Efficient Implementation of ECDH Key Exchange for MSP430-Based Wireless Sensor Networks

Zhe Liu
University of Luxembourg, Luxembourg
zhe.liu@uni.lu

Hwajeong Seo
Pusan National University, Republic of Korea
hwajeong84@gmail.com

Zhi Hu
Central South University, P.R. China
huzhi@math.pku.edu.cn

Xinyi Huang
Fujian Normal University, P.R. China
xyhuang81@gmail.com

Johann Großschädl
University of Luxembourg, Luxembourg
johann.groszschaedl@uni.lu

ABSTRACT

Public-Key Cryptography (PKC) is an indispensable building block of modern security protocols, and, thus, essential for secure communication over insecure networks. Despite a significant body of research devoted to making PKC more "lightweight," it is still commonly perceived that software implementations of PKC are computationally too expensive for practical use in ultra-low power devices such as wireless sensor nodes. In the present paper we aim to challenge this perception and present a highly-optimized implementation of Elliptic Curve Cryptography (ECC) for the TI MSP430 series of 16-bit microcontrollers. Our software is inspired by MoTE-ECC and supports scalar multiplication on two families of elliptic curves, namely Montgomery and twisted Edwards curves. However, in contrast to MoTE-ECC, we use pseudo-Mersenne prime fields as underlying algebraic structure to facilitate inter-operability with existing ECC implementations. We introduce a novel "zig-zag" technique for multiple-precision squaring on the MSP430 and assess its execution time. Similar to MoTE-ECC, we employ the Montgomery model for variable-base scalar multiplications and the twisted Edwards model if the base point is fixed (e.g. to generate an ephemeral key pair). Our experiments show that the two scalar multiplications needed to perform an ephemeral ECDH key exchange can be accomplished in 4.88 million clock cycles altogether (using a 159-bit prime field), which sets a new speed record for ephemeral ECDH on a 16-bit processor. We also describe the curve generation process and analyze the execution time of various field and point arithmetic operations on curves over a 159-bit and a 191-bit pseudo-Mersenne prime field.

ASIA CCS'15, April 14–17, 2015, Singapore, Singapore.

http://dx.doi.org/10.1145/2714576.2714608.

Categories and Subject Descriptors

E.3 [**Data**]: Data Encryption—*Public Key Cryptosystems*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms

Algorithms, Experimentation, Performance, Security

Keywords

Wireless sensor networks, Ephemeral ECDH key exchange, Multi-precision arithmetic, Pseudo-Mersenne prime

1. INTRODUCTION

In recent years, the security of Wireless Sensor Networks (WSNs) has been an active area of research, mainly due to the widespread deployment of WSN technology in critical application domains like medical monitoring, traffic control or disaster detection [1]. The wireless nature of communication among sensor nodes implies that WSNs face the same security threats as other wireless networks; for example, an attacker may eavesdrop on the communication, inject false messages, or replay old messages [17]. In addition, WSNs can be subject to many other, very specific attacks that are difficult to defend against [17]. Deploying sensor nodes in an unattended environment exposes them to attackers who could tamper with individual nodes or even capture a node with the goal of extracting sensitive information stored on it. Strong cryptography is essential to protect stored data against unauthorized access or to thwart eavesdropping on the communication between nodes. However, sophisticated cryptographic algorithms introduce significant overheads in terms of execution time and energy consumption, which is undesirable for resource-restricted devices such as wireless sensor nodes. A state-of-the-art sensor node features an 8 or 16-bit microcontroller clocked at a frequency of less than 10 MHz and is equipped with a few kB of RAM and up to 256 kB of flash memory for storing program code. Despite these computational constraints, the (by far) most precious resource of a wireless sensor node is energy. Once deployed in the field, the sensor nodes are expected to work several months, or even years, with the energy of two AA batteries that can not be easily recharged or replaced.

A prerequisite for secure (i.e. encrypted) communication between two sensor nodes is the establishment of a shared secret key. Standardized security protocols like SSL or TLS utilize Public-Key Cryptography (PKC) to set up a shared key between a client and a server, either through key transport based on RSA or by means of key agreement based on the Diffie-Hellman scheme. However, due to the restricted resources of wireless sensor nodes, it was generally believed that PKC is not feasible for WSNs [1]. Therefore, a large body of research has been devoted to find more lightweight approaches for establishing shared keys between nodes, see e.g. [17] and [12, Section 1.1] for a brief overview. A simple idea is *random key pre-distribution*, where, prior to deployment, each sensor node is loaded with a set of keys chosen randomly from a large key pool so that any two nodes will share (at least) one key with a certain probability. While this approach is fairly easy to implement and preserves the scarce resources of the nodes (since no costly cryptographic operations are involved), it has limitations with respect to scalability and resilience to node capture. Another method to obtain pairwise secret keys in a WSN is to use a trusted third party (e.g. the base station) as a *key distribution center* that generates, upon request, a unique "link key" for a pair of nodes. This approach requires each node to share a long-term key with the trusted third party, which is used to transfer the link key securely (i.e. encrypted) to the two nodes. However, key establishment techniques relying on a key distribution center require communication among three parties, possibly over large distances, which is expensive in terms of energy.

In 2004, Gura et al [7] published a now-classical paper in which they demonstrated that, in contrast to conventional wisdom, strong PKC is feasible for small battery-powered sensor nodes. In particular, they showed that Elliptic Curve Cryptography (ECC) [8], when carefully implemented and optimized, is computationally less costly than was believed at that time. For example, a scalar multiplication (which is the major operation of an ECC cryptosystem) in a 160-bit elliptic curve group (providing a similar level of security as 1024-bit RSA) can be executed in only 0.81 seconds on an 8-bit ATmega128 microcontroller clocked at 8.0 MHz. The feasibility of ECC on such resource-restricted sensor nodes paved the way to another option for key establishment in a WSN, namely the well-known Elliptic Curve Diffie-Hellman (ECDH) key exchange protocol. ECDH is quite similar to the traditional Diffie-Hellman protocol, but operates in an elliptic curve group $E(\mathbb{F}_q)$ instead of $\mathbb{Z}_p^*$ [8]. As mentioned before, the computationally expensive part of virtually all ECC schemes, including ECDH, is scalar multiplication, an operation of the form $k \cdot P$ whereby P is a point of prime order n on an elliptic curve, and k is simply an integer in the range $[1, n-1]$. There exist two major variants of the ECDH protocol, namely static and ephemeral ECDH. The latter yields a unique secret key in each run of the protocol and, hence, can provide forward secrecy, but this comes at the cost of an additional scalar multiplication. Ephemeral ECDH requires each node to execute two scalar multiplications; one by a fixed base point (to generate an ephemeral key pair) and the other by a variable base point (to obtain the shared secret key).

