Efficient Implementation of NIST-Compliant Elliptic Curve Cryptography for Sensor Nodes

Zhe Liu[1], Hwajeong Seo[2], Johann Großschädl[1], and Howon Kim[2]

[1] University of Luxembourg,
Laboratory of Algorithmics, Cryptology and Security (LACS),
6, rue R. Coudenhove-Kalergi, 1359 Luxembourg-Kirchberg, Luxembourg
{zhe.liu,johann.groszschaedl}@uni.lu
[2] Pusan National University,
School of Computer Science and Engineering,
San-30, Jangjeon-Dong, Geumjeong-Gu, Busan 609–735, Republic of Korea
{hwajeong,howonkim}@pusan.ac.kr

Abstract. In this paper, we present a highly-optimized implementation of standards-compliant Elliptic Curve Cryptography (ECC) for wireless sensor nodes and similar devices featuring an 8-bit AVR processor. The field arithmetic is written in Assembly language and optimized for the 192-bit NIST-specified prime $p = 2^{192} - 2^{64} - 1$, while the group arithmetic (i.e. point addition and doubling) is programmed in ANSI C. One of our contributions is a novel *lazy doubling* method for multi-precision squaring which provides better performance than any of the previously-proposed squaring techniques. Based on our highly optimized arithmetic library for the 192-bit NIST prime, we achieve record-setting execution times for scalar multiplication (with both fixed and arbitrary points) as well as multiple scalar multiplication. Experimental results, obtained on an AVR ATmega128 processor, show that the two scalar multiplications of ephemeral Elliptic Curve Diffie-Hellman (ECDH) key exchange can be executed in 1.75 s altogether (at a clock frequency of 7.37 MHz) and consume an energy of some 42 mJ. The generation and verification of an ECDSA signature requires roughly 1.91 s and costs 46 mJ at the same clock frequency. Our results significantly improve the state-of-the-art in ECDH and ECDSA computation on the P-192 curve, outperforming the previous best implementations in the literature by a factor of 1.35 and 2.33, respectively. We also protected the field arithmetic and algorithms for scalar multiplication against side-channel attacks, especially Simple Power Analysis (SPA).

1 Introduction

Wireless Sensor Networks (WSNs) are a key technology of the 21st century, enabling new applications in such domains as infrastructure protection, industrial automation and health monitoring, to name a few [1]. A WSN can be defined as a network composed of autonomous, battery-powered computing devices (called nodes) with sensing and wireless networking capabilities. The sensor nodes are

S. Qing et al. (Eds.): ICICS 2013, LNCS 8233, pp. 302–317, 2013.

deployed in a certain environment or area of interest to monitor a phenomenon or condition such as temperature, humidity, luminosity, etc. They cooperatively collect and aggregate sensor readings and send them to a central unit (the base station) for further processing and decision making. Unfortunately, WSNs face all security threats inherent in any wireless network, plus additional ones that are hard to protect against [24]. Since WSNs are often deployed in unattended areas, an attacker may be able to access individual nodes and perform various kinds of physical attacks, e.g. side-channel cryptanalysis [10]. The integration of countermeasures against such attacks is a nontrivial task due to the resource constraints (in particular limited energy) of battery-powered sensor nodes like the MICAz mote [8]. These constraints make a good case for using lightweight cryptosystems that can be effectively protected against side-channel attacks. In the context of public-key cryptography, elliptic-curve based algorithms such as ECDH and ECDSA are known to meet these requirements [15].

Energy is the most precious resource of a wireless sensor node. The MICAz mote [8], for example, is powered by two 1.5 V AA batteries, which can not be easily recharged or replaced after deployment. In general, the energy consumption of cryptographic software depends primarily on the execution time of the algorithm and the average power dissipation of the processor it is executed on [11]. However, a cryptographic engineer can only influence the former since the choice of the processor is normally not under his control. The overall execution time of most elliptic curve cryptosystems is dominated by the time needed to perform a scalar multiplication, which, in turn, depends on a number of factors such as the order of the elliptic-curve group, the actual implementation of the point arithmetic, and the efficiency of certain operations (e.g. multiplication) in the underlying finite field [15]. Another important aspect is the concrete form of the elliptic curve; for example, Montgomery curves [25] or Twisted Edwards curves [16] allow for more efficient point addition/doubling than conventional Weierstraß curves. Unfortunately, these special curve shapes are not standardized, which prevents their use in commercial applications that need to undergo a certification process. On the other hand, curves specified by standards bodies like the NIST facilitate inter-operability and maximize access to resources and services. Therefore, we decided to adopt the NIST-recommended elliptic curve P-192 from [26] for our implementation.

Elliptic Curve Cryptography (ECC) has been exhaustively researched in the past 25 years and is nowadays considered an excellent option for the implementation of key exchange and digital signatures. Virtually all ECC cryptosystems of practical importance require to execute one (resp. two) of the following three variants of scalar multiplication: (i) $k \cdot P$ where the point P is fixed and known a priori (called *fixed-point scalar multiplication*), (ii) $k \cdot Q$ with Q being an arbitrary point not known in advance, and (iii) $k \cdot P + l \cdot Q$ where P is fixed and Q is an arbitrary point (called *double scalar multiplication*). For example, the classical ECDH key exchange protocol consists of two stages; in the first stage a key-pair is created, which comprises a fixed-point scalar multiplication by the generator of an elliptic-curve group of prime order. The second stage involves

a scalar multiplication by a point that, unlike to the first stage, is neither fixed nor known in advance. Something similar holds for ECDSA since the signature-generation process entails a fixed-point scalar multiplication like the first stage of ECDH. However, the verification of an ECDSA signature requires to execute a double scalar multiplication of the form $k \cdot P + l \cdot Q$, whereby one of the two points is not known in advance.

