Twisted Edwards-Form Elliptic Curve Cryptography for 8-bit AVR-based Sensor Nodes

Dalin Chu
Shandong University, China
chudalin@gmail.com

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

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

ABSTRACT

Wireless Sensor Networks (WSNs) pose a number of unique security challenges that demand innovation in several areas including the design of cryptographic primitives and protocols. Despite recent progress, the efficient implementation of Elliptic Curve Cryptography (ECC) for WSNs is still a very active research topic and techniques to further reduce the time and energy cost of ECC are eagerly sought. This paper presents an optimized ECC implementation that we developed from scratch to comply with the severe resource constraints of 8-bit sensor nodes such as the MICAz and IRIS motes. Our ECC software uses Optimal Prime Fields (OPFs) as underlying algebraic structure and supports two different families of elliptic curves, namely Weierstraß-form and twisted Edwards-form curves. Due to the combination of efficient field arithmetic and fast group operations, we achieve an execution time of $5.9 \cdot 10^6$ clock cycles for a full 160-bit scalar multiplication on an 8-bit ATmega128 microcontroller, which is 2.78 times faster than the widely-used TinyECC library. Our implementation also shows that the energy cost of ephemeral ECDH key exchange between two MICAz (or IRIS) motes amounts to only 38.7 mJ per mote (including radio communication). A mote with a standard AA battery pack could theoretically perform up to 174,278 ECDH key exchanges before running out of energy.

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

MICAz Mote, AVR Processor, Twisted Edwards Curve

1. INTRODUCTION

In recent years, Wireless Sensor Networks (WSNs) have found widespread adoption in such areas as environmental monitoring, military surveillance, industrial control, home automation, and health care [1]. Many of said applications collect or process sensitive information, which initiated an extensive body of research on security and privacy aspects of WSNs. The special adversary models and threat scenarios of WSNs pose a multitude of unique research problems (see e.g. [21] for an overview), including some that are still not properly solved and, hence, need further consideration [15]. Wang et al [23] identify the following building blocks as essential for the design and implementation of a secure WSN: cryptography, key management, secure routing, secure data aggregation, and intrusion detection. One of the open research issues mentioned in [23] is to further improve the efficiency of Public-Key Cryptography (PKC) on small sensor nodes with limited computational power. They state that "public key cryptography can greatly ease the design of security in WSNs" [23, p. 19], but perceive overheads in execution time and energy consumption as limiting factors for the widespread deployment of PKC.

The benefits and drawbacks of using PKC in WSNs have been widely researched in the past ten years. Early work on the feasibility of PKC in WSNs includes that of Carman et al [5], who analyzed and compared the computation time and energy requirements of RSA, DSA, Diffie-Hellman and a few other public-key algorithms. The first really practical RSA implementation for an 8-bit sensor node, namely the prevalent MICAz mote [7], was presented by Gura et al in 2004 [11]. They also introduced highly-optimized software for Elliptic Curve Cryptography (ECC) on 8-bit AVR micro-controllers and reported an execution time of less than $10 \cdot 10^6$ clock cycles for a 192-bit scalar multiplication. This result set a new speed record for ECC on an 8-bit platform and has since then been generally regarded as the ultimate proof that strong PKC is feasible on resource-constrained sensor nodes. One of the most widely used ECC implementations for WSNs is TinyECC [18], whose first version was released in the late 2007. TinyECC is a highly configurable ECC library for wireless sensor nodes running TinyOS and supports Weierstraß curves over arbitrary prime fields. To increase efficiency, TinyECC contains special optimizations for standardized 128, 160, and 192-bit fields.

In this paper, we describe a carefully-optimized software implementation of ECC for 8-bit AVR-based sensor nodes like the MICAz and IRIS motes. The aim of our work is to advance the state-of-the-art in lightweight ECC for WSNs

by exploring the potential of new families of elliptic curves and prime fields with special arithmetic properties. In contrast, most existing ECC libraries for 8-bit AVR processors (in particular TinyECC) are optimized for curves and fields that have been standardized by such bodies as the Natinal Institute of Standards and Technology (NIST) [20]. These so-called NIST curves were specified some 15 years ago and do not reflect the current state-of-the-art of ECC in terms of efficiency. Our implementation "departs" from these old standards and puts forward a novel approach for ECC on small sensor nodes that combines twisted Edwards curves (which provide very fast point arithmetic [3]) with so-called Optimal Prime Fields (which allow for efficient modular reduction [24]). Besides achieving high performance, we also aim for a "lightweight" implementation with low RAM and ROM footprint. Therefore, we use the conventional double-and-add method for scalar multiplication, even though one could reach better execution times at the cost of additional memory for storing multiples of the base point. Our results show that an 8-bit sensor node, such as the MICAz mote, is able to perform a full 160-bit scalar multiplication in some $5.9 \cdot 10^6$ clock cycles, which is roughly 2.8 times faster than the widely-used TinyECC library.