The feasibility of performing ECC on resource-restricted sensor nodes does not necessarily imply that ECDH is an attractive option for key establishment in a WSN [23]. Even though ECDH has clear advantages over other techniques for key establishment with respect to communication energy cost, it is still widely believed that scalar multiplication is too computation-intensive and, therefore, consuming too much energy for "real-word" WSN applications. To address this critique, a large body of research has been devoted to improve the performance of scalar multiplication on 8 and 16-bit microcontrollers with the goal of making ECC more attractive for resource-restricted environments. Besides the 8-bit ATmega128, also Texas Instruments' MSP430 series of low-power microcontrollers [24] has been frequently used as experimental platform since it can be found in a range of sensor nodes (e.g. the Tmote Sky). One of the first ECC software implementations for the TI MSP430 was reported by Wang et al [25], who achieved an execution time of 25.0 and 28.1 million cycles for a fixed-base and a variable-base scalar multiplication, respectively, using a Weierstraß curve over a 160-bit prime field. The by far most important ECC software for MSP430-based WSNs is TinyECC [13], whose source code is openly available and has been incorporated into numerous WSN research projects. TinyECC supports elliptic curves over 128, 160, and 192-bit primes fields and is highly scalable and configurable. In the past 5 years, the majority of research focussed on improving the execution time of the arithmetic operations in the underlying prime field, in particular the multiplication; examples for this line of research are [5, 9, 18, 20, 23, 27]. Only very recently, a second approach to speed up ECC on the MSP430 platform has been investigated, namely the use of "special" families of curves (e.g. Montgomery curves [19] or twisted Edwards curves [3]) to improve the point arithmetic; representative papers in this context are [10, 14]. A scalar multiplication on Curve25519 (a Montgomery curve over a 255-bit prime field [2]) can be executed in only 9,14 million clock cycles on an MSP430 with a (16×16)-bit multiplier [10].

In this paper, we describe an efficient ECC implementation that achieves record-setting execution times for fixed-base and variable-base scalar multiplication on an MSP430 processor. We managed to push the performance envelope through the right selection of curve models (and associated domain parameters) combined with a careful optimization of the point/field arithmetic. All previous implementations of ECC for MSP430 processors used either a conventional Weierstraß curve (e.g. [20, 23, 27]) or a Montgomery curve of unreasonably large order (e.g. [10]), both of which wastes execution time and, thus, energy. In contrast, our software supports Montgomery and Edwards curves over a 159 and a 191-bit prime field, which represents a good compromise between performance and security. Our implementation is inspired by MoTE-ECC [16], but we use pseudo-Mersenne prime fields instead of the so-called Optimal Prime Fields (OPFs) [15] to facilitate inter-operability with other ECC implementations. We implemented and optimized all field operations from scratch in Assembly language, whereby we paid particular attention to the squaring since it was often ignored in related work. Thanks to our special combination of curve parameters and efficient point/field arithmetic, we managed to significantly improve the state-of-the-art.

2. ELLIPTIC CURVES

Ephemeral ECDH requires each of the two sensor nodes involved in the key exchange to carry out two scalar multiplications; one to generate an ephemeral key pair and the

other to get the shared key. The first scalar multiplication uses a fixed and a-priori-known point as input, namely the generator G of the elliptic-curve group, whereas the second scalar multiplication has to be carried out with a random point not known in advance. Consequently, each of the two nodes has to perform both a fixed-base and a variable-base scalar multiplication [8]. Recently, Liu et al [16] introduced MoTE-ECC, an optimized software library for ECC on the 8-bit AVR platform that allows for very efficient execution of ephemeral ECDH key exchange in a WSN. MoTE-ECC uses both the Montgomery model and the twisted Edwards model, whereby solely the x-coordinates of the public keys are transferred, as in [2], which reduces the communication energy cost and makes point compression [8] obsolete. The fixed-base scalar multiplication is performed on a twisted Edwards curve, and the variable-base scalar multiplication on the birationally-equivalent Montgomery curve. In this way, MoTE-ECC is able to combine the individual computational advantages of the twisted Edwards model and the Montgomery model. Our implementation for the MSP430 follows this approach and, thus, we also support both curve models. In the rest of this section, we first recap the basics of Montgomery and twisted Edwards curves, and then we elaborate on the curve generation process.

2.1 Montgomery & Twisted Edwards Curves

In 1987, Peter Montgomery [19] presented a special class of elliptic curves with unique implementation properties. In formal terms, a so-called Montgomery curve E_M over $\mathbb{F}_p$ is defined through an equation of the form

$$E_M : \; By^2 = x^3 + Ax^2 + x, \tag{1}$$

where $A \in \mathbb{F}_p \setminus \{-2, 2\}$ and $B \in \mathbb{F}_p \setminus \{0\}$. A characteristic feature of such curves is that a scalar multiplication can be executed using only the x-coordinate of the base point and all intermediate points. The so-called Montgomery ladder is a special technique for computing $Q = k \cdot P$ that performs a differential point addition (i.e. an addition of two points P_1, P_2 whose difference $P_1 - P_2$ is known) and a doubling in each step. Since the y-coordinate is not involved in the computation, a differential point addition can be executed very efficiently with three multiplications (i.e. 3M) and two squarings (i.e. 2S) in the underlying finite field. Doubling a point requires two multiplications (i.e. 2M), two squarings (i.e. 2S), as well as a multiplication by $(A-4)/2$, which is normally a cheap operation when the curve parameter A is chosen properly [2]. For each bit of the scalar k, the Montgomery ladder has to execute a "ladder step" consisting of a point addition and a point doubling; both costs 5M and 4S altogether. Besides efficiency, the Montgomery ladder also features a highly regular execution pattern, which helps to thwart certain implementation attacks.

Twisted Edwards curves were introduced by Bernstein et al in 2008 [3] and are currently considered to be one of the fastest means to implement ECC. A twisted Edwards curve E_T over a prime field $\mathbb{F}_p$ can be defined as

$$E_T : \; ax^2 + y^2 = 1 + dx^2y^2, \tag{2}$$

where $a, d \in \mathbb{F}_p$ and $ad(a-d) \neq 0$. These curves possess a remarkable addition law that can be complete when a is a square in $\mathbb{F}_p$ and d a non-square. Completeness means the addition produces the correct result for any two points on the curve E_T, without exception, even if one of the points is the neutral element $\mathcal{O} = (0, 1)$ [3]. Similar to most of the previous implementations, we adopt the so-called extended twisted Edwards coordinates introduced in [11], which are particularly efficient when parameter $a = -1$. After some straightforward optimizations of the original formulae from [11], a mixed point addition can be carried out using seven multiplications (7M), while three multiplications (3M) and four squarings (4S) are necessary to double a point. Mixed here means that one of the points to be added is given in extended projective coordinates, and the other in extended affine coordinates. We represent a projective point via the quintuple $(X : Y : E : H : Z)$, whereby $EH = T = XY/Z$ is the fourth coordinate from [11]. An extended affine point is a triple $(u : v : w)$ where $u = (x + y)/2$, $v = (y - x)/2$, and $w = dxy$. Besides efficiency, twisted Edwards curves have a second major advantage, namely that scalar multiplication can be easily made resistant against Simple Power Analysis (SPA) thanks to the completeness of the addition law.