1.1 Overview of Related Work and Motivation for Our Work

In the past ten years, a multitude of ECC implementations for 8-bit processors appeared in the literature. The first milestone belongs to Gura et al [14], who introduced highly-optimized ECC software for 8-bit AVR microcontrollers like the ATmega128 [4] and reported an execution time of roughly 0.81 s and 1.24 s for a 160-bit and 192-bit scalar multiplication, respectively (at a frequency of 8 MHz). They also found that the relative performance advantage of ECC versus RSA increases with larger key sizes (i.e. larger groups). TinyECC [22] was one of the first publicly available and, hence, widely used ECC libraries for wireless sensor nodes. Most parts of TinyECC are implemented in nesC, but it contains also numerous processor-specific optimizations (written in Assembly language) for common 8-bit and 16-bit sensor platforms. It has been tested successfully on MICA2/MICAz, TelosB/Tmote Sky, BSNV3, and the Imote2 node. TinyECC supports the SECG-specified 128-bit and 160-bit domain parameters as well as the NIST curve P-192 through dedicated field and curve arithmetic operations [22]. There exist many other efficient ECC implementations for 8-bit AVR processors, e.g. WM-ECC [33], Nano-ECC [31], MIRACL [6], NaCl [17] using prime fields and RELIC [2] for binary fields.

All currently-existing prime-field based ECC libraries use either the hybrid multiplication technique [14] (or a variant of it [6,22,33,31,20]) or employ the Karatsuba method (e.g. NaCl [17]) for the performance-critical multi-precision multiplication and squaring. Recently, the operand caching method [18] and its successor, the consecutive operand caching method [28], were proposed as new techniques to speed up multi-precision multiplication on embedded micro-controllers, while Lee et al [21] developed several optimizations for multi-precision squaring. However, these recent papers focussed exclusively on multi-precision arithmetic and did not evaluate the impact of the described techniques on the overall execution time of a scalar multiplication. It is, therefore, interesting to combine these sophisticated multiplication and squaring techniques in order to push the envelope of ECC on AVR micro-controllers. However, performance is not our only goal since, as pointed out before, protection against side-channel cryptanalysis (i.e. timing and SPA attacks) is similarly important. Most of the previous ECC libraries, however, do not contain countermeasures; the only two exceptions are the work from [20] and NaCl [17]. Lederer et al [20] implemented ECDH for WSNs and protected the scalar multiplication against SPA attacks by adopting highly "regular" variants of the comb and window method, respectively. Their ECDH software uses a 192-bit prime field specified by the NIST as underlying algebraic structure and needs $5.20 \cdot 10^6$ and $12.33 \cdot 10^6$ clock cycles

an ATmega128 processor to to compute a fixed-point and random-point scalar multiplication, respectively. NaCl is a cryptographic library whose ECC part is based on Curve-25519 [5] and, therefore, provides a (symmetric) security level of about 128 bits. Unfortunately, a scalar multiplication on Curve-25519 needs at least $22.95 \cdot 10^6$ clock cycles when executed on an ATmega128 micro-controller (i.e. 3.11 s at a frequency of 7.37 MHz), which naturally raises the question of how well Curve25519 is suited for battery-powered sensors nodes.

1.2 Our Contributions

We introduce a number of optimizations to improve both the performance and security (i.e. resistance against timing and SPA attacks) of scalar multiplication on the NIST curve P-192 when executed on an 8-bit AVR micro-controller. The contribution of this paper is threefold.

- *New approach for the efficient implementation of multi-precision squaring on 8-bit AVR micro-controllers.* The novel "lazy doubling" method we describe in this paper has been specially devised for multi-precision squaring. When executed on the ATmega128, it needs merely 2,064 clock cycles to square a 192-bit integer, which sets a new speed record for multi-precision squaring on an 8-bit processor.
- *Highly optimized arithmetic library for the NIST P-192 field.* All operations of our library (except inversion) are implemented in a highly regular fashion independent of the value of the operands, which helps to thwart timing and SPA attacks. Yet, our implementation of arithmetic operations modulo the 192-bit NIST prime is more than twice as fast as the widely-used TinyECC library, the current de-facto standard for ECC in WSNs [22].
- *Record-setting execution times for ECDH and ECDSA over a 192-bit prime field.* We employ a regular variant of the fixed-base comb technique for the fixed-point scalar multiplication and a window method with a window size of 4 when the point is not known a priori, while the double scalar multiplication is executed in an interleaved fashion with joint doublings. Practical results, obtained on an ATmega128, demonstrate that our work exceeds the state-of-the-art in ECDH key agreement and ECDSA signature generation (resp. verification), outperforming the best implementations reported in the literature by a factor of 1.35 [20] and 2.33 [22], respectively.

The rest of the paper is organized as follows. In Section 2, we briefly discuss the mathematical foundations of ECC and describe the basic properties of the NIST curve P-192 we adopt in our implementation. Thereafter, we explain the algorithms for fixed-point and arbitrary-point scalar multiplication (for ECDH) as well as double-scalar multiplication (for ECDSA). In Section 3, we introduce our implementation of the field operations for the 192-bit prime, including the new "lazy doubling" method for multi-precision squaring. The implementation results (e.g. execution time, energy consumption, RAM footprint) we achieved are summarized in Section 4. Finally, we conclude the paper in Section 5.

2 Elliptic Curve Cryptography

In this section, we first discuss some implementation aspects of ECC and then present the domain parameters we used in our implementation. Thereafter, we describe algorithms for fixed-point and arbitrary-point scalar multiplication as well as double scalar multiplication.