2. OPTIMAL PRIME FIELDS

The finite field we use in our implementation belongs to the family of *Optimal Prime Fields (OPFs)*, which were first mentioned in the literature in an extended abstract from 2006 [10]. These fields are represented by "low-weight" primes that can be written as $p = u \cdot 2^k + v$, where u and v are relatively small compared to 2^k; in our case, u has a length of 16 bits so that it fits into two registers of an ATmega128 processor, while v is equal to 1. More specifically, our implementation uses $p = 65356 \cdot 2^{144} + 1$, which happens to be a 160-bit prime that looks as follows when written in hex notation.

0xFF4C0000000000000000000000000000000000000001

Primes of such form are characterized by a low Hamming weight since only the two most significant bytes and the least significant byte are non-zero; all other bytes are zero. The low weight of p allows for optimization of the modular arithmetic because only the non-zero bytes of p need to be processed in the reduction operation. For example, Montgomery multiplication [19] can be optimized for these primes so that the modular reduction has only linear complexity, similar to generalized-Mersenne or pseudo-Mersenne primes [12].

Our previous work [24] describes an efficient OPF arithmetic library for 8-bit AVR processors end explains how to speed up modular reduction for low-weight primes. The ECC implementation we introduce in this paper uses the OPF library from [24] as a building block for the low-level field arithmetic. However, we had to write the Assembly code for inversion in OPFs from scratch since it was not included in the library. For the sake of completeness, we start this section with an overview of how the library performs addition, subtraction, multiplication, and squaring in OPFs, before we describe the inversion in more detail. Throughout this paper, we will use the following notation. Uppercase letters denote arrays of w-bit words representing field elements, while indexed uppercase letters refer to individual words within an array, e.g. A_i is the i-th word of an array A that represents $a \in \mathbb{F}_p$. Even though AVR is an 8-bit architecture, we use a word-size of $w = 32$ bits for performance reasons, which means the arithmetic operations of our library generally process four bytes at once [11]. Given an operand length of n bits, the total number of words is $s = \lceil n/w \rceil$, which means $s = 5$ for a 160-bit OPF.

2.1 Addition and Subtraction

Addition and subtraction are the most basic operations in multiple-precision arithmetic. To calculate the modular sum $a + b \bmod p$, we first do the addition and then perform the reduction. Let A_i, B_i be the i-th word of the arrays A and B, which represent $a, b \in \mathbb{F}_p$. The addition starts with $A_0 + B_0$, and then repeatedly calculates $A_i + B_i + c$ for $0 < i < s$, whereby c denotes the carry bit generated in the previous addition of 32-bit words. After addition of the most significant words, we have a sum that is up to $n + 1$ bits long. We simply use the carry bit from the last addition of words to decide whether or not to subtract p, which is faster than an exact comparison, but may lead to an incompletely reduced result in the range of $[0, 2^n - 1]$ instead of the least non-negative reside. However, this is not a problem in practice since all arithmetic functions of the OPF library can handle incompletely reduced operands.

The modular substraction $a - b \bmod p$ is very similar to the modular addition, except that the prime p has to be added if the difference $a - b$ is negative.

2.2 Multiplication and Squaring

Modular multiplication and squaring are the most important arithmetic operations for public-key cryptography, especially for ECC. Our OPF library uses Montgomery's algorithm [19] for the modular multiplication and takes the low weight of the prime into account. As mentioned before, the primes we use have the form $p = u \cdot 2^{16} + 1$, whereby u has a length of at most 16 bits. An s-word array P representing p contains only two non-zero words, namely P_{s-1} and P_0. There exist different techniques for efficient computation of the Montgomery product; one of these is the so-called *Finely Integrated Product Scanning (FIPS)* method [16], which performs multiplication and modular reduction in an interleaved fashion. The FIPS method normally executes $2s^2 + s$ word-level (i.e. $(w \times w)$-bit) multiplications, but this number drops by roughly one half to $s^2 + s$ when the modulus is a low-weight prime as defined above [24]. Computing the product of two s-word operands requires s^2 word-level multiplications, which means the overhead of modular reduction is only s word-level multiplications.

The FIPS method consists of two nested loops, both performing Multiply-ACcumulate (MAC) operations in the inner loop. In our case (i.e. $w = 32$), the inner loop operation requires multiplying two 32-bit words (which takes 16 `mul` instructions on an AVR processor) and adding the 64-bit product to a 72-bit cumulative sum held in nine registers. The concrete implementation of the FIPS method included the OPF library follows the basic idea of hybrid multiplication [11], but executes the inner-loop operation in a more efficient way as described in [24]. An iteration of the inner loop needs only need 101 clock cycles, which allows a (160×160)-bit multiplication (without reduction) to be performed in 3006 clock cycles (including function-call overhead). The execution of a FIPS Montgomery multiplication modulo a 160-bit low-weight prime p as defined above takes

3521 clock cycles of no final subtraction of p is required. On the other hand, if a final subtraction is necessary, the execution time increases to 3588 cycles. However, the average execution time of multiplication in a 160-bit OPF was reported in [24] to be 3542 cycles.