Bernstein et al formally proved in [3] that every twisted Edwards curve E_T over a non-binary field $\mathbb{F}_q$ is birationally equivalent over $\mathbb{F}_q$ to a Montgomery curve E_M and vice versa. They also gave a set of formulae for the conversion of points on E_T to points on E_M and back.

2.2 Generation of MoTE Curves

A MoTE curve can be described as a Montgomery curve that is birationally equivalent to a twisted Edwards curve with a fast and complete addition law [16]. "Fast" in this context means that we can apply the 7M mixed-addition formula mentioned above, which is only possible when the curve parameter a is -1. On the other hand, completeness of the twisted Edwards addition requires a to be a square in $\mathbb{F}_p$ and d a non-square. Consequently, it is only possible to have a fast and complete addition law when $a = -1$ is a square in the underlying prime field $\mathbb{F}_p$, which is the case if and only if $p \equiv 1 \bmod 4$. In other words, a MoTE curve can only be generated over a prime field $\mathbb{F}_p$ whose order is congruent to 1 modulo 4 [6, 16]. The authors of the original MoTE-ECC paper [16] used so-called Optimal Prime Fields (OPFs) as underlying algebraic structure, which are defined by primes of the form $p = u \cdot 2^k + 1$ where u has a length of up to 16 bits [15]. While such primes are always congruent to 1 mod 4, they have the disadvantage that the computation of square roots mod p is costly, which poses a problem for application scenarios where point compression is desirable[1]. A further drawback of OPFs is that, despite their excellent arithmetic properties (see e.g. [15]), they are not (yet) widely supported by other ECC implementations for WSNs. Therefore, and since we want our software to be suitable for applications that require point compression, we eventually decided to not use OPFs.

Our ECC software for 16-bit TI MSP430 microcontrollers employs pseudo-Mersenne prime fields in order to facilitate inter-operability with other implementations and support a variety of ECC protocols and applications, including ones that require point compression. Although we focus only on ECDH key exchange in this paper, the two MoTE curves we present in this section are not limited to ECDH but can

[1]It should be noted that point compression is obsolete in an "x-coordinate-only" ECDH key exchange as described in the MoTE-ECC paper [16] and earlier in [2]. The ability to efficiently de-compress a point is mainly useful for schemes that need both the x and y coordinate (e.g. signatures).

be used with any other scheme. A pseudo-Mersenne prime has the form $p = 2^k - c$ where c is small enough to fit into a single register on the target platform, which means c can be (at most) 16 bits long in our case. In order to achieve both a complete addition law and fast square-root computations, we use primes that are congruent to 5 mod 8. Since $p \equiv 5 \bmod 8$ implies $p \equiv 1 \bmod 4$, the parameter $a = -1$ is clearly a square modulo p. On the other hand, if a prime p satisfies $p \equiv 5 \bmod 8$, it is possible to compute the square root of an element of $\mathbb{F}_p$ in a fairly efficient way using the method of Atkin [6], which, in essence, costs an exponentiation modulo p. Following the approach of Bernstein from [2], we adopt primes with a bitlength slightly less than the "nominal" length, e.g. 159 instead of 160 bits. To be more concrete, the exponent k of our pseudo-Mersenne primes is a multiple of 32 minus 1. Using primes that leave one bit of "headroom" simplifies the implementation of arithmetic operations modulo p if one aims for both high performance and some basic resistance against SPA attacks [14]. Once the exponent k is fixed, a suitable value for c needs to be determined, which we simply did by choosing the minimal c so that $p = 2^k - c$ is a prime congruent to 5 modulo 8. In this way, we found the 159-bit prime $p = 2^{159} - 91$ and the 191-bit prime $p = 2^{191} - 19$.

When generating elliptic curves for cryptographic applications, one needs to take into account both security and efficiency requirements. The SafeCurves website [4] defines a number of criteria a curve has to satisfy to be considered secure, whereby it distinguishes between ECC security and ECDLP security. On top of the list of requirements for the latter is that the group of rational points on a given curve contains a large subgroup of prime order ℓ [4]. SafeCurves requires ℓ to be at least 2^{200}, in which case computing the ECDLP is infeasible with today's technology. However, the requirement to use groups of such large order contradicts with the common practice of using groups of order between 2^{160} and 2^{192} for WSN applications, see e.g. [13]. This is, in general, due to the fact that sensor nodes are a low-value target for cryptanalytic attacks, and, consequently, orders of about 2^{160} still provide ample protection for applications in such areas like home automation. Furthermore, it has to be taken into account that the ECDLP is almost never the weakest link in the security of WSNs [17]. In practice, the most serious vulnerability of sensor nodes is their physical exposure to attackers along with the lack of tamper resistance. A serious attacker would never attempt to break the ECDLP in a 160-bit group as he can get the secret key in a much cheaper way via reverse engineering. Therefore, we decided to use elliptic curves with orders of about 2^{160} and 2^{192}; both orders are well established and supported in the WSN research community.

As specified in [6], the "Montgomery shape" of a MoTE curve is given by an equation of the form

$$E_M : \ -(A+2)y^2 = x^3 + Ax^2 + x, \tag{3}$$

which means the parameter $B = -(A+2)$. This contrasts with the "conventional" approach of using $B = 1$ as in the case of e.g. Curve25519 [2]. Choosing a Montgomery curve with $B = -(A+2)$ has the the advantage that such a curve is "directly" birationally equivalent to a twisted Edwards curve possessing a fast addition law [11], i.e. to a twisted Edwards curve with $a = -1$. Given a Montgomery curve with parameters A, B as in Equation (1), the parameters of the birationally-equivalent twisted Edwards curve can be computed as follows (see [3] for details).

$$a = (A+2)/B \quad \text{and} \quad d = (A-2)/B \tag{4}$$

Therefore, when $B = -(A+2)$, the curve parameter a we obtain is $a = (A+2)/(-(A+2)) = -1$, which simplifies the conversion of points between the Montgomery and the twisted Edwards shape (see [6] for additional details). On the other hand, when $B = 1$, the conversion of a point on the Montgomery curve to a point on the birationally-equivalent twisted Edwards curve with $a = (A+2)/B$ and then from there to an isomorphic twisted Edwards curve of the form $-x^2 + y^2 = 1 + (-d/a)x^2y^2$ (which allows one to use the fast 7M addition formula) is more complex and requires the pre-computation of $1/\sqrt{-a}$ as explained in [11, Section 3.1]. The parameter A of the Montgomery form of a MoTE curve is chosen such that $A - 2$ is a square and $A + 2$ is a non-square in $\mathbb{F}_p$, in which case $d = (A-2)/(-(A+2))$ is also a non-square (at least if $p \equiv 1 \bmod 4$) and the twisted Edwards addition law is complete. Last, but not least, the parameter A is congruent to 2 mod 4 to ensure $(A+2)/4$ is small and a multiplication by $(A+2)/4$ can be performed efficiently. Taking all this and some further considerations (e.g. twist security) into account, we generated the MoTE curves P159 and P191, which we specify below. The "P" in the name of these curves stands for pseudo-Mersenne prime and the subsequent 3-digit number denotes the bitlength.