2.1 NIST Curve P-192

Let $\mathbb{F}_p$ be a prime field. An elliptic curve E over $\mathbb{F}_p$ can be defined through a short Weierstraß equation of the form $y^2 = x^3 + ax + b$, whereby $a, b \in \mathbb{F}_p$ and $4a^3 + 27b^2 \neq 0$. In order to improve efficiency, it is common practice to fix the curve parameter a to -3 (i.e. $a = p - 3$) since this choice allows for optimizing the point arithmetic, as will be discussed in more detail below. All prime-field curves standardized by the NIST in [26] adopt this approach; consequently, the so-called "NIST curves" can be defined via a short Weierstraß equation of the following form

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

Before an elliptic curve cryptosystem can actually be carried out, the involved parties need to agree on a set of domain parameters, which specifies besides the curve and field to be used also a base point $G = (x_G, y_G)$ that generates a large cyclic subgroup of $E(\mathbb{F}_p)$, the order n of this subgroup (which is a prime), and the co-factor $h = \#E(\mathbb{F}_p)/n$ [15]. All five NIST curves over prime fields have a co-factor of $h = 1$; consequently, any point P whose x and y coordinates fulfill Equation 1 has prime order n. This property prevents small subgroup attacks and, therefore, simplifies the implementation of ECDH key agreement. On the other hand, Edwards curves and Montgomery curves require specific measures to thwart these attacks since they always have a co-factor of $h \geq 4$. Among the five prime-field curves specified in [26], the curve P-192 is the most suitable one for resource-constrained sensor nodes as it offers a reasonable balance between security and execution time (i.e. energy consumption). This curve uses the field $\mathbb{F}_p$ defined by the generalized-Mersenne (GM) prime

$$p = 2^{192} - 2^{64} - 1 \tag{2}$$

as underlying algebraic structure to facilitate the modular reduction. As shown in [15], the product of two 192-bit integers can be reduced via three additions modulo p by exploiting the relation $2^{192} \equiv 2^{64} + 1 \bmod p$. The parameter b, the base point G, and the order n of curve P-192 can be found in [26].

In order to avoid expensive inversions in $\mathbb{F}_p$, we represent the points on the curve using projective coordinates. According to [15, Table 3.3], Jacobian projective coordinates yield the most efficient formula for point doubling, whereas mixed Jacobian-affine coordinates allows for the fastest point addition on curve P-192. Based on [15, Algorithm 3.22], a mixed addition needs 8 multiplications (8M) and 3 squarings (3S) in the underlying field. The doubling of a point costs only 4 multiplications and the same number of squarings (i.e. 4M + 4S).

2.2 Algorithms for Scalar Multiplication

The Elliptic Curve Diffie-Hellman (ECDH) key exchange technique has much in common with the "classical" Diffie-Hellman scheme, but operates in an elliptic curve group $E(\mathbb{F}_p)$ instead of $\mathbb{Z}_p^*$ [15]. There exist two principal variants of the ECDH protocol, namely static ECDH and ephemeral ECDH. The latter is computationally more demanding, but provides the important advantage of forward secrecy. Ephemeral ECDH requires each of the two involved parties to perform two scalar multiplications; the first to generate an ephemeral key pair, and the second to obtain the shared secret. The first scalar multiplication takes a fixed and a-priori-known point as input, namely the generator G, whereas the second scalar multiplication has to be carried out with an arbitrary point not known in advance. Consequently, ephemeral ECDH key agreement requires each party to execute both a fixed-point and an arbitrary-point scalar multiplication.

A fixed-point scalar multiplication can be efficiently performed through the so-called *fixed-base comb method* as described in Section 3.3.2 of [15]. The idea is to pre-compute and store 2^w multiples of the base point P and then process w bits of the scalar k at once, thereby reducing the number of point doublings by a factor of w and the number of point additions by roughly $w/2$ compared to the straightforward double-and-add method. A window size of $w = 4$ represents a good trade-off between performance and storage requirements since only 16 points need to be pre-computed. In this case, a fixed-point scalar multiplication on curve P-192 requires 48 point doublings and up to 48 point additions. The multiples of the base point to be pre-computed are linear combinations of the form $d_3 \cdot (2^{144}P) + d_2 \cdot (2^{96}P) + d_1 \cdot (2^{48}P) + d_0 \cdot P$ with $d_i \in \{0, 1\}$. As indicated before, our comb method represents (and processes) a 192-bit scalar k in 4-bit digits $K_i = 8k_{144+i} + 4k_{96+i} + 2k_{48+1} + k_i$ for $0 \leq i < 48$ (see [15] for an in-depth description of the fixed-base comb method).

A straightforward implementation of the comb method described above has an irregular execution pattern (and, hence, succumbs to Simple Power Analysis (SPA) attacks [19,15]) since the point addition is only carried out for non-zero digits K_i. Consequently, an attacker able to distinguish point additions from point doublings in the power consumption profile can get the position of 0 bits in the scalar k. One possibility to prevent this SPA leakage is to represent the 4-bit digits using a digit set not containing 0, e.g. $D' = \{\pm 1, \pm 3, \ldots, \pm 15\}$, instead of the ordinary set $D = \{0, 1, \ldots, 15\}$ and adapting the pre-computation of multiples of P accordingly. When doing so, the comb technique executes the same number of point additions and point doublings, independent of the actual value of k, since all K_i are non-zero. Liu et al introduced in [23] a simple (and highly regular) algorithm for the conversion of radix-2^4 integers represented via the canonical digit set D into an equivalent representation based on the zero-free digit set D'. We apply their algorithm to obtain the $K_i \in \{\pm 1, \pm 3, \ldots, \pm 15\}$ in an SPA-resistant fashion. Using D' instead of D also allows one to reduce the storage requirements of the comb method by half since we need to pre-compute only the multiples of P corresponding to the eight positive elements of D'. The negative multiples can be generated "on the fly" via the regular point-negation

Algorithm 1. Regular window method for scalar multiplication ($w = 4$)

Input: n-bit scalar $k = (k_{n-1}, \ldots, k_1, k_0)_2$, point $P \in E(\mathbb{F}_p)$.
Output: Scalar product $R = k \cdot P$.

