Received: 12 August 2018 | Revised: 24 February 2019 | Accepted: 12 March 2019

DOI: 10.4218/etrij.2018-0445

ORIGINAL ARTICLE

WILEY ETRI Journal

# Compact implementations of Curve Ed448 on low-end IoT platforms

**Hwajeong Seo**

Division of IT Convergence Engineering, Hansung University, Seoul, Rep. of Korea

**Correspondence**
Hwajeong Seo, Division of IT Convergence Engineering, Hansung University, Seoul, Rep. of Korea. Email: hwajeong84@gmail.com

**Funding information**
This research was supported by the Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIT) (No.2018-0-00264, Research on Blockchain Security Technology for IoT Services).

Elliptic curve cryptography is a relatively lightweight public-key cryptography method for key generation and digital signature verification. Some lightweight curves (eg, Curve25519 and Curve Ed448) have been adopted by upcoming Transport Layer Security 1.3 (TLS 1.3) to replace the standardized NIST curves. However, the efficient implementation of Curve Ed448 on Internet of Things (IoT) devices remains underexplored. This study is focused on the optimization of the Curve Ed448 implementation on low-end IoT processors (ie, 8-bit AVR and 16-bit MSP processors). In particular, the three-level and two-level subtractive Karatsuba algorithms are adopted for multi-precision multiplication on AVR and MSP processors, respectively, and two-level Karatsuba routines are employed for multi-precision squaring. For modular reduction and finite field inversion, fast reduction and Fermat-based inversion operations are used to mitigate side-channel vulnerabilities. The scalar multiplication operation using the Montgomery ladder algorithm requires only 103 and 73 M clock cycles on AVR and MSP processors.

**KEYWORDS**
efficient elliptic curve cryptography implementation, embedded processors, Internet of Things

## 1 | INTRODUCTION

"Internet of Things" (IoT) is becoming increasingly widespread in smart cities and smart nations. For example, a report from Gartner estimated that "Sixty-three million IoT devices will be attempting to connect to the enterprise network each second by 2020" [1]. Conceptually, an IoT deployment consists of a (large) number of devices, things, and/or nodes (hereafter referred to as "things") connected to the Internet (and/or to each other). These include smart mobile devices (eg, Android and iOS devices), smart switches, wearable devices (eg, smart watches such as FitBit and smart uniforms in a military context), sensors (eg, in a smart grid), and components of a large system such as next-generation air transportation systems and smart vehicles.

IoT devices are not generally designed to meet security standards, particularly when their majority are resource-constrained embedded devices (eg, having 8-bit AVR and 16-bit MSP processors). However, owing to the nature the data collected, disseminated, and shared by these things, as well as the deployment of related applications, it is important to ensure the security of IoT things and infrastructure as well as to preserve the privacy of the data and the computation results. Unsurprisingly, security and privacy are attracting increasing interest within the IoT community (eg, the provision of secure and efficient communication between IoT things and the infrastructure).

Protocols based on elliptic curve cryptography (ECC) have also attracted attention owing to its short key length and low execution time, which makes it particularly attractive for IoT deployment. In addition to being resource-constrained,

IoT things also have limited storage, computation power, and battery capacity; therefore, efficient implementation of ECC on IoT devices is an active research area.

A classic ECC implementation study is that of Gura et al. [2], where the authors proposed different security levels in NIST/SECG elliptic curves and RSA on two 8-bit microcontrollers. Since their seminal work, several approaches to efficient EEC implementation have been proposed, namely, new elliptic curve models (eg, Curve25519 [3], Curve Ed448 [4]), new techniques for multi-precision multiplication (eg, the operand caching method [5]), reverse product scanning [6], and efficient primes (eg, OPFs [7]). Existing implementations generally focus on low-level security owing to their efficiency. For example, implementing NIST P256 and Curve25519 on low-end IoT processors such as 8-bit AVR and 16-bit MSP processors. However, there are few high-level ECC implementations on such platforms, and in particular, it should be noted that Curve Ed448 has been adopted in the upcoming Transport Layer Security 1.3 (TLS 1.3) to secure communication [8].

Herein, the first evaluation of Curve Ed448 on low-end IoT processors is presented. To maximize performance, the performance-critical multi-precision multiplication and squaring operations on 8-bit AVR and 16-bit MSP processors are optimized by combining various techniques. In particular, multiple-level subtractive Karatsuba algorithms are used and implemented using C and Assembly. For multi-precision multiplication, three-level and two-level Karatsuba are used on AVR and MSP processors, and for multi-precision squaring, two-level Karatsuba is used on both processors to reduce computation complexity. For modular reduction and finite field inversion, fast reduction and Fermat-based inversion techniques are used to achieve constant timing. Field arithmetic is implemented in a constant time and regular fashion to reduce further the risk of side-channel attacks. For group arithmetic, the Montgomery ladder algorithm is adopted for scalar multiplication, and an execution time of 103 M and 73 M clock cycles is achieved. The experimental results demonstrate that high-security EEC is still feasible for IoT platforms.

The paper is organized as follows. In Section 2, preliminaries and related work are introduced. In Section 3, the optimized implementations of multi-precision multiplication and squaring operations on 8-bit AVR and 16-bit processors are presented. In Section 4, the proposed multiplication and squaring implementations are applied to Curve448, and the evaluation results are presented in Section 5. Finally, the paper is concluded in Section 6.

## 2 | PRELIMINARIES

### 2.1 | Target curve: Curve Ed448

Edwards curves were first proposed in 2007 [9]. Compared to elliptic curves in normal form, Edwards curves can be easier to implement securely because they support complete addition formulas without exceptional cases (division by zero). That is, Edwards curves are faster and simpler to handle than NIST curves [4]. Most existing implementations of (twisted) Edwards curves support at most 128-bit security [10,11], and relatively few have high-security levels. Ed448-Goldilocks is an example of an elliptic curve with high-security level (approximately 224 bits), as suggested by Hamburg [4]. The arithmetic is expressed as

$$E: y^2 + x^2 = 1 + dx^2y^2$$

defined over the field $\mathbb{F}_{2^{448}-2^{224}-1}$ with curve parameter $d = -39\,081$. The curve satisfies the SafeCurves policies and criteria, which mitigate security vulnerabilities and flaws in existing NIST curves [12]. Furthermore, the curve has been selected as the ECC standard for TLS 1.3 [8].