Similar to modular multiplication, also the implementation of modular squaring uses a word-size of $w = 32$, which means four bytes of the operand are processed per iteration of the inner loop. However, modular squaring in a 160-bit OPF is roughly 20% faster than modular multiplication due to the fact any word-product of the form $A_i \cdot A_j$ is identical to $A_j \cdot A_i$. Consequently, all word-products $A_i \cdot A_j$ with $i \neq j$ need to be computed only once and then added twice to the cumulative sum in order to be doubled. According to [24], squaring an element of a 160-bit OPF takes 2966 clock cycles in the best case (i.e. no final subtraction has to be performed) and 3032 cycles in the worst case. Further details about the efficient implementation of multiplication and squaring in OPFs can be found in [24].

2.3 Inversion

The standard method to calculate the inverse of an element of a prime field is to use the extended Euclidean algorithm [6]. A straightforward implementation of this algorithm requires computationally expensive division operations. To avoid these divisions, we employ a different method, known as binary inversion algorithm, which is also based on the Euclidean law but adopts much cheaper shifts, divisions by 2 and subtractions as described in Section 2.2.5 of [12].

The binary variant of Euclidean's algorithm computes a^{-1} mod p by finding an integer x such that $ax + py = 1$. The algorithm maintains the invariants

$$ax_1 + py_1 = u \quad \text{and} \quad ax_2 + py_2 = v \tag{1}$$

where y_1 and y_2 are not explicitly computed. We initialize $u = a$, $v = p$, $x_1 = 1$ and $x_2 = 0$[1] to get the first equations

$$a + p \cdot 0 = u \quad \text{and} \quad a \cdot 0 + p = v$$

The binary variant of the Euclidean algorithm has a simple loop structure. In the body of the loop, we manipulate the above equations in two ways. First, when u or v is even, both sides of the equation are divided by 2. Taking u as an example, if u is even, we can get $u/2$ simply by a right shift operation. At the same time, we need to calculate $x_1/2$ mod p, which is a little more complex. If x_1 is even, a right shift operation is sufficient to get $x_1/2$, otherwise, we can not do the right shift directly since the lowest bit is 1. In this case, we need to adopt another element of the same residue class, namely $x_1 + p \equiv x_1 \bmod p$. On the other hand, the second case is when u and v are both odd. Without loss of generality, we assume $u > v$, and subtract v from u. The left sides of the equations in (1) do the corresponding subtraction at the same time. The algorithm terminates when $u = 1$ or $v = 1$. In the former case, $ax_1 + py_1 = 1$ and hence $a^{-1} = x_1 \bmod p$, while in the latter case, $ax_2 + py_2 = 1$ and $a^{-1} = x_2 \bmod p$.

3. TWISTED EDWARDS CURVES

In July 2007, Harold Edwards introduced a normal form for elliptic curves along with a simple, symmetric addition law [9]. Bernstein and Lange [4] established the relevance of Edwards' work for elliptic curve cryptography and came up with more efficient formulas for point addition and doubling using standard projective coordinates [12]. They also extended Edwards' curve definition to a more general form that covers a much larger class of elliptic curves. In formal terms, a so-called Edwards curve over a prime field $\mathbb{F}_p$ can be described by the equation[2]

$$E : x^2 + y^2 = 1 + dx^2y^2 \tag{2}$$

with $d \in \mathbb{F}_p \setminus \{0, 1\}$. Edwards curves have some attractive properties for practical use, most notably efficiency of the point arithmetic and completeness of the addition law when d is not a square in $\mathbb{F}_p$. Completeness means the addition formula is valid for all $P, Q \in E(\mathbb{F}_p)$, including the special cases $P = Q$, $P = -Q$, $P = \mathcal{O}$, and $Q = \mathcal{O}$. Bernstein and Lange [4] also showed that every Edwards curve contains a point of order 4 and, thus, has a co-factor of $h \geq 4$.

In 2008, Bernstein et al [3] introduced Twisted Edwards curves as a generalization of Edwards curves. Formally, a twisted Edwards curve over a prime field $\mathbb{F}_p$ is defined via the equation

$$E : ax^2 + y^2 = 1 + dx^2y^2 \tag{3}$$

where a and d are distinct, non-zero elements of $\mathbb{F}_p$. Bernstein et al observed empirically that the twisted Edwards form covers much more curves than the "original" Edwards form[3] based on Equation 2. Furthermore, as demonstrated in [3], every twisted Edwards curve over a non-binary field $\mathbb{F}_q$ is birationally equivalent over $\mathbb{F}_q$ to a Montgomery curve (i.e. every twisted Edwards curve can be transformed to a Montgomery curve, and vice versa). Bernstein et al [3] also presented explicit formulas for addition and doubling on a twisted Edwards curve; these formulas are complete if a is a square and d a non-square in the underlying field.

The most efficient way of performing point arithmetic on a twisted Edwards curve is to use the extended coordinates proposed by Hişil et al in [13]. When using this coordinate system, a point $P = (x, y)$ is represented by the quadruple $(X : Y : T : Z)$ where $x = X/Z$, $y = Y/Z$, $xy = T/Z$, and $Z \neq 0$. Such extended twisted Edwards coordinates can be seen as homogenous projective coordinates $(X : Y : Z)$, augmented with a fourth coordinate T that corresponds to the product xy in affine coordinates. The point at infinity $\mathcal{O}$ is represented by $(0 : 1 : 0 : 1)$ and the negative of a point in extended coordinates is $(-X : Y : -T : Z)$. A point given in affine coordinates as (x, y) can be converted to extended coordinates by simply setting $X = x$, $Y = y$, $T = xy$, and $Z = 1$. The re-conversion is done in the very same way as for homogenous projective coordinates through calculation of $x = X/Z$ and $y = Y/Z$, which costs an inversion in the underlying field.

