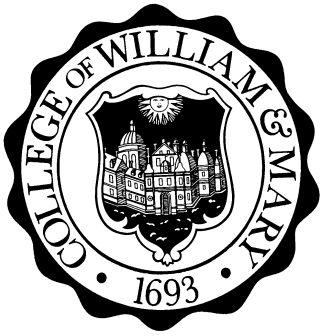

College of
William & Mary
Department of Computer Science

WM-CS-2007-11

WM-ECC: an Elliptic Curve Cryptography Suite on Sensor Motes

Haodong Wang, Bo Sheng, Chiu C. Tan and Qun Li

Oct. 30, 2007

Technical Report

WM-CS-2007-11

WM-ECC: an Elliptic Curve Cryptography Suite on Sensor Motes

Haodong Wang, Bo Sheng, Chiu C. Tan and Qun Li
Department of Computer Science
College of William and Mary
Williamsburg, VA, 23187
E-mail:{wanghd, shengbo, cct, liqun}@cs.wm.edu

Oct. 30, 2007

Abstract

While symmetric-key schemes, which have been investigated extensively for sensor network security, can fulfill many security requirements, public-key schemes are more flexible, and provide a clean interface for security components. In contradiction of the popular belief that public key scheme is not practical for sensor networks, this technical report describes the ECC (Elliptic Curve Crytography) public-key cryptosystem implementation on the existing commercial sensor devices. We detail the implementation of 160-bit ECC cryptosystems over prime field on MICAz, TelosB and Tmote Sky sensor motes. We evaluate the performance of our implementation by running digital signature generation and verification. We have achieved the performance of 0.77s for signature generation and 1.12s for signature verification on Tmote Sky sensor motes. Comparatively, we show the performance on MICAz and TelosB motes are 1.35s and 1.45s for signature generation, 1.96 and 2.25 for signature verification, respectively. This technical report summarize our previous implementation effort presented in [11, 12, 13, 10].

1 Introduction

Public-key cryptography has been used extensively in data encryption, digital signature, user authentication, and so on. Compared with the symmetric-key based schemes proposed for sensor networks, public-key schemes provide a more flexible and simple interface requiring no complicated key pre-distribution, no pair-wise key sharing negotiation. It is a popular belief, however, in sensor network research community that public-key cryptography is not practical because the required computational intensity is not suitable for sensors with limited computation capability and extremely constrained memory space. The recent progress in ECC implementation on Atmel ATmega128, a CPU of 8MHz and 8 bits[3], however, shows that a public key operation takes less than one second, which suggests public-key cryptography is feasible for sensor network security related applications.

This technical report details our implementation of 160-bit ECC cryptosystem on three commercial off-the-shelf sensor motes: MICAz, TelosB and Tmote Sky, which are the size of two AA batteries integrating USB programming capability, an IEEE 802.15.4/ZigBee Compliant radio with integrated antenna. The MICAz mote features a 8-bit, 8MHz Atmel microcontroller with 4KB RAM, 128KB programmable ROM, and optional external memory for data collection. The TelosB and Tmote Sky share the same hardware platform, they both are equipped with a 16-bit, 8MHz TI MSP430 processor with 10KB RAM, 48KB programmable ROM and 1MB on-board flash memory for data collection.

The most expensive operations ECC cryptosystems are large integer arithmetics over the finite field. To efficiently perform ECC exponentiations on the low-power CPU of sensor motes, it is essential to optimize the expensive large integer operations. In particular, multiplication and reduction are most dominant operations in ECC. Thus the efficiency of these two integer operation modules directly determines the performance of the encryption and decryption. The low-power sensor microcontrollers in the above sensor platforms have very limited number of registers (only 32 8-bit registers in ATmega, 12 16-bit registers in MSP430). The large integer operands cannot be loaded into the registers at one time, so that the latency of memory accesses has to be paid for operands loading and storing between registers and memory. The implementation challenge is to reduce the number of such memory accesses. In our implementation, we adopt the hybrid multiplication method [3], which is a very effective way to reduce the number of memory accesses for resource constrained micro-controllers. To precisely control the register and memory operations, we implement this module in assembly language. Our experiments demonstrate that the hybrid multiplication is at least 7 times faster than the conventional multi-precision multiplication programmed in C language. The modular reduction can also be optimized under certain conditions. For example, when the modulus is a pseudo-Mersenne number, the reduction can be greatly optimized to be finished more than 10 times faster than the classic long division method.

In addition, our implementation has also adopted some other optimization techniques. For example, we apply a mixed coordinate, the combination of Affine coordinate and Jacobian coordinate, to do ECC exponentiation, so that some expensive operations can be avoided (e.g., inversion) or reduced (e.g., multiplication and squaring). The rest of optimizations include Sliding-Window method [5], Non-adjacent Form [8], and Shamir's trick.