### 2.2 | Target processors: 8-bit AVR and 16-bit MSP processors

AVR is a representative 8-bit embedded processor with a clock frequency of 8–32 MHz, 128–256 KB EEPROM, and 4–8 KB RAM. The processor has 32 general purpose registers. Among these, six registers (`R26-R31`) serve as addressing pointers, and the remaining 26 are used for general purposes. The registers are categorized into call-used registers (`R18-R27, R30-R31`) and call-saved registers (`R2-R17, R28-R29`). The call-used registers are used freely in assembler subroutines, whereas the call-saved registers should be saved before assembler subroutines and restored thereafter. In terms of execution time, one arithmetic instruction requires one clock cycle, whereas memory instructions or 8-bit multiplication ($16 \leftarrow 8 \times 8$) require two clock cycles. In particular, partial products are obtained from the least significant registers (`R0-R1`). The detailed description of the instruction set for the AVR processor is given in Table 1.

MSP is a representative 16-bit embedded processor with a clock frequency of 8 MHz–25 MHz, 32 KB–48 KB FLASH memory, and 10 KB of RAM. It has 12 general purpose registers ranging from `R4` to `R15`. All registers can be used as addressing pointers and for general operations. The arithmetic and logical operations are performed within one clock cycle, whereas memory access requires several clock cycles. The MSP430 architecture provides seven different addressing modes: register direct, indexed, symbolic, absolute, register indirect, indirect auto-increment, and immediate. All addressing modes can be used for the source operand, but only the first four are available for the destination operand. Unlike AVR, MSP supports a memory-mapped hardware multiplier for $16 \times 16$-bit multiplication. Chip designers can decide whether to include a hardware multiplier, as in some

**TABLE 1** Instruction set summary for AVR

| Instructions | Operands | Description | Operation |
|---|---|---|---|
| ADD | Rd, Rr | Add without carry | Rd ← Rd + Rr |
| ADC | Rd, Rr | Add with carry | Rd ← Rd + Rr + C |
| SUB | Rd, Rr | Sub without borrow | Rd ← Rd − Rr |
| SBC | Rd, Rr | Sub with borrow | Rd ← Rd − Rr − B |
| EOR | Rd, Rr | Exclusive OR | Rd ← Rd ⊕ Rr |
| LSL | Rd | Logical shift left | C\|Rd ← «1 |
| LSR | Rd | Logical shift right | Rd\|C ← 1 » Rd |
| MUL | Rd, Rr | Multiplication | {R1, R0}← Rd × Rr |
| LD | Rr, X | Data load | {X} ← Rr |
| ST | X, Rr | Data store | {Rr} ← X |

applications smaller chip size is preferable and multiplication operations are not required. There are three different multiplication modes in the MSP430 hardware multiplier: MPY (unsigned multiplication of 8-bit and 16-bit operands), MPYS (signed multiplication of 8-bit and 16-bit operands), and MAC (unsigned multiply-and-accumulate multiplication of 8-bit and 16-bit operands). The multiplication mode is determined by memory loading procedures. Each mode has a specific memory address, and the user can assign the operands to the memory address to trigger the specific multiplication mode. The cost of multiplication is the sum of the cost of writing the operands and reading the result to/from a multiplier's memory address. The detailed description of the instruction set for the MSP processor is given in Table 2.

## 2.3 | Previous implementations

Software implementations of Curve Ed448 are mainly performed on high-end 32-bit or 64-bit processors [4]. As these processors are more powerful, high-security Curve Ed448 implementations are easily established. However, low-end devices such as 8-bit AVR and 16-bit MSP have low storage, computation power, and battery capacity. Thus, implementing Curve Ed448 on embedded processors is challenging and impractical in several real-world applications. Multi-precision multiplication and squaring are the most performance-critical field operations for public-key operations. There exist several multiplication operations on embedded processors, ranging from the classic operand scanning (OS) method to more sophisticated techniques (eg, hybrid scanning (HS) multiplication [2] and operand caching [5]). The OS method performs the partial product operation in a row [13]. An alternative approach is the product scanning (PS) method, whereby all partial products are computed in a column, thus eliminating the need to reload the intermediate results [14]. The OS method requires a number of accumulator registers to store the intermediate results, whereas PS requires a number of memory access registers for the operands. In particular, the PS method on MSP430 processors has been extensively studied because these processors support a memory-mapped hardware multiplier and MAC mode [15–17]. The MAC mode efficiently computes column-wise partial products and stores the intermediate results in the memory during the MAC routine. After the MAC routine, the intermediate results are moved from the memory-mapped hardware multiplier to the result memory.