3.1 Curve Generation

Both the IEEE standard P1363 [14] and the textbook of Cohen et al [6] specify a number of security criteria

[1] Actually, the initial values of y_1 and y_2 are 0 and 1. Because we do not need to calculate these values, we just ignore them in the initializations.

[2] Note that Bernstein and Lange originally defined Edwards curves more generally over non-binary fields. However, in this paper we only consider prime fields.

[3] Of course, the conventional Edwards curves from [4] are a subset of twisted Edwards curves since an Edwards curve is nothing else than a twisted Edwards curve with $a = 1$.

that an elliptic curve has to fulfill to be suitable for use in cryptography. The most important of these requirements, which ensure that the Elliptic Curve Discrete Logarithm Problem (ECDLP) is hard, are summarized below:

- First of all, the additive group of points on the curve has to contain a large subgroup of prime order m to ensure the Elliptic Curve Discrete Logarithm Problem (ECDLP) is hard. In other words, the curve should have a small co-factor h; ideally, the co-factor is 1.
- Secondly, in order to avoid the Semaev-Smart-Satoh-Araki (SSSA) attack, the curve must not be anomalous. More precisely, the order of the additive group of points on the curve must not be equal to the order of underlying prime field, i.e. $\#E(\mathbb{F}_p) \neq p$.
- Finally, to prevent the Menezes-Okamoto-Vanstone (MOV) attack and other attacks based on the Weil and Tate pairing, the embedding degree of the curve must not be small (i.e. the order of the EC group n must not divide $p^k - 1$ for "small" values of k)

We used the computer algebra system Magma to generate (i.e. find) a suitable twisted Edwards curve for ECC [3]. As mentioned in Section 2, our implementation is based on the prime field $\mathbb{F}_p$ with $p = 65356 \cdot 2^{144} + 1$, which belongs to the family of OPFs [24] and has a size of 160 bits. Magma provides an extensive pool of functions for computations on elliptic curves given in both short and long (non-simplified) Weierstraß form, but does not directly support the twisted Edwards from. However, a twisted Edwards curve with the parameters $a, d \in \mathbb{F}_p$ can be expressed via a non-simplified Weierstraß equation as follows.

$$a_2 = 2(a+d),\ a_4 = (a-d)^2, \text{ and } a_1 = a_3 = a_6 = 0 \quad (4)$$

The above formulas were derived by simply exploiting the fact that any twisted Edwards curve over a non-binary field $\mathbb{F}_q$ is birationally equivalent to a Montgomery curve, which was formally proven in [3]. The Magma script we wrote to generate a twisted Edwards curve contains a simple loop in which the parameter d (initially set to 1) gets incremented each iteration until a suitable curve is found. We fixed the parameter a to -1 (i.e. $a = p - 1$) to take advantage of the fast formulas for point addition and doubling presented in [13]. Furthermore, our Magma script only considers values of d that are non-square in $\mathbb{F}_p$ so as to ensure completeness of the addition formula. In each iteration of the loop, three main steps are carried out. First, the twisted Edwards curve defined by a and d is transformed into a Weierstraß curve via Equation (4). Next, we determine the number of $\mathbb{F}_p$-rational points on this curve using Magma's `Order` function and check whether it is four times a prime (i.e. whether its co-factor h is 4). If this is the case then the final step is to carry out some further checks to guarantee the ECDLP is hard. Following the outlined approach, we eventually found the curve $E : -x^2 + y^2 = 1 + 31145x^2y^2$ (i.e. $a = p - 1$ and $d = 31145$), which has a cardinality $\#E(\mathbb{F}_p)$ of

$$4 \cdot 364371875798791851509551807137352597688979500323$$

whereby the latter factor is a 158-bit prime. In addition to the already-mentioned requirements for the hardness of the ECDLP, we also checked a couple of other criteria such as "twist security." The quadratic twist E' of our curve E has a small co-factor of 8, which helps to prevent certain forms of implementation attack [2].

3.2 Point Arithmetic

In the following, $\mathbb{F}_p$ denotes a prime field and E a twisted Edwards curve in the form described by Equation (3) over $\mathbb{F}_p$. Let $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ be points on E. Then, the group law for E can be described as follows.