1: Convert k into radix-2^4 representation $k' = (K_{s-1}, \ldots, K_1, K_0)_{16}$ where $s = \lceil n/4 \rceil$ and $K_i \in \{\pm 1, \pm 3, \ldots, \pm 15\}$ for $0 \leq i \leq s-1$ as described in [23].
2: Generate look-up table T consisting of 8 points $T[i] = (2i+1) \cdot P$ for $0 \leq i \leq 7$.
3: $R \leftarrow T[(K_{s-1} - 1)/2]$ {K_{s-1} is always positive}
4: **for** i from $s-2$ by 1 down to 0 **do**
5: $R \leftarrow 16 \cdot R$ {four point doublings}
6: $R \leftarrow R + \text{sign}(K_i) \cdot T[(|K_i| - 1)/2]$ {one point addition}
7: **end for**
8: **return** R

technique described in [23]. In this way, the 4-bit comb method requires a mere 384 bytes in read-only memory (i.e. ROM or Flash) as the pre-computed points are stored in affine coordinates so that we can use the efficient mixed-addition formula given in [15, Algorithm 3.22]

Besides a fixed-point scalar multiplication, each of the two parties involved in an ECDH key agreement also has to perform a scalar multiplication with an arbitrary base point not known in advance. Unfortunately, the comb method involves an expensive pre-computation phase and is, therefore, only useful when the base point is fixed. If this is not the case, it is generally more efficient to employ a *window method* [15] such as shown in Algorithm 1 for a window size of $w = 4$. First, we convert the scalar k into a radix-2^4 representation based on the signed digit set $D' = \{\pm 1, \pm 3, \ldots, \pm 15\}$. Similar to the comb method, we follow the approach introduced in [23] to ensure this conversion does not leak any SPA-relevant information. The next step is then the generation of a table containing eight multiples of P, namely P, $3P$, $5P$, ..., $15P$. Since all of these points are needed in affine coordinates, it makes sense to do a simultaneous inversion [15, page 44] of the Z coordinates so as to reduce the cost of the table computation. The loop in Algorithm 1 is similar to that of the double-and-add method, but we process a 4-bit digit K_i of k in each iteration instead of just a single bit. At line 6, the pre-computed point from table T corresponding to the absolute value of K_i is loaded from RAM. Even though the index for this table access depends on the secret scalar, there is no information leakage since load operations always have the same latency on an ATmega128. Depending on the sign of K_i, the loaded point is added to R either directly or negated. We again refer to [23] for a description of how this can be performed in a regular fashion without the need to execute conditional statements. The window method with $w = 4$ performs 192 point doublings and 48 point additions, idenpendent of the actual value of k. It occupies 384 byte in RAM for table T.

The Elliptic Curve Digital Signature Algorithm (ECDSA) is a variant of the DSA signature scheme operating in an elliptic curve group [15]. From an arithmetic point of view, the major operation of an ECDSA signature generation is a scalar multiplication by a fixed and a-priori-known base point, similar to the

Algorithm 2. Double scalar multiplication with joint doublings

Input: Two n-bit scalars k and l, two points $P, Q \in E(\mathbb{F}_p)$.
Output: Scalar product $R = k \cdot P + l \cdot Q$.

1: $(k', l') \leftarrow \mathrm{JSF}(k, l)$ {calculate JSF of (k, l) using [15, Algorithm 3.50]}
2: $R \leftarrow \mathcal{O},\ S \leftarrow P + Q,\ T \leftarrow P - Q$
3: **for** i from n by 1 down to 0 **do**
4: $R \leftarrow 2R$
5: **if** $(k'_i = 1)$ and $(l'_i = 1)$ **then** $R \leftarrow R + S$
6: **else if** $(k'_i = 1)$ and $(l'_i = -1)$ **then** $R \leftarrow R + T$
7: **else if** $(k'_i = -1)$ and $(l'_i = 1)$ **then** $R \leftarrow R - T$
8: **else if** $(k'_i = -1)$ and $(l'_i = -1)$ **then** $R \leftarrow R - S$
9: **else if** $(k'_i = 1)$ and $(l'_i = 0)$ **then** $R \leftarrow R + P$
10: **else if** $(k'_i = 0)$ and $(l'_i = 1)$ **then** $R \leftarrow R + Q$
11: **else if** $(k'_i = -1)$ and $(l'_i = 0)$ **then** $R \leftarrow R - P$
12: **else if** $(k'_i = 0)$ and $(l'_i = -1)$ **then** $R \leftarrow R - Q$ **end if**
13: **end for**
14: **return** R

first stage of ECDH key exchange. The fixed-base comb method with $w = 4$ is the natural choice to perform this operation in an efficient and secure (i.e. SPA-resistant) fashion. On the other hand, the verification of an ECDSA signature requires a so-called double scalar multiplication of the form $k \cdot P + l \cdot Q$ where one of the points is fixed and the other not. To reduce execution time, the two scalar multiplications $k \cdot P$ and $l \cdot Q$ can be carried out simultaneously (i.e. in an "interleaved" fashion) so that n point doublings suffice to get the result. Algorithm 2 shows a possible realization of this approach, sometimes referred to as *Shamir's Trick* (see e.g. [15, p. 109]). We represent the n-bit scalars k and l in *Joint Sparse Form (JSF)* [30], which means roughly $n/2$ point additions have to be performed. In our case, i.e. 192-bit scalars, the cost of Algorithm 2 amounts to 192 point doublings and some 96 point additions, not taking into account the pre-computation of $P + Q$ and $P - Q$ in line 2. Note that the verification of an ECDSA signature is a public-key operation and, therefore, does not need to be protected against side-channel attacks.

3 Efficient Field Arithmetic for Curve P-192

In the following, we describe our implementation of basic arithmetic operations modulo the 192-bit generalized-Mersenne prime $p = 2^{192} - 2^{64} - 1$.

3.1 Addition and Subtraction

To add two elements $a, b \in \mathbb{F}_p$, we firstly perform a conventional multi-precision addition of the two byte-arrays A and B representing them. As result we get a sum-array S consisting of 24 bytes and a carry bit c, which is either 0 or 1. The carry bit c is then used to generate a mask M that, depending on c, is either an

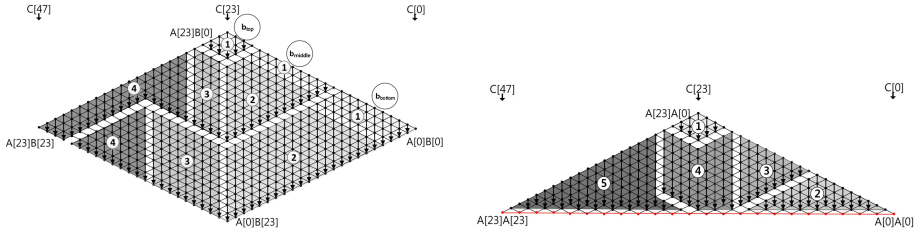

Fig. 1. COC multiplication (left) and "lazy doubling" squaring (right)

"all-1" byte (i.e. has the value 255) or an "all-0" byte. A mask of this form can be simply obtained via negation of the carry bit; we get the "all-1" byte if c is 1 and the "all-0" byte otherwise. Then, we perform two "masked" subtractions of the prime p, which means we do a logical "AND" of prime-byte $P[i]$ and the mask M before we actually subtract it from the corresponding byte of S. Two such subtractions are required to get a result of at most 192 bits, whereby the carry bit c must be updated after the first subtraction. In this way, always the same sequence of instructions is executed, independent of the value of the two operands a and b. Note, however, that the final result may not be fully reduced (even though it is always smaller than 2^{192}), but this is no problem because all functions of our arithmetic library can process incompletely reduced operands [34]. The modular subtraction is implemented in a very similar way.

