conf. Cryptology in Africa*, 2013, pp. 156–172.
[11] A. Karatsuba and Y. Ofman, "Multiplication of multidigit numbers on automata," *Soviet Physics Doklady*, vol. 7, 1963, Art. no. 595.
[12] A. Liu and P. Ning, "TinyECC: A configurable library for elliptic curve cryptography in wireless sensor networks," in *Proc. Int. Conf. Inf. Process. Sensor Netw.*, 2008, pp. 245–256.
[13] Z. Liu and J. Großschädl, "New speed records for montgomery modular multiplication on 8-bit AVR microcontrollers," in *Proc. Int. Conf. Cryptology*, 2014, pp. 215–234.
[14] Z. Liu, J. Großschädl, Z. Hu, K. Järvinen, H. Wang, and I. Verbauwhede, "Elliptic curve cryptography with efficiently computable endomorphisms and its hardware implementations for the internet of things," *IEEE Trans. Comput.*, vol. 66, no. 3, pp. 773–785, May 2017.

[15] Z. Liu, X. Huang, Z. Hu, M. K. Khan, H. Seo, and L. Zhou, "On emerging family of elliptic curves to secure internet of things: ECC comes of age," *IEEE Trans. Dependable Sec. Comput.*, vol. 14, no. 3, pp. 237–248, May/Jun. 2017.
[16] Z. Liu, E. Wenger, and J. Großschädl, "MoTE–ECC: Energy-scalable elliptic curve cryptography for wireless sensor networks," in *Proc. Appl. Cryptography Netw. Secur*, 2014, pp. 361–379.
[17] J. López and R. Dahab, "High-speed software multiplication in $\mathbb{F}_{2^m}$," in *Proc. 1st Int. Conf. Progress Cryptology*, 2000, pp. 203–212.
[18] K. MacKay, "ECDH and ECDSA for 8-bit, 32-bit, and 64-bit processors," Jan. 2017. [Online]. Available: https://github.com/kmackay/micro-ecc
[19] P. L. Montgomery, "Modular multiplication without trial division," *Math. Comput.*, vol. 44, no. 3, pp. 519–521, 1985.
[20] J. Renes, P. Schwabe, B. Smith, and L. Batina, "μKummer: Efficient hyperelliptic signatures and key exchange for microcontrollers," in *Cryptographic Hardware and Embedded Systems*, B. Gierlichs and A. Poschmann, Eds. Berlin, Germany: Springer, pp. 301–320, 2016.
[21] C. Research, "Sec 2: Recommended elliptic curve domain parameters," *Standards Efficient Cryptography Group*, 2000.
[22] M. Rivain, "Fast and regular algorithms for scalar multiplication over elliptic curves," *IACR Cryptology ePrint Archive*, vol. 2011, 2011, Art. no. 338.
[23] H. Seo, Z. Liu, J. Choi, and H. Kim, "Multi-precision squaring for public-key cryptography on embedded microprocessors," in *Progress in Cryptology*. New York, NY, USA: Springer, pp. 227–243, 2013.
[24] A. Szekely, S. Tillich, et al., "Algorithm exploration for long integer modular arithmetic on a SPARC V8 processor with cryptography extensions," in *Proc. 2nd Int. Conf. Embedded Softw. Syst.*, 2005, pp. 187–194.
[25] T. Unterluggauer and E. Wenger, "Efficient pairings and ECC for embedded systems," in *Cryptographic Hardware and Embedded Systems*. New York, NY, USA: Springer, pp. 298–315, 2014.
[26] E. Wenger, T. Unterluggauer, and M. Werner, "8/16/32 shades of elliptic curve cryptography on embedded processors," in *Progress in Cryptology*. New York, NY, USA: Springer, pp. 244–261, 2013.
[27] T. Yanik, E. Savaş, and Ç. K. Koç, "Incomplete reduction in modular arithmetic," in *IEEE Proc. Comput. Digit. Tech.* 2002, vol. 2002, pp. 46–52.