1. **Identity**: $(0, 1)$ is a the neutral element.
2. **Inverse**: The inverse element of the point $P_1 = (x_1, y_1)$ on curve E is $-P_1 = (-x_1, y_1)$.
3. **Addition law**: The addition law for the twisted Edwards curve is:

$$\begin{aligned} P_1 + P_2 &= (x_1, y_1) + (x_2, y_2) \\ &= \left(\frac{x_1y_2 + y_1x_2}{1 + dx_1y_1x_2y_2}, \frac{y_1y_2 - ax_1x_2}{1 - dx_1y_1x_2y_2} \right) \end{aligned} \quad (5)$$

When using projective coordinates, the twisted Edwards curve E can be brought into the form

$$(aX^2 + Y^2)Z^2 = Z^4 + dX^2Y^2 \quad (6)$$

where the point $(X : Y : Z)$ is equivalent to $(X/Z, Y/Z)$ in affine coordinates.

As mentioned earlier in this section, Hişil et al [13] introduced a more efficient approach for the representation of points, known as extended twisted Edwards coordinates. In this representation, an additional coordinate t is used, which is simply the product of the (affine) x and y coordinates, i.e. $t = xy$. One can transfer a point in affine to projective coordinates via the map $(x, y, t) \mapsto (x : y : t : 1)$. A point $(X : Y : T : Z) = (\lambda X : \lambda Y : \lambda T : \lambda Z)$ that satisfies Equation (6) corresponds to the extended affine point $(X/Z, Y/Z, T/Z)$ with $Z \neq 0$. Obviously, the auxiliary coordinate T has the property that $T = XY/Z$ (see [13]). Given two "extended" points $P_1 = (X_1 : Y_1 : T_1 : Z_1)$ and $P_2 = (X_2 : Y_2 : T_2 : Z_2)$ with $Z_1 \neq 0$ and $Z_2 \neq 0$, a dedicated addition, derived from Equation 5, can be performed as $(X_1 : Y_1 : T_1 : Z_1) + (X_2 : Y_2 : T_2 : Z_2) = (X_3 : Y_3 : T_3 : Z_3)$ where

$$\begin{cases} X_3 = (X_1Y_2 - Y_1X_2)\,(T_1Z_2 + Z_1T_2) \\ Y_3 = (Y_1Y_2 + aX_1X_2)\,(T_1Z_2 - Z_1T_2) \\ T_3 = (T_1Z_2 + Z_1T_2)\,(T_1Z_2 - Z_1T_2) \\ Z_3 = (Y_1Y_2 + aX_1X_2)\,(X_1Y_2 - Y_1X_2) \end{cases} \quad (7)$$

When $P_1 = P_2$, one can obtain the formula for point doubling in extended twisted Edwards coordinates as follows.

$$\begin{cases} X_3 = 2X_1Y_1\left(2Z_1^2 - Y_1^2 - aX_1^2\right) \\ Y_3 = \left(Y_1^2 + aX_1^2\right)\left(Y_1^2 - aX_1^2\right) \\ T_3 = 2X_1Y_1\left(Y_1^2 - aX_1^2\right) \\ Z_3 = \left(Y_1^2 + aX_1^2\right)\left(2Z_1^2 - Y_1^2 - aX_1^2\right) \end{cases} \quad (8)$$

Note that the above formulas are independent of the parameter d and can be further simplified if $a = -1$, which is the case for our curve. When P_2 is given in affine coordinates (i.e. $Z_2 = 1$), the addition (which is actually a mixed addition) can be carried out using only seven multiplications (7M) in the underlying field (not taking into account the multiplication by a). On the other hand, the point doubling costs four multiplications (4M), four squarings (4S), and a

multiplication by a, the latter of which can be neglected in our case since $a = -1$.

As explained in [13, Sect. 4.3], it is possible to save one field-multiplication in the point doubling by mixing extended with conventional (i.e. non-extended) twisted Edwards coordinates. This cost reduction is based on the observation that the auxiliary coordinate T_1 of P_1 is only used in the point addition, but not in the point doubling described by Equation (8). Consequently, the computation of T_3 in Equation (8) can be omitted if a point doubling is followed by another point doubling. The same holds for the point addition: T_3 in Equation (7) does not need to be computed if the subsequent operation is a point doubling. In order to optimize the point doubling, we eliminate the field-multiplication that produces T_3 in Equation (8) and output the two factors $E_3 = 2X_1Y_1$ and $H_3 = Y_1^2 - aX_1^2$ it is composed of instead. When doing so, the resulting point $P_3 = 2P_1$ consists of five coordinates instead of four, which means P_3 is actually represented by a quintuple of the form $(X_3 : Y_3 : E_3 : H_3 : Z_3)$. In other words, the auxiliary coordinate T is split up into E and H such that $EH = T = XY/Z$, thereby saving a field multiplication in the point doubling. The subsequently-executed operation can recover $T3$, if needed, by simply multiplying E_3 by H_3. Of course, this optimization of the doubling makes it necessary to modify the point addition accordingly. We implemented the addition formula shown in Equation (7) to output the two factors $E_3 = T_1Z_2 + Z_1T_2$ and $H_3 = T_1Z_2 - Z_1T_2$ instead of $T_3 = E_3H_3$. When performing an addition using P_1 represented by $(X_1 : Y_1 : E_1 : H_1 : Z_1)$ as input, the auxiliary coordinate $T_1 = E_1H_1$ has to be computed first since it is needed as operand. Note, however, that this modification does not change the overall cost of a point addition since the computation of $T_3 = E_3H_3$ is simply replaced by forming the product $T_1 = E_1H_1$. Putting everything together, the optimized implementation of point doubling requires 3M and 4S in the underlying field, while a point addition costs 7M.