MoTE Curve P159

P159 is a MoTE curve over the 159-bit prime field $\mathbb{F}_p$ given by $p = 2^{159} - 91$. The Montgomery shape of curve P159 is defined through the equation

$$E_{M159} : \ -3191568y^2 = x^3 + 3191566x^2 + x \tag{5}$$

(i.e. $A = 3191566$ and $B = -(A+2) = -3191568$). This Montgomery curve is birationally equivalent to the twisted Edwards curve of the form

$$E_{T159} : \ -x^2 + y^2 = 1 + dx^2y^2, \tag{6}$$

where $d = 837225916393474870456088348941705219765626$ 63492. Curve P159 has an order of $q = 4\ell < p$ where ℓ is a prime slightly smaller than 2^{157}. On the other hand, the quadratic twist of P159 has an order of $q' = 8\ell' > p$ where ℓ' is a prime slightly larger than 2^{156}. In other words, curve P159 has a co-factor of 4, whereas its quadratic twist has a co-factor of 8. Both the curve and its twist have a large embedding degree of above 2^{100}. Moreover, curve P159 has a large CM field discriminant that fully complies with the SafeCurves requirements (see [6] for further details).

MoTE Curve P191

P191 is a MoTE curve over the 191-bit prime field $\mathbb{F}_p$ given by $p = 2^{191} - 19$. The Montgomery shape of curve P191 is defined through the equation

$$E_{M191} : \ -2678312y^2 = x^3 + 2678310x^2 + x \tag{7}$$

(i.e. $A = 2678310$ and $B = -(A+2) = -2678312$). This Montgomery curve is birationally equivalent to the twisted Edwards curve of the form

$$E_{T191} : \ -x^2 + y^2 = 1 + dx^2y^2, \tag{8}$$

where $d = 103951507655322023199378042616749966478813$ 474077612237402. Curve P191 has very similar features as

curve P159 with respect to order (resp. co-factor) of curve and twist, embedding degree of curve and twist, as well as CM field discriminant. A more detailed description of this curve can be found in [6].

3. FIELD ARITHMETIC

Most practical implementations of ECC adopt some kind of "special" prime fields to speed up the modular reduction operation; well-known examples are fields whose order is a Mersenne-like prime, e.g. a generalized or pseudo-Mersenne prime. In this section, we will focus only on the latter class of primes. A so-called pseudo-Mersenne prime has the form $p = 2^k - c$ where c is small in relation to 2^k; typically, c is chosen to fit into a register of the target processor. This is clearly met by the primes $p = 2^{159} - 91$ and $p = 2^{191} - 19$ from the previous section. The major idea of fast reduction modulo a pseudo-Mersenne prime $p = 2^k - c$ is to utilize the congruence relation $2^k \equiv c \bmod p$ repetitively until the obtained residue has the same bitlength as p. Suppose z is an integer of a length of $l > k$ bits. At first, z needs to be split up into a lower part z_L comprising the k least significant bits of z and an upper part z_H that comprises all the other bits, i.e. we can write $z = z_H 2^k + z_L$. Now, we have to substitute 2^k by c and get

$$z = z_H 2^k + z_L \equiv z_H c + z_L \tag{9}$$

This new value for z is, of course, in the same residue class as the original z, but at least $k - 1$ bits shorter. To obtain a further reduced result, we simply apply this substitution repeatedly until z has the same bitlength as p.

The elements of a prime field $\mathbb{F}_p$ with $p = 2^k - c$ are the integers from 0 to $p - 1$, which are at most k bits long. As usual, we represent the field elements in the form of arrays of w-bit words, where w corresponds to the word-size of the target processor, i.e. $w = 16$ in our case. A k-bit operand consists of exactly $m = \lceil k/w \rceil$ words, which means $m = 10$ for 159-bit operands and $m = 12$ for $k = 191$. We will use indexed lowercase letters to denote the individual words in an array, e.g. $a \in \mathbb{F}_p$ is stored in an m-word array of the form $a = (a_{m-1}, \ldots, a_1, a_0)$ with $0 \leq a_i < 2^w$ where a_0 and a_{m-1} represent the least and most significant w-bit word of a, respectively. Similar to numerous other ECC software implementations, e.g. [15], we adopt the idea of incomplete modular reduction, which means all arithmetic operations modulo p accept operands that are not fully reduced (and hence not less than p), as long as they fit into m words. In our case, the operands can be up to $n = mw = k + 1$ bits long because the exponent k of the primes we use is not a multiple of w but one bit shorter. To give a more concrete example, the operands of the arithmetic operations modulo our 159-bit prime can be up to $n = 160$ bits long and also the results can have a length of up to 160 bits.

3.1 Addition and Subtraction

An addition of two elements of $\mathbb{F}_p$ is, basically, a normal integer addition, yielding a sum that can be (at most) one bit longer than the operands, followed by a reduction mod p. Algorithm 1 shows our implementation of the addition modulo a k-bit pseudo-Mersenne prime p, whereby we use a similar notation as in Section 2.2.1 of [8]. The word-wise additions in line 3 and line 9 are add-with-carry operations with ε representing the carry bit. As explained above, the operands a, b do not necessarily need to be smaller than p, but they must fit into m words, i.e. they can be up to $n = k + 1$ bits long. The first part of Algorithm 1 performs a conventional multiple-precision addition, yielding a sum of up to $n + 1$ bits, of which n bits are stored in the words of s and the final carry bit in ε. In line 5 we combine the "excess bits" (i.e. ε and the most significant bit of s_{m-1}) in t, which corresponds to z_H in Equation (9). Since the sum s before reduction is (at most) $n + 1$ bits long, t can have a length of two bits, i.e. $0 \leq t \leq 3$. The next step (line 6) is to clear the most significant bit of s_{m-1}, after which the sum s is at most k bits long. Now we multiply t by c, add the product to s_0, and finally propagate the carry up to s_{m-1}. The result is at most $k + 1$ bits long because it had a length of k bits before the addition of $t \cdot c$. Thus, it can be used as operand in another field operation. Note that the modular addition in Algorithm 1 has a constant execution time as it does not contain any conditional statements.