Zhe Liu received the PhD degree from the Laboratory of Algorithmics, Cryptology and Security (LACS), University of Luxembourg, Luxembourg. He is a professor with the College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, China and a research associate with SnT, University of Luxembourg, Luxembourg. His PhD thesis has received the prestigious FNR Awards 2016-Outstanding PhD Thesis Award for his contributions in cryptographic engineering on IoT devices. His research interests include computer arithmetic and information security. He has co-authored more than 70 research peer-reviewed journal and conference papers, including the *IEEE Transactions on Computers*, the *IEEE Transactions on Information Forensics and Security*, the *IEEE Transactions on Dependable and Secure Computing*, IACR CHES etc. He is a senior member of the IEEE.

Hwajeong Seo received the BSEE, MS and PhD degrees in computer engineering from Pusan National University. He is currently an assistant professor with Hansung University. His research interests include Internet of Things and information security. He is the recipient of WISA 2016 Best Paper Award. He has published more than 100 research papers in computer arithmetic and cryptographic engineering, including more than 50 SCI journal papers. He is a member of International Association for Cryptologic Research (IACR).

Aniello Castiglione received the PhD degree in computer science from the University of Salerno, Italy. Actually he is an adjunct professor with the University of Salerno, Italy, and with the University of Naples "Federico II", Italy. He received the Italian national qualification as an associate professor in computer science. He published more than 150 papers in international journals and conferences. He served as program chair and TPC member in around 130 international conferences. One of his papers has been selected as "Featured Article" in the IEEE Cybersecurity initiative. He served as a reviewer for several international journals and he is the managing editor of two ISI-ranked international journals. He acted as a guest editor in several journals and serves as an editor in several editorial boards of international journals. His current research interests include information forensics, digital forensics, security and privacy on cloud, communication networks, and applied cryptography. He is a member of several associations, including IEEE and ACM. He has been involved in forensic investigations, collaborating as a consultant with several law enforcement agencies. From its establishment, he is a member of the European Electronic Crime Task Force (ECTF). He is a senior member of the IEEE.

Kim-Kwang Raymond Choo received the cloud technology endowed professorship with The University of Texas at San Antonio. He was a visiting scholar with INTERPOL Global Complex for Innovation between October 2015 and February 2016, and a visiting Fulbright scholar with Rutgers University School of Criminal Justice and Palo Alto Research Center (formerly Xerox PARC) in 2009. In 2016, he was named the Cybersecurity Educator of the Year - APAC (Cybersecurity Excellence Awards are produced in cooperation with the Information Security Community on LinkedIn), and in 2015 he and his team won the Digital Forensics Research Challenge organized by Germany's University of Erlangen-Nuremberg. He is the recipient of ESORICS 2015 Best Paper Award, 2014 Highly Commended Award by the Australia New Zealand Policing Advisory Agency, Fulbright Scholarship in 2009, 2008 Australia Day Achievement Medallion, and British Computer Society's Wilkes Award in 2008. He is also a fellow of the Australian Computer Society, and an honorary commander of the 502nd Air Base Wing, Joint Base San Antonio-Fort Sam Houston. He is a senior member of the IEEE.

Howon Kim received the BSEE degree from Kyungpook National University, Daegu, Republic of Korea, in 1993 and the MS and PhD degrees in electronic and electrical engineering from the Pohang University of Science and Technology (POSTECH), Pohang, Republic of Korea, in 1995 and 1999, respectively. From July 2003 to June 2004, he studied with the COSY group at the Ruhr-University of Bochum, Germany. He was a senior member of the technical staff with the Electronics and Telecommunications Research Institute (ETRI), Daejeon, Republic of Korea. He is currently working as an associate professor with the Department of Computer Engineering, School of Computer Science and Engineering, Pusan National University, Pusan, Republic of Korea. His research interests include RFID technology, sensor networks, information security, and computer architecture. Currently, his main research interests include mobile RFID technology and sensor networks, public key cryptosystems, and their security issues. He is a member of the IEEE, and the International Association for Cryptologic Research (IACR).