Algorithm 1 Point addition on TE curve with $a = -1$.

Input: Point $P_1 = (X_1 : Y_1 : E_1 : H_1 : Z_1)$ with $E_1H_1 = T_1 = X_1Y_1/Z_1$ and $P_2 = (X_2 : Y_2 : T_2)$ with $T_2 = X_2Y_2$.
Output: Sum $P_3 = P_1 + P_2 = (X_3 : Y_3 : E_3 : H_3 : Z_3)$.
1: $T_1 \leftarrow E_1 \cdot H_1$
2: $A \leftarrow (Y_1 - X_1) \cdot (Y_2 + X_2)$
3: $B \leftarrow (Y_1 + X_1) \cdot (Y_2 - X_2)$
4: $C \leftarrow 2 \cdot Z_1 \cdot T_2$
5: $D \leftarrow 2 \cdot T_1$
6: $E_3 \leftarrow D + C$
7: $H_3 \leftarrow D - C$
8: $F \leftarrow B - A$
9: $G \leftarrow B + A$
10: $X_3 \leftarrow E_3 \cdot F$
11: $Y_3 \leftarrow G \cdot H_3$
12: $Z_3 \leftarrow F \cdot G$
13: **return** $(X_3 : Y_3 : E_3 : H_3 : Z_3)$

Algorithm 1 specifies the sequence of field operations for a mixed addition where P_1 is given in extended projective coordinates of the form $(X_1 : Y_1 : E_1 : H_1 : Z_1)$ with $E_1H_1 = T_1 = X_1Y_1/Z_1$. The second point P_2, which is usually the base point of a scalar multiplication, is represented by extended affine coordinates $(X_2 : Y_2 : T_2)$ with $T_2 = X_2Y_2$. Algorithm 2 shows the field operations needed to double a point P_1 using our "special" extended projective coordinates. Both algorithms are optimized for twisted Edwards curves with parameter $a = -1$. Note that, when using extended coordinates, both the addition and doubling formulae are independent of the curve parameter d [13].

Algorithm 2 Point doubling on TE curve with $a = -1$.

Input: Point $P_1 = (X_1 : Y_1 : E_1 : H_1 : Z_1)$.
Output: Double $P_3 = 2 \cdot P_1 = (X_3 : Y_3 : E_3 : H_3 : Z_3)$.
1: $A \leftarrow X_1^2$
2: $B \leftarrow Y_1^2$
3: $C \leftarrow 2 \cdot Z_1^2$
4: $H_3 \leftarrow A + B$
5: $E_3 \leftarrow (X_1 + Y_1)^2 - H_3$
6: $G \leftarrow B - A$
7: $F \leftarrow C - G$
8: $X_3 \leftarrow E_3 \cdot F$
9: $Y_3 \leftarrow G \cdot H_3$
10: $Z_3 \leftarrow F \cdot G$
11: **return** $(X_3 : Y_3 : E_3 : H_3 : Z_3)$

There exist a number of algorithms for computing a scalar multiplication $k \cdot P$ through point additions and doublings; see e.g. [12] for an overview. The most basic algorithm is the so-called double-and-add method, which requires to perform n point doublings and roughly $n/2$ point additions when the scalar k has a length of n bits. The number of point additions can be reduced to roughly $n/3$ when k is represented in Non-Adjacent Form (NAF) [12]. Of course, there exist several faster scalar multiplication techniques, but they either rely on the pre-computation of multiples of P (which costs memory) or can only be used when P is fixed and known a-priori. Since we aim for a "lightweight" implementation with low memory footprint, we decided to adopt the double-and-add method with NAF-representation of the scalar k.

4. IMPLEMENTATION RESULTS

We have implemented scalar multiplication on a twisted Edwards curve for 8-bit AVR processors such as the ATMega128. The results we describe in this section have been obtained using a 160-bit OPF, but the order of the field can be easily extended to e.g. 192, 224, or even 256 bits.

Atmel's AVR Studio provides a cycle-accurate simulation system for 8-bit micro-controllers like the ATmega128, which is contained in the MICAz and IRIS sensor nodes. One can easily obtain detailed information about the efficiency of a program by analyzing the running time (μs) or the number of clock cycles of a certain function or block. Therefore, we use AVR Studio to profile and simulate the execution time of our ECC implementations and the TinyECC library [18].

4.1 Execution Times

We started with simulating the underlying field arithmetic operations of TinyECC and our implementation. Our simulation results of the OPF arithmetic are essentially the same as that reported in Zhang's paper [24]. These execution times are summarized in Table 1, along with the execution time of inversion, which we implemented from scratch since it was not part of Zhang's OPF library. The timings of the

corresponding arithmetic operations of TinyECC are also specified. To get accurate cycle counts, we executed each function several times in a loop and use the average value as result.