To bridge the gap between OS and PS, the HS method combines their features [2]. By adjusting row and column width, the number of operands and intermediate result accesses is optimized. An operand caching method designed to reduce the number of `load` operations by caching the operands was presented in CHES 2011 [5]. Later in 2012 and 2013, Seo and Kim proposed a further optimization of operand caching [18,19] and presented the consecutive operand caching method, which has a continuous operand caching process. More recently, the Karatsuba technique has been adopted to improve performance, including implementations of

**TABLE 2** Instruction set summary for MSP

| Mnemonics | Operands | Description | Operation |
|---|---|---|---|
| ADD | Rr, Rd | Add without carry | Rd ← Rd + Rr |
| ADDC | Rr, Rd | Add with carry | Rd ← Rd + Rr + C |
| AND | Rr, Rd | Logical AND | Rd ← Rd & Rr |
| XOR | Rr, Rd | Exclusive OR | Rd ← Rd ⊕ Rr |
| RLA | Rd | Logical shift left | C\|Rd ← Rd « 1 |
| RRA | Rd | Logical shift right | Rd\|C ← 1 » Rd |
| RLC | Rd | Rotate left through carry | C\|Rd ← Rd « 1\|\|C |
| RRC | Rd | Rotate right through carry | Rd\|C ← C\|\|1 » Rd |

Curve25519 on embedded processors [20]. These implementations achieved the speed record on three different embedded processors: 8-bit AVR, 16-bit MSP, and 32-bit ARM. The performance enhancements are primarily due to the subtractive Karatsuba multiplication and squaring implementations suggested by Hutter and Schwabe [21]. The authors proved that subtractive Karatsuba multiplication can achieve performance enhancements using short (48-bit) integers. In particular, 256-bit integer multiplication using three-level Karatsuba requires only 4936 clock cycles, whereas other state-of-art methods require approximately 6115–6180 clock cycles [5,18,19]. For long integer multiplication, Seo, Liu, Nogami, Choi, and Kim used multiple-level Karatsuba based on Hutter and Schwabe's work [22]. They obtained 512, 1024, and 2048-bit results in C wrapper with 256-bit assembly-optimized implementations. However, there are no implementations for 448-bit operand length, as this is a unique operand length and not a multiple of that in Hutter and Schwabe's work.

Herein, highly optimized multiplication and squaring implementations with different lengths (ie, 56, 112, 224, and 448 bits) on 8-bit AVR processors are presented. Multiple-level subtractive Karatsuba algorithms are used. For multiplication, three-level and two-level Karatsuba are used on AVR and MSP, whereas for squaring, two-level Karatsuba is used on both processors to reduce computation complexity.

## 3 | OPTIMIZATION TECHNIQUES FOR MULTI-PRECISION MULTIPLICATION AND SQUARING

### 3.1 | Multiplication

The subtractive Karatsuba algorithm is an efficient integer multiplication method on 8-bit AVR processors [21]. To perform multiple-level Karatsuba for 448-bit integer multiplication, the length of the starting partial product is set to 56 bits (56 → 112 → 224 → 448). In [21], 48-bit multiplication is accelerated by one-level Karatsuba. However, a 56-bit word cannot be divided into two full 8-bit words on an 8-bit architecture ($3.5 = 56/(2 \cdot 8)$). Accordingly, the product scanning method is chosen for 56-bit integer multiplication. The operand length is only 56 bits, and thus, 14 general purpose registers are required to hold all the operands. Subsequently, three and one general purpose registers are assigned to the result accumulation and the zero parameter, respectively. A total of 18 registers are required for product scanning.

For 112-bit multiplication ($A \times B$), the one-level Karatsuba algorithm is performed. First, the lower partial product ($C_L = A \bmod 2^{56} \times B \bmod 2^{56}$) is computed and the intermediate result ($C = C_L + C_L \cdot 2^{56}$) is obtained. Then, the higher partial product ($C_H = A \operatorname{div} 2^{56} \times B \operatorname{div} 2^{56}$) is computed and the intermediate result ($C = C + C_H \cdot 2^{56} + C_H \cdot 2^{112}$) is obtained. Finally, the absolute differences of operands ($|A \bmod 2^{56} - A \operatorname{div} 2^{56}|$ $|B \bmod 2^{56} - B \operatorname{div} 2^{56}|$) are multiplied, and the results $C_M$ and the sign bit are obtained. The signed two's complements of the intermediate results are computed and then are added to obtain the final result ($C = C + C_M \cdot 2^{56}$). It is noted that all registers and two stack bytes are used.

For 224-bit multiplication, two-level Karatsuba is performed based on the 112-bit one-level Karatsuba. First, the lower partial product ($C_L = A \bmod 2^{112} \times B \bmod 2^{112}$) and the higher partial product ($C_H = A \operatorname{div} 2^{112} \times B \operatorname{div} 2^{112}$) are computed. Both intermediate results are stored in the memory owing to the limited number of registers. Then, the intermediate result ($C = C_L + (C_L + C_H) \cdot 2^{112} + C_H \cdot 2^{224}$) is obtained. For efficient computation, the value of $C_L$ is directly added to the value $C_H$ placed in the register. Finally, the absolute differences of the operands ($|A \bmod 2^{112} - A \operatorname{div} 2^{112}|$ $|B \bmod 2^{112} - B \operatorname{div} 2^{112}|$) are multiplied, and the results $C_M$ and the sign bit are obtained. The signed two's complements of the intermediate results are calculated and then are added to obtain the final result ($C = C + C_M \cdot 2^{112}$).

For 448-bit multiplication, three-level Karatsuba is performed based on the 224-bit two-level Karatsuba. First, the 224-bit-wise lower partial product ($C_L = A \bmod 2^{224} \times B \bmod 2^{224}$) and the 224-bit-wise higher partial product ($C_H = A \operatorname{div} 2^{224} \times B \operatorname{div} 2^{224}$) are computed. Then, the intermediate results yield the sum ($C = C_L + (C_L + C_H) \cdot 2^{224} + C_H \cdot 2^{448}$). Finally, the absolute differences of the operands ($|A \bmod 2^{224} - A \operatorname{div} 2^{224}|$, $|B \bmod 2^{224} - B \operatorname{div} 2^{224}|$) are multiplied, and the results $C_M$ and the sign bit are obtained. The signed two's complement of these are calculated and then are added to obtain the final the result ($C = C + C_M \cdot 2^{224}$). As the result of a 224-bit-wise partial product cannot be stored in a register, it is stored in the stack instead.

The subtractive Karatsuba algorithm is the most efficient integer multiplication method on 16-bit MSP processors [20]. To perform multiple-level Karatsuba for 448-bit integer multiplication, the length of the starting partial product is set to 112 bits (112 → 224 → 448).