3.2 Multiplication and Squaring

Multiplication and squaring are two extremely performance-critical arithmetic operations in ECC [2]. Our implementation employs an improved variant of the Consecutive Operand Caching (COC) method [28] for the former and a novel "lazy doubling" technique to speed up the latter. We use the following notation:

- n: operand size (192 bits in our case)
- w: word size of the processor (8 bits)
- m: number of elements in an operand-array, i.e. $m = n/w = 24$
- e: number of operand words (i.e. bytes) to be cached (10 in our case)
- r: number of row sections, $r = \lfloor m/e \rfloor$
- A, B: operands represented by byte arrays: $A = (A[m-1], \ldots, A[1], A[0])$ and $B = (B[m-1], \ldots, B[1], B[0])$
- C: $2n$-bit product $C = A \cdot B$ whereby $C = (C[2m-1], \ldots, C[1], C[0])$

As shown in Figure 1, we describe the execution flow using a rhombus and triangular forms. Each dot represents a byte-product of the form $A[i] \times B[j]$ or $A[i] \times A[j]$. The rightmost corner of the rhombus indicates the lowest indices (i.e. $i, j = 0$), whereas the highest indices (i.e. $i, j = m-1$) can be found at the leftmost corner. All bytes $C[k]$ of the product C are located at the bottom edge of the rhombus, whereby $C[0]$ is at the right and $C[2m-1]$ at the left.

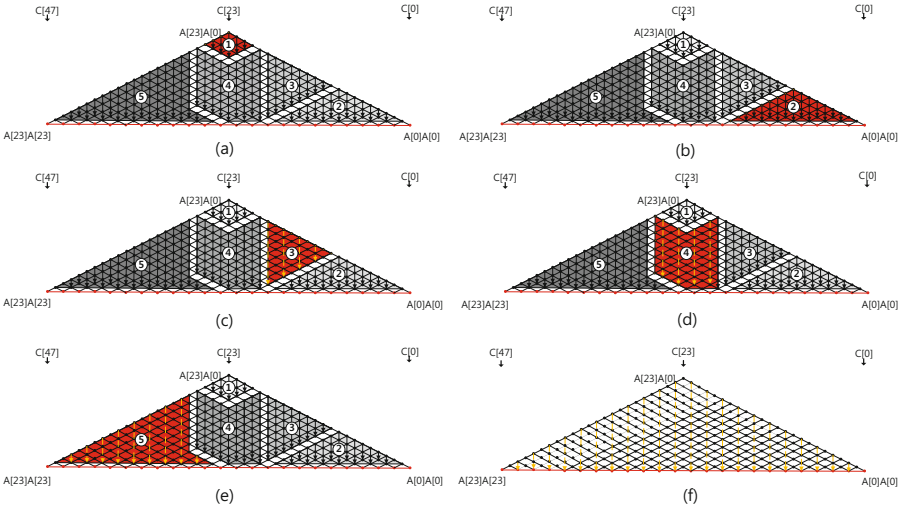

Fig. 2. Execution flow of the novel "lazy doubling" technique

To speed up the multiplication, we combine the COC method [28] with the so-called carry-once technique described in [29]. The COC method reduces the number of load instructions by re-scheduling the byte-multiplication sequences so that each byte is loaded only once (i.e. it fully prevents re-loadings). On the other hand, the carry-once technique minimizes the number of add instructions by first delaying and then updating intermediate results at once instead of doing it one-by-one. The execution flow is illustrated on the left of Figure 1. We have $r = \lfloor \frac{24}{10} \rfloor = 2$, which means there are two row sections, namely b_{bottom} and b_{middle}. First, we calculate the 16 byte-products in block b_{top} at the top of the rhombus; this requires to load the operand bytes $A[20-23]$ and $B[0-3]$. The latter four bytes are cached as they are used again in the next block, which is b_{bottom}. This block consists of four regions (labeled ① to ④ in Figure 1) and is processed from right (i.e. region ①) to left (i.e. region ④). The computations in the regions ① to ③ involve the loading of the bytes $A[0-9]$, $B[4-23]$, and $A[10-19]$. Following the carry-once strategy, we load in region ② and ③ the intermediate-result bytes $C[20-27]$ and update them pair-wise in the following order: $C[20,21]$, $C[22,23]$, $C[24,25]$, and $C[26,27]$. In this way, we save an add instruction in the processing of each of these pairs. Once b_{botton} is finished, we pass the bytes $A[10-19]$ to the block b_{middle}, which is processed similarly.

We describe now our novel "lazy doubling" approach for squaring. Unlike to multiplication, an optimized squaring algorithm does not need to calculate all m^2 byte products since there exist a large number of pairs that have the same value, e.g. $A[1] \cdot A[0]$ and $A[0] \cdot A[1]$. After elimination of these "duplicates," we get a squaring algorithm with a triangular execution flow as illustrated on the right of Figure 1. Figure 2 shows the main steps of the our squaring algorithm

in more detail. At first, in step (a), all byte products on the top of the triangle are computed, which includes to load operand bytes $A[20-23]$ and to load and cache bytes $A[0-3]$. In step (b), all byte products of region ② are formed and the bytes $A[0-8]$ are cached. Thereafter, in step (c), we apply the carry-once technique, indicated by yellow dotted lines in Figure 2. This step uses operand bytes $A[9-17]$ and $A[0-8]$, but only the former ones are cached. The bytes $C[10-18]$ of the intermediate result are computed and updated in a pair-wise fashion, thereby saving one clock cycle per pair. In step (d), byte products are generated using bytes $A[10-23]$ and $A[0-13]$, and in step (e), computations are continued with $A[13-23]$ and $A[6-23]$, and the intermediate results are updated. Finally, in step (f), we double the whole intermediate result we got so far and then compute the remaining byte products. Our lazy doubling method requires only 2064 clock cycles to square a 192-bit integer, which improves the best previous result in the literature [21] by about 2%.