Operation	Our work	TinyECC
int_add	166	321
int_sub	166	313
int_mul	3006	3390
int_sqr	2428	3390
mod_add	531	637
mod_sub	531	682
mod_mul	3588	6098
mod_sqr	3032	5725
mod_inv	356650	861901

Table 1: Execution time (in clock cycles) of arithmetic operations in a 160-bit prime field

As specified in Table 1, a multiplication (including modular reduction) in a 160-bit OPF takes 3588 clock cycles, which is only 582 cycles more than a conventional (160×160)-bit multiplication without modular reduction. OPF-squaring is about 15% faster than multiplication in an OPF, whereas addition and subtraction need only one sixth of the multiplication cycles. Inversion is more than 100 times slower than multiplication. Compared with the field arithmetic operations of TinyECC, our OPF modular multiplication is 1.7x faster, modular squaring is 1.9x faster, while inversion is 2.4x faster.

Next, we simulated the point arithmetic operations of our implementation, which uses a twisted Edwards curve over a 160-bit OPF, and compared it with the point arithmetic of TinyECC. The execution times of point addition and doubling are given in Table 2

Operation	Our work	TinyECC
point addition	28867	59070
point doubling	26069	48483

Table 2: Execution time (in cycles) of point addition and point doubling in a 160-bit EC group

Both the point addition and doubling of our implementation are much faster than that of TinyECC, which is not only due to the advanced field arithmetic, but also because of the more efficient addition/doubling formulae. In detail, the point addition of our implementation outperforms TinyECC by a factor of more than two, while point doubling is about 1.86x faster.

To assess the execution time of a scalar multiplication $k \cdot P$, we simulated our implementation as well as TinyECC for three different 160-bit values for the scalar k. When using the double-and-add method, the number of point additions is not constant but depends on the number of non-zero digits in the NAF representation of k. In the first simulation we used $k = 2^{159} =$ 0x80...00, which is a 160-bit scalar with the minimum Hamming weight, yielding the best possible execution time. The second simulation was performed with a random value of k such that the number of point additions is exactly one third of the bitlength of k, which corresponds to the average case in terms of execution time. Finally, we used $k = 2^{160} - 1 =$ 0xff...ff (i.e. maximum Hamming weight) as scalar to assess the worst-case execution time. The results are summarized in Table 3.

Scalar k	Our work	TinyECC
$k =$ `0x80...00`	4408966	11314866
random integer	5920717	16489433
$k =$ `0xff...ff`	6746829	23237223

Table 3: Best-case, average, and worst-case execution time of 160-bit scalar multiplication

The results from Table 3 clearly show that, for all three different values of k, our implementation is about 2.78x faster than TinyECC. Obviously, the overall improvement of our implementation is bigger than the performance gain due to the underlying field arithmetic. Consequently, not only the field arithmetic but also the point arithmetic on our twisted Edwards curve is more efficient than that used by TinyECC. In fact, the contribution of the OPF arithmetic to the overall speed-up is about 68%, while the advanced point addition/doubling formulae contribute roughly 32%.

4.2 Energy Evaluation

In this subsection, we aim to estimate the number of ECDH key exchange operations a MICAz or IRIS mote can accomplish before running out of battery. For this, we need to first figure out the power (resp. energy) consumption characteristics of the MICAz mote and its micro-controller, the ATmega128. According to [7], the ATmega128 processor of a MICAz mote draws an average current of 8`mA` (at a supply voltage of 3.0`V`) when its is active. Since the clock frequency of the mote is known to be 7.3728`MHz`, we can get the energy consumption of one scalar multiplication of our implementation by a simple calculation of the form $W = U \cdot I \cdot t$, whereby U denotes the supply voltage (i.e. 3`V` when using two conventional 1.5`V` AA batteries), I the average current drawn by the processor (i.e. 8`mA` in our case), and t the execution time. In our implementation, we have an average execution time of 5920717 clock cycles, which means the energy cost of computing a single scalar multiplication amounts to $W_c = U \cdot I \cdot t = 3\texttt{V} \cdot 8\texttt{mA} \cdot (5920717/7.3728 \cdot 10^6) =$ 19.27`mJ`. An ECDH key exchange requires each node to compute two scalar multiplications and to send a message (containing the public key) to the other node. According to the energy model described in [8], the energy cost of transmitting a protocol message is $W_t = P \cdot t =$ 0.185`mJ`. Consequently, the overall energy cost of ECDH key exchange is primarily determined by the computation energy W_c for two scalar multiplications; the communication energy W_t is essentially negligible, which was already found before in [17]. Putting everything together, the total energy consumption of performing an ECDH key exchange is $W = 2 \cdot W_c + W_t =$ 38.73`mJ` per node.

Piotrowski et al state in [22] that the rated capacity of a 1.5`V` AA alkaline battery is about 2500`mAh` and, accordingly, two AA batteries can theoretically deliver an energy of 21600`Ws`. However, the ATmega128 requires a supply voltage of at least 2.7`V`, which means according to [22] that "the node powered by two AA alkaline batteries uses only 31.25% of the total capacity, i.e., the node can consume about 6750 `Ws` until the batteries are useless." Consequently, we can perform $6750/(38.73 \cdot 10^{-3}) = 174278$ ECDH key exchanges

(using 160-bit keys) before the supply voltage of the MICAz mote drops below 2.7V.