Algorithm 1. Addition modulo a pseudo-Mersenne prime

Input: Two m-word operands $a = (a_{m-1}, \ldots, a_0)$ and $b = (b_{m-1}, \ldots, b_0)$, and a prime of the form $p = 2^k - c$.
Output: Modular sum $s = a + b \bmod p = (s_{m-1}, \ldots, s_0)$.

```
1:  (ε, s_0) ← a_0 + b_0
2:  for i = 1 to m − 1 do
3:      (ε, s_i) ← a_i + b_i + ε
4:  end for
5:  t ← (ε ≪ 1) + (s_{m−1} ≫ 15)          { 0 ≤ t ≤ 3}
6:  s_{m−1} ← s_{m−1} & 0x7fff            { clear the MSB of s_{m−1}}
7:  (ε, s_0) ← t · c + s_0                { main step of reduction}
8:  for i = 1 to m − 1 do
9:      (ε, s_i) ← s_i + ε
10: end for
11: return s
```

A subtraction in $\mathbb{F}_p$ can be performed in a similar way as the modular addition described above. To prevent negative results, we execute the subtraction $s = a - b \bmod p$ via an operation of the form $s = 3p + a - b \bmod p$. Since we use pseudo-Mersenne primes, the addition of $3p$ does not cause much overhead due to the fact that most words of $3p$ have the same value, namely $2^w - 1$, and so they do not need to be loaded from memory. The reduction step can be done in exactly the same way as described above.

3.2 Multiplication and Squaring

Multiplication and squaring in $\mathbb{F}_p$ are in general the two most performance-critical field operations performed in the course of a scalar multiplication [8]. When using a pseudo-Mersenne prime, it is common practice to do the reduction mod p after the multiplication instead of executing them in an interleaved fashion as in [15]. Also our implementation for the MSP430 follows this basic approach. Some MSP430 models feature a memory-mapped hardware multiplier able to execute (16×16)-bit multiply and multiply-accumulate operations on both signed and unsigned integers [24]. Since the multiplier is a memory-mapped peripheral, it has to be accessed by writing the two operands to specific locations in memory. Concretely, the first operand of a multiply or a MAC operation has to be written to one of four addresses (`MPY`, `MPYS`, `MAC`, or `MACS`) and this address determines the actual operation to be issued. The second operand must be written to another specific address (`OP2`), and once this has happened, the selected operation is immediately executed

[24]. After a few clock cycles, the 32-bit result is available at the addresses RESLO (lower 16 bits) and RESHI (upper 16 bits). In the case of a MAC operation, the product of the two operands is added to the content of RESHI|RESLO and the obtained cumulative sum is written back to RESHI and RESLO [24]. The carry bit produced by the accumulation is written to another address, namely SUMEXT. When several multiplications with one and the same operand need to be carried out, it is possible to "re-use" this operand, provided that it is the first operand. Namely, the first operand can be used in consecutive multiplications or consecutive MAC operations without having to write it again to one of the four addresses that select the type of operation.

Multiple-Precision Multiplication

Most ECC implementations for the MSP430 platform use the so-called product-scanning technique (or an optimized variant of it) to perform multi-precision multiplication, see e.g. [5, 21, 22]. The product-scanning method computes the product of two multiple-precision integers in a column-wise fashion and performs MAC operations in its loops, i.e. two w-bit words are multiplied and the $2w$-bit product is added to a cumulative sum. Hence, the product-scanning method performs well on MSP430 processors. Our implementation is also based on product scanning, but we incorporated a number of low-level optimizations. For example, we use all free registers to "cache" 16-bit words of the operands so as to minimize the number of memory accesses. Moreover, we adapt the order in which the partial products are processed with the goal of increasing the number of subsequent MAC operations that can use one and the same operand. In this way, we can save a few clock cycles as the re-used operand has to be written to MAC only once. Listing 1 shows a code snippet that illustrates the computation of the first two partial products of a column. The MOV instructions in line 1 and 2 write two operand words (which are accessed via the pointers APTR and BPTR) to MAC and OP2. As explained above, the 32-bit result of a MAC operation is placed in RESHI and RESLO, while the carry from the accumulation is written to SUMEXT. In line 3 and 7, the carry bit is added into a register named CARRY. More precisely, the carry from the first MAC operation of a column can be directly moved to CARRY, while subsequent carries need to be added.

```
1:          MOV @APTR+, &MAC
2:          MOV @BPTR, &OP2
3:          MOV &SUMEXT, CARRY
4:          SUB #2, BPTR
5:          MOV @APTR+, &MAC
6:          MOV @BPTR, &OP2
7:          ADD &SUMEXT, CARRY
8:          SUB #2, BPTR
```

Listing 1: Product-scanning technique using MAC operations

Multiple-Precision Squaring

Since squaring received rather modest attention in previous work (and was completely ignored in e.g. [18]), it bears the potential for noticeable speed-ups through optimization. In essence, squaring is a special case of multiplication since all partial products of the form $a_i \cdot a_j$ with $i \neq j$ appear twice in the result due to the fact that $a_i \cdot a_j = a_j \cdot a_i$. Dedicated squaring techniques compute these partial products once and then double them, which reduces the number of word-level multiplication or MAC operations by nearly 50%. The squaring routine we implemented involves two steps; in the first step, all partial products to be doubled are generated and summed up. Then, in the second step, the intermediate result obtained so far is doubled and the partial products from the "'main diagonal" (i.e. the partial products that are themselves squares of the form $a_i \cdot a_i$) are added to yield the full result. The first step can be optimized in the same way as the multi-precision multiplication, which means we should use free registers to "cache" operand words and we should re-order the processing of partial products with the goal of executing a number of consecutive MAC operations with the same first operand. For example, once a_0 has been written to address MAC, we can process the partial products $a_0 \cdot a_2$ and $a_0 \cdot a_3$ by only writing a_2 and a_3 to OP2.

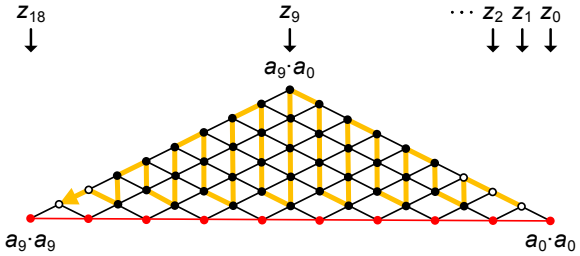

Figure 1: Zig-zag squaring of a 160-bit integer

```
1:          CLR R15
2:          MOV R15, &RESLO
3:          MOV R15, &RESHI
4:          // A[0]*A[1]
5:          MOV @APTR, &MAC
6:          MOV @APTR2+, &OP2
7:          MOV &RESLO, 2(R14)
8:          MOV &RESHI, &RESLO
9:          MOV R15, &RESHI
10:         // A[0]*A[2]
11:         MOV @APTR2+, &OP2
12:         MOV &RESLO, 4(R14)
13:         MOV &RESHI, &RESLO
14:         MOV R15, &RESHI
15:         // A[0]*A[3]
16:         MOV @APTR2+, &OP2
```