The result of a multiplication (or squaring) is a 384-bit integer, which must be reduced modulo $p = 2^{192} - 2^{64} - 1$ to get a 192-bit residue. As mentioned in Subsection 2.1 (and explained in more detail in [15, Section 2.2.6]), it is possible to perform this reduction via three 192-bit additions modulo p. However, in the worst case, three subtractions of p are necessary to get a reduced result, which can considerably slow down this operation, especially if one aims for resistance against SPA or timing attacks. Therefore, we use the "sum scanning" method for reduction modulo p proposed in [13, Algorithm 2] so that at most one final subtraction of p has to be carried out. We perform this final subtraction in an "unconditional way" using a byte-mask as described in Subsection 3.1.

3.3 Inversion

When using projective coordinates, it is generally necessary to invert the Z coordinate of the point obtained at the end of the scalar multiplication to have a final result in affine coordinates. The Extended Euclidean Algorithm (EEA) is commonly used for computing multiplicative inverses in $\mathbb{F}_p$. Unfortunately, the EEA has a very irregular execution profile and, therefore, may leak information about Z, which, in turn, could be used by an attacker to recover parts of the secret scalar. To thwart such attacks, we firstly multiply Z by a random value R, invert this product, and then multiply $(ZR)^{-1}$ again by R to get Z^{-1}.

4 Implementation Results

In this section, we firstly report the execution times of our implementation and compare them with the results of previous work. Then, we analyze the memory footprint and energy consumption of our ECC software.

4.1 Execution Time

We implemented all field operations (except of a few parts of the inversion) in AVR Assembly language and the rest (i.e. point addition, point doubling, and

Table 1. Execution time of 192-bit arithmetic operations (in clock cycles)

Implementation	mod-add	mod-sub	mod-mul	mod-sqr	mod-inv
Liu et al [22]	832	786	8,152	7,493	1,305,616
Chu et al [7]	632	632	4,845	4,052	476,055
This work	378	378	4,042	2,658	280,829

the scalar multiplication algorithms) in ANSI C. In order to achieve peak performance, we unrolled the loops of all field operations except inversion. Table 1 summarizes the execution times of the five basic arithmetic operations modulo the 192-bit NIST prime. The modular addition takes exactly the same time as the modular subtraction, namely 378 clock cycles on an 8-bit AVR ATmega128 processor. Our modular multiplication executes in about 4,000 cycles, whereas the modular squaring has an execution time of 2,658 clock cycles, which means the squaring requires merely two-third of the multiplication cycles. This result impressively demonstrates the efficiency of our "lazy doubling" technique since modular squaring is typically only about 20% faster than modular multiplication. A comparison with Liu et al's widely-used TinyECC software [22] shows that our implementation of modular addition, subtraction and multiplication is more than twice as fast as theirs, while the modular squaring gains a speed-up by a factor of roughly 2.8. Our implementation is also significantly faster than that of Chu et al [7], who used a 192-bit Optimal Prime Field (OPF) but did not unroll the loops. Our inversion modulo p has an (average) execution time of roughly 280k cycles, which means it is approximately 70 times slower than a modular multiplication. However, our inversion needs only 56% of the execution time reported in [7] and 21% of the time of the TinyECC inversion.

Table 2. Execution time (in cycles) of point addition and point doubling

Implementation	Point addition	Point doubling
Liu et al [22] (NIST P-192)	80,774	63,355
Chu et al [7] (Tw. Edwards)	54,158	41,630
This work (NIST P-192)	43,604	29,914

Table 2 shows the execution time of point addition and doubling. Compared to TinyECC, our addition achieves a speed-up of slightly below 2.0x, whereas the speed-up factor of point doubling is a bit above 2.0x. Interestingly, we are also faster than Chu et al [7], who used a twisted Edwards curve that features more efficient addition and doubling formulae than our NIST curve.

We also simulated the execution times of fixed-point and variable-point scalar multiplication as well as double scalar multiplication; they amount to some 3.67, 9.23, and 10.4 million cycles, respectively. Considering the MICAz mote's clock frequency of 7.37 MHz [8], these cycle counts translate to execution times of 0.5 s, 1.25 s, and 1.41 s. Each run of the (ephemeral) ECDH key agreement

Table 3. Comparison of fixed-point and arbitrary-point scalar multiplication, double scalar multiplication, ECDH, and ECDSA on an ATmega128 clocked at 7.37 MHz

Implementation	Field	$k \cdot P$	$l \cdot Q$	$k \cdot P + l \cdot Q$	ECDH	ECDSA
Gura et al [14]	160 b	0.88 s	0.88 s	n/a	1.76 s	n/a
Wang et al [33]	160 b	1.34 s	1.46 s	3.09 s	2.80 s	4.43 s
Szczechowiak et al [31]	160 b	1.27 s	1.27 s	n/a	2.54 s	n/a
Ugus et al [32]	160 b	0.57 s	1.03 s	n/a	1.60 s	n/a
Liu et al [22]	160 b	2.05 s	2.30 s	2.60 s	4.35 s	4.65 s
Großschädl et al [12]	160 b	0.74 s	0.74 s	n/a	1.48 s	n/a
Chu et al [7]	160 b	0.78 s	0.78 s	n/a	1.56 s	n/a
Liu et al [22]	192 b	2.99 s	2.99 s	n/a	5.98 s	n/a
Gura et al [14]	192 b	1.35 s	1.35 s	n/a	2.70 s	n/a
Lederer et al [20]	192 b	0.71 s	1.67 s	n/a	2.38 s	n/a
This work	192 b	0.50 s	1.25 s	1.41 s	1.75 s	1.91 s