In [20], 56-bit multiplication is accelerated by one-level Karatsuba. On the 8-bit AVR processor, a 56-bit starting partial product is chosen. However, on the 16-bit MSP processor, a 56-bit word is not divided by 16-bit words ($3.5 = 56/16$). Accordingly, a 112-bit starting partial product is chosen, which is efficiently implemented by the MAC and product scanning methods.

For 224-bit multiplication, one-level Karatsuba is performed based on 112-bit product scanning. First, the lower partial product ($C_L = A \bmod 2^{112} \times B \bmod 2^{112}$) and the higher partial product ($C_H = A \operatorname{div} 2^{112} \times B \operatorname{div} 2^{112}$) are

computed. Then, the absolute differences of the operands ($|A \bmod 2^{112} - A \operatorname{div} 2^{112}|$, $|B \bmod 2^{112} - B \operatorname{div} 2^{112}|$) are multiplied and the results $C_M$ and the sign bit are output. As the MSP processor cannot hold all the values in the registers, the stack storage is accessed through the stack pointer (`R1`) and the values are stored. Finally, the two's complement of the partial product ($C_M$) is calculated and then added to other partial products ($C_L$, $C_H$) in the stack storage. Then, the final result is obtained by adding the intermediate results ($C = C_L + (C_L + C_H - |C_M|) \cdot 2^{112} + C_H \cdot 2^{224}$). Similarly, 448-bit multiplication is performed using a two-level Karatsuba (consisting of two 224-bit one-level Karatsuba algorithms).

### 3.2 | Squaring

The integer squaring operation can be computed by the multiplication routine. However, the partial products ($A[i] \times A[j]$ where $A$ is an array of operands with $i \neq j$) can be efficiently computed using one partial product and one addition ($2 \times A[i] \times A[j]$) rather than two partial products and one addition ($A[i] \times A[j] + A[j] \times A[i]$). 56-bit and 112-bit squaring are evaluated, and a 112-bit starting partial product is selected (112 → 224 → 448). The 224-bit Karatsuba technique is chosen. For 112-bit squaring, the sliding block doubling (SBD) method is used [23]. It consists of three steps:

1. The partial products ($A[i] \times A[j]$, where $i \neq j$) are obtained by product scanning.
2. The intermediate results are doubled using the addition operation.
3. The remaining partial products ($A[i] \times A[j]$, where $i = j$) are calculated and added to the intermediate results.

For 112-bit squaring, 14 general purpose registers are used to hold the operands, whereas three and one general purpose registers are assigned to the result accumulation and the zero value, respectively. A total of 18 registers are used for 112-bit SBD.

For 224-bit squaring, one-level subtractive Karatsuba is performed based on the 112-bit SBD implementations. First, the lower partial product ($C_L = A \bmod 2^{112} \times A \bmod 2^{112}$) and the higher partial product ($C_H = A \operatorname{div} 2^{112} \times A \operatorname{div} 2^{112}$) are computed. Unlike in the case of multiplication, only one side of the operand should be stored. Accordingly, the remaining general purpose registers are used to hold the intermediate results. Then, the intermediate result ($C = C_L + (C_L + C_H) \cdot 2^{112} +, C_H \cdot 2^{224}$) is obtained. Particularly, the square of the absolute difference ($|A \bmod 2^{112} - A \operatorname{div} 2^{112}| \cdot |A \bmod 2^{112} - A \operatorname{div} 2^{112}|$) is always positive. Thus, the two's complement need not be obtained for the result ($C = C + C_M \cdot 2^{112}$).

For 448-bit squaring, two-level subtractive Karatsuba is performed based on 224-bit one-level Karatsuba squaring. First, the lower partial product ($C_L = A \bmod 2^{224} \times A \bmod 2^{224}$) and the higher partial product ($C_H = A \operatorname{div} 2^{224} \times A \operatorname{div} 2^{224}$) are computed. Then, the intermediate result ($C = C_L + (C_L + C_H) \cdot 2^{224} + C_H \cdot 2^{448}$) is obtained. The square of the absolute difference ($|A \bmod 2^{224} - A \operatorname{div} 2^{224}| \cdot |A \bmod 2^{224} - A \operatorname{div} 2^{224}|$) is obtained and added to the results ($C = C + C_M \cdot 2^{224}$).

To perform multiple-level Karatsuba for 448-bit integer squaring, the length of the starting partial product is set to 112 bits (112 → 224 → 448). 112-bit squaring is efficiently implemented using the MAC and SBD methods.

For 224-bit squaring, one-level Karatsuba is performed based on the 112-bit SBD. First, the lower partial product ($C_L = A \bmod 2^{112} \times B \bmod 2^{112}$) and the higher partial product ($C_H = A \operatorname{div} 2^{112} \times B \operatorname{div} 2^{112}$) are computed. Then, the intermediate result ($C = C_L + (C_L + C_H) \cdot 2^{112} + C_H \cdot 2^{224}$) is obtained. The square of the absolute difference ($|A \bmod 2^{112} - A \operatorname{div} 2^{112}| \cdot |A \bmod 2^{112} - A \operatorname{div} 2^{112}|$) is always positive, and the two's complement need not obtained for the result ($C = C + C_M \cdot 2^{112}$). Similarly, 448-bit squaring is performed using two-level Karatsuba (two 224-bit one-level Karatsuba algorithms).

## 4 | OPTIMIZATION TECHNIQUES FOR CURVE ED448

ECC implementations require finite field and group operations. For the finite field operations, modular addition, subtraction, multiplication, squaring, and inversion operations are required.