Listing 2: Zig-zag squaring using MAC operations

Taking all these optimization strategies into account, we found that the most efficient way to perform the first step is to process the partial products in a "zig-zag" fashion as illustrated by the thick yellow line in Figure 1. We call this approach "zig-zag squaring" because it is somewhat related to the zig-zag multiplication described in [27]. Each dot in Figure 1 represents one partial product. The computation starts at the right corner and proceeds to the left. Both the first column on the yellow line (i.e. the column yielding the word z_1 of $z = a^2$) and the second column (z_2) consist of a single partial product each, namely $a_0 \cdot a_1$ and $a_0 \cdot a_2$, while the third column (producing the word z_3) contains the two partial products $a_0 \cdot a_3$ and $a_1 \cdot a_2$. Listing 2 demonstrates the computation of the first three partial products on the yellow line, whereby this code snippet shows that a_0 needs to be written to the address MAC only once and can then be

used as operand in three MAC operations. The first three partial products (i.e. $a_0 \cdot a_1$, $a_0 \cdot a_2$, and $a_0 \cdot a_3$) allow some special optimizations as their accumulation into `RESLO` and `RESHI` can not cause an overflow, i.e. `SUMEXT` is 0 and does not need to be added to register `CARRY`. This also simplifies the w-bit right-shift of the cumulative sum, which has to be performed at the end of each column in order to ensure a proper alignment of the partial products. The white dots in Figure 1 indicate all partial products that do not require a carry propagation. When the first step of the squaring is completed, the intermediate result has to be doubled and the partial products located on the red line at the bottom of Figure 1 (i.e. $a_0 \cdot a_0$ to $a_9 \cdot a_9$) must be added, which can be done together with the doubling in one pass.

Modular Reduction

Since the operands of a multiplication or squaring may be incompletely reduced, the resulting product or square has a length of up to $2n = 2k + 2$ bits, but nonetheless always fits into $2m$ words. Our implementation of the reduction is fully optimized for pseudo-Mersenne primes $p = 2^k - c$ and consists of two steps. Let z be a $2m$-word product. In the first step, z is split into an upper part z_H that comprises the m most significant words and a lower part z_L with all the other words. At the beginning of this section we defined the exponent k of our primes to be a multiple of 32 minus 1, which implies that k is a multiple of w minus 1. Thus, we have $2p = 2(2^k - c) = 2^n - d$ where $n = k + 1 = mw$ and $d = 2c$. Given $z = z_H \cdot 2^{mw} + z_L$, the first reduction step is to compute $z' \equiv z \bmod p$ as follows

$$z' = z_H \cdot d + z_L \tag{10}$$

Taking our 159-bit prime $p = 2^{159} - 91$ as an example, we have $d = 2c = 182$. Since the two operands to be multiplied must be less than 2^{160} each, it can be shown that z' has a maximum length of 168 bits and fits into 11 words. Using $2p$ instead of p in the first reduction step allows us to avoid shift operations, which would otherwise be necessary since k is not a multiple of the word size. The second step of the reduction operation is the same as in the modular addition (i.e. line 5 to 10 in Algorithm 1), except that the variable t has to be composed of z'_m (shifted one bit to the left) and the MSB of z'_{m-1}. More formally, $t = z'/2^k$, which means t has a length of 9 bits in our 159-bit example. The product $t \cdot c$ has a length of 16 bits, and, consequently, the result we get after the second reduction step is at most 160 bits as in the modular addition.

3.3 Fermat-Based Inversion

Using projective coordinates for the point arithmetic has the advantage that only a single inversion in $\mathbb{F}_p$ needs to be executed, namely at the very end of a scalar multiplication to convert the result from projective to affine coordinates [8]. There exist two principal approaches for performing an inversion in $\mathbb{F}_p$, namely the Extended Euclidean Algorithm (EEA) and inversion via exponentiation based on Fermat's little theorem. The EAA is, in general, more efficient than Fermat's technique, but has an irregular execution pattern and an operand-dependent execution time, both of which is problematic if one aims for resistance against side-channel attacks. Therefore, we implemented the inversion by means of an exponentiation of the form $a = z^{-1} \equiv z^{p-2} \bmod p$. In order to minimize the number of modular multiplications needed for this exponentiation, we use an addition-chain as shown in Algorithm 2 for our 159-bit prime. The comments in each line specify the computational cost of the operation carried out in that line, i.e. the number of multiplications (M) and squarings (S). Also given is the intermediate value of the exponent based on all operations executed until that line. In total, an inversion modulo our 159-bit prime costs 158 squarings and 11 multiplications.

Algorithm 2. Fermat-based inversion mod $p = 2^{159} - 91$

Input: Integer a satisfying $1 \leq a \leq p - 1$.
Output: Inverse $z = a^{p-2} \bmod p = a^{-1} \bmod p$.

1:	$a_2 \leftarrow a^2$	{exp: 2, cost: 0M+1S}
2:	$a_3 \leftarrow a_2 \cdot a$	{exp: 3, cost: 1M+0S}
3:	$a_{15} \leftarrow (a_3)^{2^2} \cdot a_3$	{exp: 15, cost: 2S+1M}
4:	$t_1 \leftarrow (a_{15})^{2^4} \cdot a_{15}$	{exp: $2^8 - 1$, cost: 4S+1M}
5:	$t_2 \leftarrow (t_1)^{2^8} \cdot t_1$	{exp: $2^{16} - 1$, cost: 8S+1M}
6:	$t_3 \leftarrow (t_2)^{2^{16}} \cdot t_2$	{exp: $2^{32} - 1$, cost: 16S+1M}
7:	$t_4 \leftarrow (t_3)^{2^{32}} \cdot t_3$	{exp: $2^{64} - 1$, cost: 32S+1M}
8:	$t_5 \leftarrow (t_4)^{2^{64}} \cdot t_4$	{exp: $2^{128} - 1$, cost: 64S+1M}
9:	$t_6 \leftarrow ((t_5)^{2^{16}} \cdot t_2)^{2^8} \cdot t_1$	{exp: $2^{152} - 1$, cost: 24S+2M}
10:	$z \leftarrow ((t_6)^{2^2} \cdot a)^{2^5} \cdot a_3$	{exp: $2^{159} - 93$, cost: 7S+2M}
11:	**return** z	

A Fermat-based inversion modulo $p = 2^{191} - 19$ can be performed in a similar fashion using an optimized addition chain, whereby the overall computational cost amounts to 190 modular squarings and 12 modular multiplications.

4. EXPERIMENTAL RESULTS

Ephemeral ECDH key exchange requires each of the two involved nodes to perform a fixed-base and a variable-base scalar multiplication. Since our implementation is based on MoTE curves, we can execute the former with a fixed-base comb method (using the twisted Edwards model), whereas the second scalar multiplication can take advantage of the existence of a birationally-equivalent Montgomery model and its high efficiency in variable-base scenarios. Similar to Curve25519 [2], we exchange only the x-coordinates of the ephemeral public keys. Thus, it is necessary to convert the point obtained as result of the first scalar multiplication to a point on the equivalent Montgomery curve. This can be done in combination with the projective-affine conversion so that only one inversion is necessary for both conversions (see [16] for the conversion formulae).