protocol requires the two involved parties to execute both a fixed-point and an arbitrary-point scalar multiplication; adding them up gives an execution time of $12.9 \cdot 10^6$ clock cycles (1.75 s) altogether. On the other hand, the two main operations of ECDSA signature generation and verification, namely fixed-point scalar multiplication and double scalar multiplication, have an overall execution time of some $14 \cdot 10^6$ cycles (1.91 s). Table 3 compares our work with previous ECC implementations for 8-bit AVR-based processors. We are much faster than any other ECC software using a 192-bit prime field and outperform even some 160-bit implementations. For example, our ECDH key exchange improves the best result in the literature (which can be found in [20]) by a factor of 1.35. On the other hand, our ECDSA implementation is 2.33 times faster than the best ECDSA software reported in the literature, namely the one in [33].

4.2 Memory Footprint

Low memory footprint is another very important requirement on ECC software for sensor nodes, which becomes evident when considering that the ATmega128 on a MICAz mote has only 4 kB RAM and 128 kB flash ROM [3]. Our implementation occupies about 1.4 kB in RAM; this includes the two 384-bit tables of the comb and windows method for scalar multiplication. However, there are several options to reduce the RAM footprint. For example, when executing the comb method, it is not necessary to have the full table of pre-computed points in RAM since, at any time, only one entry of the table is required. Optimizing our implementation in this direction would reduce the RAM footprint by some 350 bytes at the expense of a slight performance degradation. The binary executable of our ECC software has a size of 28 kB, which leaves about 100 kB in flash memory for the operating system and applications.

4.3 Energy Consumption

According to [8], the ATmega128 processor of a MICAz mote draws an average current of about 8.0 mA (at a supply voltage of 3.0 V) when it is active. Since the clock frequency of the mote is known to be 7.37 MHz, we can evaluate the energy consumption of a scalar multiplication algorithm by simply forming the product of average power consumption, supply voltage, and execution time. In this way, the energy cost of a fixed-point scalar multiplication, arbitrary-point scalar multiplication, and double scalar multiplication amounts to roughly 12.0 mJ, 30.0 mJ, and 33.84 mJ, respectively. The energy consumption of the two scalar multiplications of ECDH key exchange is approximately 42.0 mJ, while the overall energy cost (for both nodes) is about 84.0 mJ. Normally, one also has to take into account the energy required for transmitting (i.e. sending and receiving) the public keys, but previous work in [9,20,27] shows that ECDH is clearly dominated by the computation energy cost. The energy required for the scalar multiplications to generate/verify an ECDSA signature is 45.84 mJ.

5 Conclusions

We introduced a carefully-optimized implementation of NIST-compliant ECC for sensor nodes equipped with an 8-bit AVR processor. Our software achieves record-setting execution times for fixed-point scalar multiplication, arbitrary-point scalar multiplication, and double scalar multiplication. For example, we outperform the best implementation of ephemeral ECDH key agreement in the literature by a factor of 1.35 and improve the state-of-the-art in ECDSA by a factor of 2.33. These speed-ups are mainly due to the performance of our field arithmetic, which is implemented in Assembly language and protected against SPA and timing attacks. We also conducted a simple energy evaluation for the ATmega128 and found that (ephemeral) ECDH key agreement consumes some 42.0 mJ per node. On the other hand, the two scalar multiplications needed to generate and verify an ECDSA signature have an energy cost of 45.84 mJ. The RAM footprint of our ECC software is 1.4 kB, which is just slightly more than one third of the total RAM of the MICAz mote. In summary, our results show that an efficient and secure (i.e. SPA-resistant) implementation of ECC on the NIST curve P-192 is possible.

References

1. Akyildiz, I.F., Su, W., Sankarasubramaniam, Y., Cayirci, E.: A survey on sensor networks. IEEE Communications Magazine 40(8), 102–114 (2002)
2. Aranha, D.F., Dahab, R., López, J.C., Oliveira, L.B.: Efficient implementation of elliptic curve cryptography in wireless sensors. Advances in Mathematics of Communications 4(2), 169–187 (2010)
3. Atmel Corporation. ATmega128(L) Datasheet (Rev. 2467O–AVR–10/06) (October 2006), http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf

4. Atmel Corporation. 8-bit ARV® Microcontroller with 128K Bytes In-System Programmable Flash: ATmega128, ATmega128L. Datasheet (June 2008), http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf
5. Bernstein, D.J.: Curve25519: New Diffie-Hellman speed records. In: Yung, M., Dodis, Y., Kiayias, A., Malkin, T. (eds.) PKC 2006. LNCS, vol. 3958, pp. 207–228. Springer, Heidelberg (2006)
6. CertiVox Corporation. CertiVox MIRACL SDK. Source code (June 2012), http://www.certivox.com
7. Chu, D., Großschädl, J., Liu, Z., Müller, V., Zhang, Y.: Twisted Edwards-form elliptic curve cryptography for 8-bit AVR-based sensor nodes. In: Xu, S., Zhao, Y. (eds.) Proceedings of the 1st ACM Workshop on Asia Public-Key Cryptography (AsiaPKC 2013), pp. 39–44. ACM Press (2013)
8. Crossbow Technology, Inc. MICAz Wireless Measurement System. Data sheet (January 2006), http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICAz_Datasheet.pdf
9. de Meulenaer, G., Gosset, F., Standaert, F.-X., Pereira, O.: On the energy cost of communication and cryptography in wireless sensor networks. In: Proceedings of the 4th IEEE International Conference on Wireless and Mobile Computing, Networking and Communications (WIMOB 2008), pp. 580–585. IEEE Computer Society Press (2008)
10. de Meulenaer, G., Standaert, F.-X.: Stealthy compromise of wireless sensor nodes with power analysis attacks. In: Chatzimisios, P., Verikoukis, C., Santamaría, I., Laddomada, M., Hoffmann, O. (eds.) MOBILIGHT 2010. LNICST, vol. 45, pp. 229–242. Springer, Heidelberg (2010)
11. Großschädl, J., Avanzi, R.M., Savaş, E., Tillich, S.: Energy-efficient software implementation of long integer modular arithmetic. In: Rao, J.R., Sunar, B. (eds.) CHES 2005. LNCS, vol. 3659, pp. 75–90. Springer, Heidelberg (2005)
12. Großschädl, J., Hudler, M., Koschuch, M., Krüger, M., Szekely, A.: Smart elliptic curve cryptography for smart dust. In: Zhang, X., Qiao, D. (eds.) QShine 2010. LNICST, vol. 74, pp. 623–634. Springer, Heidelberg (2012)
13. Großschädl, J., Savaş, E.: Instruction set extensions for fast arithmetic in finite fields gF(p) and gF(2^m). In: Joye, M., Quisquater, J.-J. (eds.) CHES 2004. LNCS, vol. 3156, pp. 133–147. Springer, Heidelberg (2004)
14. Gura, N., Patel, A., Wander, A., Eberle, H., Shantz, S.C.: Comparing elliptic curve cryptography and RSA on 8-bit cPUs. In: Joye, M., Quisquater, J.-J. (eds.) CHES 2004. LNCS, vol. 3156, pp. 119–132. Springer, Heidelberg (2004)
15. Hankerson, D.R., Menezes, A.J., Vanstone, S.A.: Guide to Elliptic Curve Cryptography. Springer (2004)
16. Hisil, H., Wong, K.K.-H., Carter, G., Dawson, E.: Twisted Edwards curves revisited. In: Pieprzyk, J. (ed.) ASIACRYPT 2008. LNCS, vol. 5350, pp. 326–343. Springer, Heidelberg (2008)
17. Hutter, M., Schwabe, P.: NaCl on 8-bit AVR microcontrollers. In: Youssef, A., Nitaj, A., Hassanien, A.E. (eds.) AFRICACRYPT 2013. LNCS, vol. 7918, pp. 156–172. Springer, Heidelberg (2013)
18. Hutter, M., Wenger, E.: Fast multi-precision multiplication for public-key cryptography on embedded microprocessors. In: Preneel, B., Takagi, T. (eds.) CHES 2011. LNCS, vol. 6917, pp. 459–474. Springer, Heidelberg (2011)
19. Kocher, P.C., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 388–397. Springer, Heidelberg (1999)

20. Lederer, C., Mader, R., Koschuch, M., Großschädl, J., Szekely, A., Tillich, S.: Energy-efficient implementation of ECDH key exchange for wireless sensor networks. In: Markowitch, O., Bilas, A., Hoepman, J.-H., Mitchell, C.J., Quisquater, J.-J. (eds.) Information Security Theory and Practice. LNCS, vol. 5746, pp. 112–127. Springer, Heidelberg (2009)
21. Lee, Y., Kim, I.-H., Park, Y.: Improved multi-precision squaring for low-end RISC microcontrollers. Journal of Systems and Software 86(1), 60–71 (2013)
22. Liu, A., Ning, P.: TinyECC: A configurable library for elliptic curve cryptography in wireless sensor networks. In: Proceedings of the 7th International Conference on Information Processing in Sensor Networks (IPSN 2008), pp. 245–256. IEEE Computer Society Press (2008)
23. Liu, Z., Wenger, E., Großschädl, J.: MoTE-ECC: Energy-scalable elliptic curve cryptography for wireless sensor networks (submitted for publication, 2013)
24. Lopez, J., Zhou, J.: Wireless Sensor Network Security. Cryptology and Information Security Series, vol. 1. IOS Press (2008)
25. Montgomery, P.L.: Speeding the Pollard and elliptic curve methods of factorization. Mathematics of Computation 48(177), 243–264 (1987)
26. National Institute of Standards and Technology (NIST). Recommended Elliptic Curves for Federal Government Use. White paper (July 1999), `http://csrc.nist.gov/encryption/dss/ecdsa/NISTReCur.pdf`
27. Piotrowski, K., Langendörfer, P., Peter, S.: How public key cryptography influences wireless sensor node lifetime. In: Zhu, S., Liu, D. (eds.) Proceedings of the 4th ACM Workshop on Security of Ad Hoc and Sensor Networks (SASN 2006), pp. 169–176. ACM Press (2006)
28. Seo, H., Kim, H.: Multi-precision multiplication for public-key cryptography on embedded microprocessors. In: Lee, D.H., Yung, M. (eds.) WISA 2012. LNCS, vol. 7690, pp. 55–67. Springer, Heidelberg (2012)
29. Seo, H., Lee, Y., Kim, H., Park, T., Kim, H.: Binary and prime field multiplication for public key cryptography on embedded microprocessors. In: Security and Communication Networks (2013)
30. Solinas, J.A.: Low-weight binary representations for pairs of integers. Technical Report CORR 2001-41, Centre for Applied Cryptographic Research (CACR), University of Waterloo, Waterloo, Canada (2001)
31. Szczechowiak, P., Oliveira, L.B., Scott, M., Collier, M., Dahab, R.: NanoECC: Testing the limits of elliptic curve cryptography in sensor networks. In: Verdone, R. (ed.) EWSN 2008. LNCS, vol. 4913, pp. 305–320. Springer, Heidelberg (2008)
32. Ugus, O., Westhoff, D., Laue, R., Shoufan, A., Huss, S.A.: Optimized implementation of elliptic curve based additive homomorphic encryption for wireless sensor networks. In: Wolf, T., Parameswaran, S. (eds.) Proceedings of the 2nd Workshop on Embedded Systems Security (WESS 2007), pp. 11–16 (2007), `http://arxiv.org/abs/0903.3900`
33. Wang, H., Li, Q.: Efficient implementation of public key cryptosystems on mote sensors. In: Ning, P., Qing, S., Li, N. (eds.) ICICS 2006. LNCS, vol. 4307, pp. 519–528. Springer, Heidelberg (2006)
34. Yanık, T., Savaş, E., Koç, Ç.K.: Incomplete reduction in modular arithmetic. IEE Proceedings – Computers and Digital Techniques 149(2), 46–52 (2002)