### 4.1 | Finite field operations

#### 4.1.1 | Finite field addition/subtraction

The 448-bit addition and subtraction operations are performed together with modular reduction for finite field addition and subtraction for the Ed448 curve. Modular reduction is performed when addition or subtraction generate carry or borrow bits. To avoid information leakage from branch statements, masked modular reduction is used. When the carry or borrow bit is set, a mask value is generated. For example, if the borrow bit is set, the mask value is `0xff` or `0xffff` for 8-bit AVR and 16-bit MSP processors, respectively. Subsequently, the modulus is masked accordingly. Then, the masked modulus is reduced in the intermediate results, thus achieving constant timing.

#### 4.1.2 | Finite field multiplication / squaring

448-bit multiplication or squaring outputs 896-bit results, which should be reduced to 448-bit form. As in finite field addition or subtraction, subtraction or addition can be performed

**Algorithm 1** Fast reduction Ed448

**Require:** 896-bit intermediate result A (A[3]∼A[0] in 224-bit form)
**Ensure:** 448-bit result C (C[1]‖C[0] in 224-bit form)

1: $\varepsilon 0 \| T \leftarrow A[2]+A[3]$
2: $\varepsilon 1 \| C[0] \leftarrow A[0]+\varepsilon 0 \| T$
3: $\varepsilon 2 \| C[1] \leftarrow A[1]+A[3]+\varepsilon 0 \| T$
4: $\varepsilon 3 \| C[0] \leftarrow C[0]+\varepsilon 2$
5: $\varepsilon 4 \| C[1] \leftarrow C[1]+(\varepsilon 1+\varepsilon 2+\varepsilon 3)$
6: $\varepsilon 5 \| C[0] \leftarrow C[0]+\varepsilon 4$
7: $C[1] \leftarrow C[1]+(\varepsilon 4+\varepsilon 5)$
**return** C

to obtain a 448-bit form, but this is not efficient; rather, the fast reduction technique is chosen. As the modulus of the Ed448 curve is static, the most efficient reduction step can be predetermined with a combination of addition and subtraction operations. Fast reduction for Ed448 is described in detail in Algorithm 1. In Step 1, the parts of the intermediate results (A[2]: 672 ∼ 449-bit, A[3]: 896 ∼ 673-bit) are added to obtain the output of T and the carry bit ($\varepsilon 0$). In Step 2, the lowest intermediate result (A[0]: 224∼1-bit) and previous results ($\varepsilon 0 \| T$) are added, and their sum is the output ($\varepsilon 1 \| C[0]$). In Step 3, the parts of the intermediate results (A[1]: 448 ∼ 225-bit, A[3]: 896 ∼ 673-bit) results ($\varepsilon 0 \| T$) are added, and their sum is the output ($\varepsilon 2 \| C[1]$). In Step 4, the carry bit ($\varepsilon 2$) is added to the result (C[0]), and the sum the output ($\varepsilon 3 \| C[0]$). In Step 5, the carry bits ($\varepsilon 1$, $\varepsilon 2$, $\varepsilon 3$) are added to the result (C[1]), and the sum is the output ($\varepsilon 4 \| C[1]$). In Step 6, the carry bit ($\varepsilon 4$) is added to the result (C[0]), and the sum is the output ($\varepsilon 5 \| C[0]$). In Step 7, the carry bits ($\varepsilon 4$, $\varepsilon 5$) are added to the result (C[1]), and the sum is the output (C[1]). Finally, the result (C) is returned.

In the last step of scalar multiplication, exact modular reduction is performed, as in some cases, carry bits are not generated, but the results are larger than the modulus.

### 4.1.3 | Finite field inversion

Constant time inversion for the Ed448 curve is used based on Fermat's Theorem. To compute the inversion modulo, the prime $p = 2^{448} - 2^{224} - 1$ can be realized using only 447 modular squaring and 13 modular multiplication operations. One can compute $a = z^{-1} \equiv z^{2^{448}-2^{224}-3} \bmod p$. The detailed descriptions are given in Algorithm 2. The alternative extended Euclidean algorithm is not considered here because it is vulnerable to simple power analysis and timing attacks.

**Algorithm 2** Fermat-based inversion for Ed448 ($p = 2^{448} - 2^{224} - 1$)

**Require:** Integer $z$ satisfying $1 \le z \le p-1$.
**Ensure:** Inverse $t_7 = z^{p-2} \bmod p = z^{-1} \bmod p$.

1: $z_3 \leftarrow z^{2^1} \cdot z$ ▷ cost: 1S+1M
2: $t_0 \leftarrow z_3^{2^2} \cdot z_3$ ▷ cost: 2S+1M
3: $t_1 \leftarrow t_0^{2^1} \cdot z$ ▷ cost: 1S+1M
4: $t_2 \leftarrow t_1^{2^4} \cdot t_0$ ▷ cost: 4S+1M
5: $t_3 \leftarrow t_2^{2^9} \cdot t_2$ ▷ cost: 9S+1M
6: $t_4 \leftarrow (t_3^{2^{18}} \cdot t_3)^2 \cdot z$ ▷ cost: 19S+2M
7: $t_5 \leftarrow (t_4^{2^{37}} \cdot t_4)^{2^{37}} \cdot t_4$ ▷ cost: 74S+2M
8: $t_6 \leftarrow t_5^{2^{111}} \cdot t_5$ ▷ cost: 111S+1M
9: $t_7 \leftarrow (t_6^{2^1} \cdot z^{2^{223}} \cdot t_6)^{2^2} \cdot z$ ▷ cost: 226S+3M
**return** $t_7$

## 4.2 | Group operations

Ed448 requires point addition and doubling operations. To ensure constant timing, the Montgomery ladder algorithm [24] is used. For each bit, one-point addition and one-point doubling are performed for two points $P1(x1, y1, z1, e1, h1)$ in extended projective coordinates and $P2(u2, v2, w2)$ in extended affine coordinates. This outputs the point $P3(x3, y3, z3, e3, h3)$ in extended projective coordinates. The detailed procedure of point addition is given in Algorithm 3.