Operation	159 bit	191 bit
Addition	108	133
Subtraction	124	156
Multiplication	1,828	2,555
Squaring	1,505	1,983
Inversion	268,547	419,823

Table 1: Execution time (in clock cycles) of field arithmetic operations.

We determined the execution time of various arithmetic operations with help of the cycle-accurate simulator that is part of IAR Embedded Workbench 6.10 using the F1611 as target device. Table 1 summarizes the results of the field

Implementation	Field Mul	Fixed SM	Variab SM	Full ECDH	Regular	Device
Implementations using curves over a 159 or 160-bit prime field						
Liu and Ning [13]	n/a	12,645,000	12,645,000	25,290,000	No	MSP430F1611
Marin et al [18]	6,293	10,020,000	10,020,000	20,040,000	No	n/a
Wenger and Werner [27]	3,112	8,779,931	8,779,931	17,559,862	n/a	MSP430F1611
Hinterwälder et al [9]	2,266	6,312,785	6,312,785	12,625,570	No	MSP430F2618
Szczechowiak et al [23]	2,736	5,760,000	5,760,000	11,520,000	No	MSP430F1611
Wenger [26]	1,905	5,721,420	5,721,420	11,442,840	Yes	MSP430C11x1
Gouvêa and López [5]	1,952	1,831,063	4,417,661	6,248,724	No	n/a
This work (curve P159)	1,828	1,635,056	3,248,819	4,883,875	Yes	MSP430F1611
Implementations using curves over a 191 or 192-bit prime field						
Wenger and Werner [27]	n/a	11,949,000	11,949,000	23,898,000	n/a	MSP430F1611
Wenger [26]	2,559	9,100,128	9,100,128	18,200,256	Yes	MSP430C11x1
This work (curve P191)	2,555	2,604,338	5,121,517	7,725,855	Yes	MSP430F1611

Table 3: Execution times (in clock cycles) of our ECC software and some previous implementations.

arithmetic for both of our prime fields. For example, a full multiplication (including reduction) in our 159-bit field has an execution time of just 1828 clock cycles, which improves the 160-bit multiplication times in [5] and [26] by 124 and 77 cycles, respectively. An execution time of 1828 cycles is, to our knowledge, the best result for multiplication in a prime field of about 160 bits ever reported in the literature and, therefore, represents a new speed record. Squaring in our 159-bit prime field is approximately 18% faster than a multiplication, whereas the Fermat-based inversion has an execution time of roughly 147 multiplications. Note that all field operations listed in Table 1 have a regular execution profile and a constant execution time, independent of the actual value of the operands, which helps to thwart certain implementation attacks.

Operation	159 bit	191 bit
TE Point Add.	14,685	19,852
TE Point Dbl.	13,263	17,514
Mon Point Add.	10,276	13,586
Mon Point Dbl.	8,183	10,733

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

We implemented the point arithmetic (i.e. point addition and doubling) in C, whereby we used Assembly functions for the field operations as subroutines. The execution times for the Montgomery shape and the twisted Edwards shape on both curve P159 and P191 are specified in Table 2. As expected, the point arithmetic on the Montgomery curve is faster than that on the twisted Edwards curve, mainly due to the fact that the differential point addition/doubling on a Montgomery curve does not involve the y coordinate. In summary, the simulation results in Table 2 agree with the number of field multiplications and squarings as analyzed in Subsection 2.1. A fixed-base scalar multiplication using the twisted Edwards model of MoTE curve P159 requires approximately $1.635 \cdot 10^6$ cycles on the MSP430F1611. We perform a fixed-base scalar multiplication via a fixed-base comb method with 8 pre-computed multiples of the base point so that four bits of the scalar can be processed at a time. A detailed description of this comb method, which is highly regular and has constant (i.e. operand-independent) execution time, can be found in [16]. As mentioned at the outset of this section, the fixed-base scalar multiplication also includes the conversion of the obtained point from the twisted Edwards curve to the birationally-equivalent Montgomery curve, on which the second scalar multiplication is performed. However, since this second scalar multiplication is variable-base, we use the Montgomery ladder to execute it in an efficient fashion. Taking curve P159 as example, the second (i.e. variable-base) scalar multiplication executes in $3.249 \cdot 10^6$ cycles, which means the total computation time of ephemeral ECDH amounts to $4.884 \cdot 10^6$ cycles.

Table 3 compares our work with previous ECC software implementations for MSP430 devices in terms of execution time of a field multiplication (second column), a fixed-base scalar multiplication (third column), a variable-base scalar multiplication (fourth column), and an ECDH key exchange (fifth column). Also specified is whether an implementation features a regular execution profile (sixth column) and the specific device (seventh column). Our implementations on both curve P159 and curve P191 significantly improve the state-of-the-art; for example, ECDH on curve P159 outperforms the best previous implementation (i.e. [5]) by more than 21.84%. However, it should be taken into account in this comparison that most of the previous implementations only considered variable-base scalar multiplication.

5. CONCLUSIONS

We presented a high-speed implementation of ephemeral ECDH key exchange based on MoTE elliptic curves for the MSP430 family of microcontrollers. Our software exploits the birational equivalence between the Montgomery model and the twisted Edwards model of a MoTE curve with the goal of maximizing the performance of both fixed-base and variable-base scalar multiplication. In the case of a MoTE curve over a 159-bit pseudo-Mersenne prime, a fixed-base scalar multiplication (using the twisted Edwards form and eight pre-computed points) takes $1.635 \cdot 10^6$ cycles on an MSP430F1611, while a variable-base scalar multiplication on the birationally-equivalent Montgomery curve requires $3.249 \cdot 10^6$ clock cycles. Consequently, both scalar multiplications can be executed in only $4.884 \cdot 10^6$ cycles, which sets a new speed record for the computation of an ephemeral ECDH key exchange on the MSP430 platform and improves the previously best result in the literature by 21,84%. We

achieved this performance gain through a careful selection of curve models and domain parameters, aiming for a good balance between security and efficiency, combined with a highly-optimized implementation of the low-level field and group arithmetic.

6. ACKNOWLEDGEMENTS

Zhe Liu is supported by the Fonds National de la Recherche (FNR) Luxembourg under AFR grant no. 1359142.

Hwajeong Seo is supported by the MSIP (Ministry of Science, ICT and Future Planning), Korea, under the ITRC (Information Technology Research Center) support program (NIPA-2014-H0301-14-1048) supervised by the NIPA (National IT Industry Promotion Agency.

Xinyi Huang is supported by National Natural Science Foundation of China (61472083, U1405255, 61202450), Fok Ying Tung Education Foundation (141065), Ph.D. Programs Foundation of the Ministry of Education of China (2012350 3120001), Program for New Century Excellent Talents in Fujian University (JA14067), Distinguished Young Scholars Fund of Department of Education, Fujian Province, China (JA13062).