5. CONCLUSIONS

We presented a highly-optimized implementation of elliptic curve cryptography for 8-bit AVR processors such as used in the MICAz mote and various other sensor nodes. Our software is able to perform a full 160-bit scalar multiplication in only $5.9 \cdot 10^6$ clock cycles on average (i.e. 0.8 seconds on a MICAz), which is about 2.78 times faster than TinyECC. We achieved this execution time by combining OPFs (a special type of prime fields) with twisted Edwards curves, a relatively new family of elliptic curve allowing for fast point arithmetic. TinyECC, on the other hand, uses standardized curves and fields that were devised some 15 years ago and do not represent the state of the art anymore. Roughly two third of the speed-up is due to the more efficient field arithmetic and about one third is contributed by the fast point addition/doubling formulae of twisted Edwards curves. Going along with this performance gain is a reduction of the energy cost of ECC by approximately the same factor. For example, an ECDH key exchange using our ECC software requires only one third of the energy of the ECDH implementation included in TinyECC. As a consequence, a conventional MICAz mote is able to perform over 174,000 ECDH key exchanges before running out of battery. Our work makes ECDH key exchange more attractive for wireless sensor networks since high energy consumption is generally considered the most significant drawback of ECDH and other ECC-based key establishment techniques.

6. 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 M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, editors, *Public Key Cryptography — PKC 2006*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer Verlag, 2006.

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

[4] D. J. Bernstein and T. Lange. Faster addition and doubling on elliptic curves. In K. Kurosawa, editor, *Advances in Cryptology — ASIACRYPT 2007*, volume 4833 of *Lecture Notes in Computer Science*, pages 29–50. Springer Verlag, 2007.

[5] D. W. Carman, P. S. Kruus, and B. J. Matt. Constraints and Approaches for Distributed Sensor Network Security. Technical Report #00-010, NAI Labs, Network Associates, Inc., Glenwood, MD, USA, Sept. 2000.

[6] H. Cohen and G. Frey. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*, volume 34 of *Discrete Mathematics and Its Applications*. Chapmann & Hall\CRC, 2006.

[7] Crossbow Technology, Inc. MICAz Wireless Measurement System. Data sheet, available for download at `http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICAz_Datasheet.pdf`, Jan. 2006.

[8] G. de Meulenaer, F. Gosset, F.-X. Standaert, and O. Pereira. 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)*, pages 580–585. IEEE Computer Society Press, 2008.

[9] H. M. Edwards. A normal form for elliptic curves. *Bulletin of the American Mathematical Society*, 44(3):393–422, July 2007.

[10] J. Großschädl. TinySA: A security architecture for wireless sensor networks. In C. Diot, M. Ammar, C. Sá da Costa, R. J. Lopes, A. R. Leitão, N. Feamster, and R. Teixeira, editors, *Proceedings of the 2nd International Conference on Emerging Networking Experiments and Technologies (CoNEXT 2006)*, pages 288–289. ACM Press, 2006.

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

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

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

[14] Institute of Electrical and Electronics Engineers (IEEE). IEEE Std 1363-2000: IEEE Standard Specifications for Public-Key Cryptography, Aug. 2000.

[15] M. K. Jain. Wireless sensor networks: Security issues and challenges. *International Journal of Computer and Information Technology*, 2(1):62–67, July 2011.

[16] Ç. K. Koç, T. Acar, and B. S. Kaliski. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, June 1996.

[17] 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 O. Markowitch, A. Bilas, J.-H. Hoepman, C. J. Mitchell, and J.-J. Quisquater, editors, *Information Security Theory and Practice — WISTP 2009*, volume 5746 of *Lecture Notes in Computer Science*, pages 112–127. Springer Verlag, 2009.

[18] 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)*, pages 245–256. IEEE Computer Society Press, 2008.

[19] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, Apr. 1985.

[20] National Institute of Standards and Technology (NIST). Recommended Elliptic Curves for Federal Government Use. White paper, available for download at `http://csrc.nist.gov/encryption/dss/ecdsa/NISTReCur.pdf`, July 1999.

[21] A. Perrig, J. A. Stankovic, and D. Wagner. Security in wireless sensor networks. *Communications of the ACM*, 47(6):53–57, June 2004.

[22] K. Piotrowski, P. Langendörfer, and S. Peter. How public key cryptography influences wireless sensor node lifetime. In S. Zhu and D. Liu, editors, *Proceedings of the 4th ACM Workshop on Security of Ad Hoc and Sensor Networks (SASN 2006)*, pages 169–176. ACM Press, 2006.

[23] Y. Wang, G. Attebury, and B. Ramamurthy. A survey of security issues in wireless sensor networks. *IEEE Communications Surveys & Tutorials*, 8(2):2–23, Apr. 2006.

[24] Y. Zhang and J. Großschädl. Efficient prime-field arithmetic for elliptic curve cryptography on wireless sensor nodes. In *Proceedings of the 1st International Conference on Computer Science and Network Technology (ICCSNT 2011)*, volume 1, pages 459–466. IEEE, 2011.