Point doubling performs doubling of a point $P1(x1, y1, z1, e1, h1)$ in extended projective coordinates. This outputs the point $P3$ $(x3, y3, z3, e3, h3)$ in extended projective coordinates. The detailed procedure of point doubling is given in Algorithm 4.

**Algorithm 3** Point addition

**Require:** Point $P1 = (x1, y1, z1, e1, h1)$ in extended projective coordinates, Point $P2 = (u2, v2, w2)$ in extended affine coordinates
**Ensure:** $P3 = (x3, y3, z3, e3, h3)$ in extended projective coordinates

1: $t1 \leftarrow e1 \cdot h1$
2: $e3 \leftarrow y1 - x1$
3: $h3 \leftarrow y1 + x1$
4: $x3 \leftarrow e3 \cdot v2$ ▷ $A = (y1 - x1) \cdot (y2 - x2)$
5: $y3 \leftarrow h3 \cdot u2$ ▷ $B = (y1 + x1) \cdot (y2 + x2)$
6: $e3 \leftarrow y3 - x3$ ▷ $E = B - A$
7: $h3 \leftarrow y3 + x3$ ▷ $H = B + A$
8: $x3 \leftarrow t1 \cdot w2$ ▷ $C = t1 \cdot w2$
9: $t1 \leftarrow z1 - x3$ ▷ $F = z1 - C$
10: $x3 \leftarrow z1 + x3$ ▷ $G = z1 + C$
11: $z3 \leftarrow t1 \cdot x3$ ▷ $Z3 = F \cdot G$
12: $y3 \leftarrow x3 \cdot h3$ ▷ $Y3 = G \cdot H$
13: $x3 \leftarrow e3 \cdot t1$ ▷ $X3 = E \cdot F$
**return** $P3(x3, y3, z3, e3, h3)$

**Algorithm 4** Point doubling

**Require:** Point $P1 = (x1, y1, z1, e1, h1)$ in extended projective coordinates

**Ensure:** $P3 = (x3, y3, z3, e3, h3)$ in extended projective coordinates

1: $e3 \leftarrow x1 \cdot x1$ ▷ $A = x1 \cdot x1$
2: $h3 \leftarrow y1 \cdot y1$ ▷ $B = y1 \cdot y1$
3: $t1 \leftarrow e3 - h3$ ▷ $G = A - B$
4: $h3 \leftarrow e3 + h3$ ▷ $H = A + B$
5: $x3 \leftarrow x1 + y1$
6: $e3 \leftarrow x3 \cdot x3$
7: $e3 \leftarrow h3 - e3$ ▷ $E = H - (x1 + y1) \cdot (x1 + y1)$
8: $y3 \leftarrow z1 \cdot z1$
9: $y3 \leftarrow 2 \cdot y3$ ▷ $C := 2 \cdot z1 \cdot z1$
10: $y3 \leftarrow t1 + y3$ ▷ $F := G + C$
11: $x3 \leftarrow e3 \cdot y3$ ▷ $X3 := E \cdot F$
12: $z3 \leftarrow y3 \cdot t1$ ▷ $Z3 := F \cdot G$
13: $y3 \leftarrow t1 \cdot h3$ ▷ $Y3 := G \cdot H$
**return** $P3(x3, y3, z3, e3, h3)$

## 5 | EVALUATION

In this section, the proposed implementations of 448-bit multi-precision multiplication and squaring on 8-bit AVR processors are evaluated, prior to the presentation of the performance enhancement of Ed448 operations.

**TABLE 4** Speed (in cycles) and size (in bytes) comparison of multi-precision multiplication on 16-bit MSP430; all counts include function-call overhead

| Implementation | | 224-bit | 256-bit | 448-bit |
|---|---|---|---|---|
| Multi-precision multiplication | | | | |
| [15] | Cycles | – | 3597 | – |
| | Bytes | – | N/A | – |
| [25] | Cycles | – | 3554[1] | – |
| | Bytes | – | 4270 | – |
| This study | Cycles | 2870[1] | – | 10 709[2] |
| | Bytes | 2712 | – | 3908 |
| Multi-precision squaring | | | | |
| [15] | Cycles | – | 2960 | – |
| | Bytes | – | N/A | – |
| This study | Cycles | 2109[1] | – | 7868[2] |
| | Bytes | 2026 | – | 3222 |

[1]1-level of Karatsuba.
[2]2-level of Karatsuba.

In the 8-bit AVR benchmarks, the focus is on the ATxmega128A1 model, which is widely used in IoT devices. This microcontroller unit (MCU) features 8 KB of SRAM and 128 KB of flash memory, and operates at 32 MHz. The implementation was written using a mixture of ANSI C and Assembly. In particular, the main

**TABLE 3** Speed (in cycles) and size (in bytes) comparison of multi-precision multiplication on 8-bit AVR processors; all counts exclude function-call overhead

| Implementation | | 56-bit | 64-bit | 112-bit | 128-bit | 224-bit | 256-bit | 448-bit |
|---|---|---|---|---|---|---|---|---|
| Multi-precision multiplication | | | | | | | | |
| [19] | Cycles | – | – | – | 1523 | – | 6115 | – |
| | Bytes | – | – | – | 2346 | – | N/A | – |
| [5] | Cycles | – | – | – | – | – | 6121 | – |
| | Bytes | – | – | – | – | – | 9476 | – |
| [21] | Cycles | – | 366[1] | – | 1361[2] | – | 4936[3] | – |
| | Bytes | – | 572 | – | 2140 | – | 7564 | – |
| This study | Cycles | 315 | – | 1192[1] | – | 4481[2] | – | 15 238[3] |
| | Bytes | 474 | – | 1780 | – | 6506 | – | 21 780 |
| Multi-precision squaring | | | | | | | | |
| [23] | Cycles | – | – | – | 1003 | – | – | – |
| | Bytes | – | – | – | N/A | – | – | – |
| [20] | Cycles | – | – | – | 872[2] | – | 3324[3] | – |
| | Bytes | – | – | – | N/A | – | N/A | – |
| This study | Cycles | 201 | – | 691 | – | 2619[1] | – | 9308[2] |
| | Bytes | 304 | – | 1088 | – | 3968 | – | 13 716 |

[1]1-level of Karatsuba.
[2]2-level of Karatsuba.
[3]3-level of Karatsuba.