7. REFERENCES

[1] I. F. Akyildiz and M. C. Vuran. *Wireless Sensor Networks*. John Wiley and Sons, 2010.

[2] D. J. Bernstein. Curve25519: New Diffie-Hellman speed records. In *Public Key Cryptography — PKC 2006*, vol. 3958 of *Lecture Notes in Computer Science*, pp. 207–228. Springer Verlag, 2006.

[3] D. J. Bernstein, P. Birkner, M. Joye, T. Lange, and C. Peters. Twisted Edwards curves. In *Progress in Cryptology — AFRICACRYPT 2008*, vol. 5023 of *Lecture Notes in Computer Science*, pp. 389–405. Springer Verlag, 2008.

[4] D. J. Bernstein and T. Lange. SafeCurves: Choosing safe curves for elliptic-curve cryptography. Avialable online at `http://safecurves.cr.yp.to`, 2013.

[5] C. P. Gouvêa and J. López. Software implementation of pairing-based cryptography on sensor networks using the MSP430 microcontroller. In *Progress in Cryptology — INDOCRYPT 2009*, vol. 5922 of *Lecture Notes in Computer Science*, pp. 248–262. Springer Verlag, 2009.

[6] J. Großschädl. A family of implementation-friendly MoTE elliptic curves. Technical report TR-LACS-2013-01, Laboratory of Algorithmics, Cryptology and Security (LACS), University of Luxembourg, Luxembourg, 2013.

[7] N. Gura, A. Patel, A. S. Wander, H. Eberle, and S. Chang Shantz. Comparing elliptic curve cryptography and RSA on 8-bit CPUs. In *Cryptographic Hardware and Embedded Systems — CHES 2004*, vol. 3156 of *Lecture Notes in Computer Science*, pp. 119–132. Springer Verlag, 2004.

[8] D. R. Hankerson, A. J. Menezes, and S. A. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer Verlag, 2004.

[9] G. Hinterwälder, C. Paar, and W. P. Burleson. Privacy preserving payments on computational RFID devices with application in intelligent transportation systems. In *Radio Frequency Identification Security and Privacy Issues — RFIDSec 2012*, vol. 7739 of *Lecture Notes in Computer Science*, pp. 109–122. Springer Verlag, 2012.

[10] G. Hinterwälder, A. Moradi, M. Hutter, P. Schwabe, and C. Paar. Full-size high-security ECC implementation on MSP430 microcontrollers. In *Progress in Cryptology — LATINCRYPT 2014*, vol. 8895 of *Lecture Notes in Computer Science*, pp. 31–47. Springer Verlag, 2015.

[11] H. Hişil, K. K.-H. Wong, G. Carter, and E. Dawson. Twisted Edwards curves revisited. In *Advances in Cryptology — ASIACRYPT 2008*, vol. 5350 of *Lecture Notes in Computer Science*, pp. 326–343. Springer Verlag, 2008.

[12] C. Lederer, R. Mader, M. Koschuch, J. Großschädl, A. Szekely, and S. Tillich. Energy-efficient implementation of ECDH key exchange for wireless sensor networks. In *Information Security Theory and Practice — WISTP 2009*, vol. 5746 of *Lecture Notes in Computer Science*, pp. 112–127. Springer Verlag, 2009.

[13] A. Liu and P. Ning. 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, 2008.

[14] Z. Liu, J. Großschädl, L. Li, and Q. Xu. Energy-efficient elliptic curve cryptography for MSP430-based wireless sensor nodes. Preprint, submitted for publication, 2015.

[15] Z. Liu, J. Großschädl, and D. S. Wong. Low-weight primes for lightweight elliptic curve cryptography on 8-bit AVR processors. In *Information Security and Cryptology — INSCRYPT 2013*, vol. 8567 of *Lecture Notes in Computer Science*, pp. 217–235. Springer Verlag, 2014.

[16] Z. Liu, E. Wenger, and J. Großschädl. MoTE-ECC: Energy-scalable elliptic curve cryptography for wireless sensor networks. In *Applied Cryptography and Network Security — ACNS 2014*, vol. 8479 of *Lecture Notes in Computer Science*, pp. 361–379. Springer Verlag, 2014.

[17] J. Lopez and J. Zhou. *Wireless Sensor Network Security*. IOS Press, 2008.

[18] L. Marin, A. J. Jara, and A. F. Gómez-Skarmeta. Shifting primes: Extension of pseudo-Mersenne primes to optimize ECC for MSP430-based future Internet of things devices. In *Availability, Reliability and Security for Business, Enterprise and Health Information Systems*, vol. 6908 of *Lecture Notes in Computer Science*, pp. 205–219. Springer Verlag, 2011.

[19] P. L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, Jan. 1987.

[20] C. Pendl, M. Pelnar, and M. Hutter. Elliptic curve cryptography on the WISP UHF RFID tag. In *RFID Security and Privacy — RFIDSec 2011*, vol. 7055 of *Lecture Notes in Computer Science*, pp. 32–47. Springer Verlag, 2012.

[21] H. Seo, Y. Lee, H. Kim, T. Park, and H. Kim. Binary and prime field multiplication for public key cryptography on embedded microprocessors. *Security and Communication Networks*, 7(4):774–787, Apr. 2014.

[22] H. Seo, K.-A. Shim, and H. Kim. Performance enhancement of TinyECC based on multiplication optimizations. *Security and Communication Networks*, 6(2):151–160, Feb. 2013.

[23] P. Szczechowiak, L. B. Oliveira, M. Scott, M. Collier, and R. Dahab. NanoECC: Testing the limits of elliptic curve cryptography in sensor networks. In *Wireless Sensor Networks — EWSN 2008*, vol. 4913 of *Lecture Notes in Computer Science*, pp. 305–320. Springer Verlag, 2008.

[24] Texas Instruments, Inc. MSP430x1xx Family User's Guide (Rev. F). Manual, vailable for download at `http://www.ti.com/lit/ug/slau049f/slau049f.pdf`, 2006.

[25] H. Wang, B. Sheng, and Q. Li. Elliptic curve cryptography-based access control in sensor networks. *International Journal of Security and Networks*, 1(3–4):127–137, Dec. 2006.

[26] E. Wenger. Hardware architectures for MSP430-based wireless sensor nodes performing elliptic curve cryptography. In *Applied Cryptography and Network Security — ACNS 2013*, vol. 7954 of *Lecture Notes in Computer Science*, pp. 290–306. Springer Verlag, 2013.

[27] E. Wenger and M. Werner. Evaluating 16-bit processors for elliptic curve cryptography. In *Smart Card Research and Advanced Applications — CARDIS 2011*, vol. 7079 of *Lecture Notes in Computer Science*, pp. 166–181. Springer Verlag, 2011.