**TABLE 5** Speed evaluation (in cycles) of finite field operation and group operation on 8-bit AVR and 16-bit MSP processors

| Finite field operation | | | | | Group Op | |
|---|---|---|---|---|---|---|
| **Target** | **Addition** | **Subtraction** | **Multiplication** | **Squaring** | **Inversion** | **Scalar Mul** |
| AVR | 737 | 735 | 16 709 | 10 774 | 5 044 229 | 103 228 541 |
| MSP | 570 | 569 | 11 401 | 8557 | 3 977 400 | 73 477 660 |

**TABLE 6** Speed evaluation (in cycles) of scalar multiplication on 8-bit AVR and 16-bit MSP processors

| | | 128-bit security | | 224-bit security |
|---|---|---|---|---|
| **Target** | **Implementation** | **NIST P-256** | **Curve25519** | **Ed448** |
| AVR | [26] | 34 930 000 | – | – |
| | [20] | – | 13 900 397 | – |
| | This work | – | – | 103 228 541 |
| MSP | [26] | 22 170 000 | – | – |
| | [15] | 20 476 234 | – | – |
| | This work | – | – | 73 477 660 |

structure of the ECC scheme and the interface were written in C, whereas the modular operations were implemented in Assembly. The implementation was compiled using the speed optimization option -O3 on Atmel Studio 6.2. The results of multi-precision multiplication and squaring on 8-bit AVR are given in Table 3. For 56-bit multiplication, ordinary multi-precision multiplication was performed. This is 13.9% faster than 64-bit multiplication by Hutter-Schwabe [21]. For 112-bit multiplication, one-level Karatsuba was used. This is 12.4% faster than 128-bit multiplication by Hutter-Schwabe [21]. For 224-bit multiplication, two-level Karatsuba was used. This is 9.2% faster than 256-bit multiplication by Hutter-Schwabe [21]. Finally, for 448-bit multiplication, three-level Karatsuba was used, achieving 15 238 clock cycles. For the short squaring operations, SBD was used, achieving 201 and 691 clock cycles for 56-bit and 112-bit implementations, respectively. For 224-bit multiplication, one-level Karatsuba was used, which is 21.2% faster than 256-bit multiplication by Düll et al [20]. Finally, for 448-bit squaring, two-level Karatsuba was used, achieving 9 308 clock cycles.

For the 16-bit MSP430 benchmarks, the MSP430F1611 model was chosen, which is widely used in wireless sensor nodes such as Tmote Sky and TelosB. This MCU features 10 KB of SRAM and 48 KB of flash memory, and operates at 8 MHz.[1] The same methodology for cycle count acquisition was adopted as for the case of AVR, using IAR Embedded Workbench (MSP430 6.50.1). The comparison results for multi-precision arithmetic are presented in Table 4.

[1]MSP430 supports various speed options, and the CC430 and FRAM series support 20 and 24 MHz, respectively.

Hinterwälder et al performed one-level Karatsuba and obtained 3 554 clock cycles [25]. The proposed 448-bit multiplication used two-level Karatsuba and achieved 10 709 clock cycles with smaller code sizes. For squaring, Gouvêa et al. used product scanning [15]. However, Karatsuba was used here for 224-bit squaring, and 448-bit squaring was further improved by two-level Karatsuba.

The evaluation results for finite field and group operations are presented in Table 5. On the 8-bit AVR processor, finite field addition and subtraction require 737 and 735 clock cycles, respectively. Finite field multiplication and squaring require 16 709 and 10 774 clock cycles, respectively. Both results include multiplication and squaring with fast reduction. Finite field inversion is implemented in constant timing using Fermat's Theorem. The clock cycles are the sum of the costs of 447 modular squarings and 13 modular multiplications. Finally, the group operation is evaluated. It requires 103 M clock cycles, which include 448 point additions, 448 point doublings, and finite field inversion operations.

The comparison results for scalar multiplication on 8-bit AVR processors are presented in Table 6. The NIST curve (P256) requires 34 930 000 clock cycles, and the Montgomery curve (Curve25519) requires 13 900 397 clock cycles. Both curves achieve 128-bit security level. The target Edward curve Ed448 has a security level of 224 bits, which requires 103 228 541 clock cycles. Scalar multiplication requires 3.2 s on a 32 MHz AVR processor. For the MSP430 processor, ECC with 128-bit security level, 20 M–22 M clock cycles are required [15,26]. The implementation of ECC with 224-bit security level requires 73 M clock cycles, which implies that 9 s are required for scalar multiplication on 8 MHz 16-bit MSP processors. This is insufficient for real-world

applications. However, modern MSP processors provide high CPU frequency (eg, 24 MHz), which can reduce execution timing significantly. The implementations have constant timing, which is secure against simple power analysis and timing attacks.

## 6 | CONCLUSION

IoT will play an increasingly important role, particularly as hardware design becomes more advanced and sophisticated (eg, longer battery life with more computational power), and a particular research challenge is to implement high-speed cryptography on low-end IoT things.

In this study, a high-speed implementation of ECC was proposed by optimizing the Curve Ed448 on 8-bit AVR and 16-bit MSP processors (ie, two low-end IoT things). Specifically, the subtractive Karatsuba approach was first optimized for efficient implementation of 448-bit multi-precision multiplication and squaring on 8-bit AVR and 16-bit MSP microcontrollers. The 448-bit three-level Karatsuba multiplication and the two-level Karatsuba squaring could be accomplished within 15 238, and 21 780 clock cycles. This is currently a new speed record on an 8-bit AVR processor. Similarly, 10 709 and 7868 clock cycles were achieved for MSP430 processors. Then, Ed448 was proposed, which combines the computational advantages of Edwards curves. Finally, based on the proposed efficiently implemented cryptographic library, record-setting execution times were achieved for scalar multiplication over 448-bit security prime fields. For example, scalar multiplication using Ed448 requires 103 M and 73 M cycles. This is also the first implementation of Curve Ed448 on 8-bit and 16-bit IoT platforms.

Future research may include deployment in a real-world environment, for example, in a smart grid in collaboration with a local utility provider. This will allow a more comprehensive evaluation and would possibly identify further opportunities for optimizing the implementations.

## REFERENCES


1. L. O. Wallin and T. Zimmerman, *Strategic Roadmap for IoT Network Technology*, 2017, available at: https://www.gartner.com/doc/3587517/-strategic-roadmap-iot-network.
2. N. Gura et al., *Comparing elliptic curve cryptography and RSA on 8-bit CPUs*, in Int. Workshop Cryptographic Hardw. Embedded Syst., Cambridge, MA, USA, Aug. 2004, pp. 119–132.
3. D. J. Bernstein, *Curve25519: new Diffie-Hellman speed records*, in Int. workshop Public Key Cryptography, New York, USA, Apr. 2006, pp. 207–228.
4. M. Hamburg, *Ed448-Goldilocks, a new elliptic curve*, Cryptology ePrint Archive: Report 2015/625, 2015.
5. M. Hutter and E. Wenger, *Fast multi-precision multiplication for public-key cryptography on embedded microprocessors*, in Int. Workwhop Crytographic Hardw. Embedded Syst., Nara, Japan, 2011, pp. 459–474.
6. Z. Liu et al., *Reverse product-scanning multiplication and squaring on 8-bit AVR processors*, in Int. Conf. Inform. Commun. Security, Hong Kong, China, Dec. 2014, pp. 158–175.
7. Z. Liu, E. Wenger, and J. Großschädl, *MoTE-ECC: Energy-scalable elliptic curve cryptography for wireless sensor networks*, in Int. Conf. Appl. Cryptography Netw. Security., Lausanne, Switzerland, June 2014, pp. 361–379.
8. E. Rescorla et al., *The Transport Layer Security (TLS) Protocol Version* 1.3., 2017, available at https://tools.ietf.org/html/draft-ietf-tls-tls13-21.
9. H. Edwards, *A normal form for elliptic curves*, Bull. Amer. Math. Soc. **44** (2007), no. 3, 393–422.
10. D. J. Bernstein et al., *High-speed high-security signatures*. J. Crypto. Eng. **2** (2012), no. 2, 77–89.
11. Z. Liu et al., *On emerging family of elliptic curves to secure internet of things: ECC comes of age*, IEEE Trans. Dependable Secure Comput. **14** (2017), no. 3, 237–248.
12. D. J. Bernstein et al., *SafeCurves: choosing safe curves for elliptic-curve cryptography*, 2013, available at: http://safecurves.cr.yp.to.
13. A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, *Handbook of applied cryptography*, CRC press, Boca Raton, FL, USA, 1996.
14. P. G. Comba, *Exponentiation cryptosystems on the IBM PC*, IBM Syst. J. **29** (1990), no. 2, 526–538.
15. P. L. Gouvêa and J. López, *Software implementation of pairing-based cryptography on sensor networks using the MSP430 microcontroller*, in Int. Conf. Cryptology India New Delhi, India, Dec. 2009, pp. 248–262.
16. Z. Liu et al., *Efficient implementation of ECDH key exchange for MSP430-based wireless sensor networks*, in Proc. ACM Symp. Inform., Comput. Commun. Security, Singapore, 2015, pp. 145–153.
17. L. Qiu et al., *Implementing RSA for sensor nodes in smart cities*, Pers. Ubiquit. Comput. **21** (2017), no. 5, 807–813.
18. H. Seo and H. Kim, *Multi-precision multiplication for public-key cryptography on embedded microprocessors*, in Int. Workshop Inform. Security Applicat., Nara, Japan, 2012, pp. 55–67.
19. H. Seo and H. Kim, *Optimized multi-precision multiplication for public-key cryptography on embedded microprocessors*, Inter. J. Comput. Commun. Eng. **2** (2013), no. 3, 255–259.
20. M. Düll et al., *High-speed Curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers*, Des. Codes Crypt. **77** (2015), no. 2–3, 493–514.
21. M. Hutter and P. Schwabe, *Multiprecision multiplication on AVR revisited*, J. Crypto. Eng. **5** (2015), no. 3, 201–214.
22. H. Seo et al., *Hybrid Montgomery reduction*, ACM Trans. Emb. Comput. Syst. **15** (2016), no. 3, Article no. 58.
23. H. Seo et al., *Multi-precision squaring for public-key cryptography on embedded microprocessors*, in Int. Conf. Cryptology in India, Munbai, India, 2013, pp. 227–243.
24. P. L. Montgomery, *Speeding the Pollard and elliptic curve methods of factorization*, Math. Comp. **48** (1987), no. 177, 243–264.

25. G. Hinterwälder et al., *Full-size high-security ECC implementation on MSP430 microcontrollers*, in Int. Conf. Cryptology Inform. Security Latin America, Florianopolis, Brazil, 2014, pp. 31–47.
26. E. Wenger, T. Unterluggauer, and M. Werner, *8/16/32 shades of elliptic curve cryptography on embedded processors*. in Int. Conf. Cryptology India, Mumbai, India, 2013, pp. 244–261.

## AUTHOR BIOGRAPHY

**Hwajeong Seo** received the BSEE, MS, and PhD degrees in Computer Engineering from Pusan National University, Busan, Rep. of Korea. He is currently an assistant professor at Hansung University, Seoul, Rep. of Korea. His research interests include Internet of Things and information security.