Our experiments show that ECC can be efficiently run on MICAz motes. It takes 1.35s to generate a signature, and 1.96s to perform a signature verification. Our comparison tests further show that ECC is even more efficient on Tmote Sky by taking the advantage of 8MHz and 16-bit CPU. The signature generation and verification on Tmote are 0.77s and 1.12s, respectively. Since TelosB mote, sharing the same hardware with Tmote, can only run at 4MHz, the performance on TelosB is 1.54s for signature and 2.25s for verification. Overall, our experiment results demonstrate that ECC is feasible for sensor network security applications.

The rest of the technical report is organized as follows. Section 2 briefly introduces ECC public key schemes. Section 3 gives detailed description of several most important optimizations in large integer operation, as well as some specific optimizations designed for ECC implementations exclusively. Section 4 evaluates the performance of our implementations. Section 5 concludes this technical report.

2 ECC Introduction

In this section, we give some background on elliptic curve cryptography, and the corresponding elliptic curve Digital Signature Algorithm.

2.1 Elliptic Curve Cryptography

In recent years, ECC has attracted much attention as the security solutions for wireless networks due to the small key size and low computational overhead. For example, 160-bit ECC offers the comparable security to 1024-bit RSA. An elliptic curve over a finite field GF (a Galois Field of order q) is composed of a finite group of points (x_i, y_i), where integer coordinates x_i, y_i satisfy the long Weierstrass form:

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6, \tag{1}$$

and the coefficients a_i are elements in $GF(q)$. Since the field $GF(q)$ (q is a prime) is generally used in cryptographic applications, (1) can be simplified to:

$$y^2 = x^3 + ax^2 + b, \tag{2}$$

where $a, b \in GF(q)$.

The elliptic curve points form an additive abelian group, so that the addition of any two points is a point in the group. Given two points P and Q, with the coordinates (x_1, y_1), (x_2, y_2), respectively, the addition results in a point R on the curve with coordinate (x_3, y_3), where x_3 and y_3 satisfy

$$(x_1, y_1) + (x_2, y_2) = (x_3, y_3), \tag{3}$$

such that

$$x_3 = L^2 + L + x_1 + x_2 + a, \tag{4}$$

$$y_3 = L(x_1 + x_3) + x_3 + y_1, \tag{5}$$

where

$$L = (y1 + y2)/(x1 + x2) \tag{6}$$

If $x_1 = x_2$ (note $x_1 + x_2$ is 0), then R is defined as a point at infinity, O. O is an identity element of the group. Each element in the group has an inverse that satisfies $P + (-P) = O$, and $(-P) + P = O$. Also, $P + O = O + P = P$. If $P = Q$, then $R = P + P = 2P$, and coordinate (x_3, y_3) is derived by

$$x_3 = L^2 + L + a, \tag{7}$$

$$y_3 = {x_1}^2 + (L + 1)x_3, \tag{8}$$

where

$$L = x_1 + y_1/x_1. \tag{9}$$

The ECC relies on the difficulty of the Elliptic Curve Discrete Logarithm Problem, that is, given points P and Q in the group, it is hard to find a number k such that $Q = kP$.

2.2 Elliptic Curve Digital Signature Algorithm (ECDSA)

ECC signature is based on Digital Signature Algorithm. We assume Alice sends a message to Bob. To convince Bob that the message does come from Alice, Alice needs to apply a digital signature for the message so that Bob can verify it by using Alice's public key. Initially, Alice and Bob have to agree on a particular curve with base point P over the field $GF(p)$, and the order of P is q. When Alice sends a message to Bob, she attaches a digital signature (r, s) generated by following steps (suppose Alice has a private key x and a public key $Q = xP$).

1. Choose a random key k in $[1, q-1]$;
2. Compute kP, yield a point with coordinate (x_1, y_1). Let $r = x_1 \pmod q$. Check r, go back to the first step if the result is zero;
3. Compute $k^{-1} \pmod q$;
4. Compute $s = k^{-1}(Hash(m) + xr)$, where $Hash$ is a one-way hash function. Again, check s, go back to the first step if $s = 0$;

5. (r,s) is the digital signature.

To verify the message m and the signature, Bob needs to do following steps.

1. Compute $w = s^{-1} \bmod q$ and $H(m)$;
2. Compute $u_1 = H(m) \cdot w \bmod q$ and $u_2 = r \cdot w \bmod q$;
3. Compute $u_1 P + u_2 Q$, and get the resulting point (x_2, y_2);
4. The signature is verified if $x_2 = r$.

Finally, Bob compares the value of x_2 and r, and accepts the message only if x_2 equals to r.

3 Implementation

In this section, we take the MICAz mote as an example to describe our ECC implementation. MICAz is powered by an ATmega128 microcontroller. The ATmega128 incorporates an 8MHz, 8-bit RISC CPU, 128K bytes programmable flash memory (ROM) and 4K bytes SRAM. This architecture provides 133 powerful instructions and 32×8 general purpose registers. Besides, ATmega128 also features an on-chip multiplier. The fundamental ECC operation is large integer arithmetics over either prime number finite field $GF(p)$ or binary polynomial field $GF(2^m)$ (where m is a prime). Because the two heavily used operations: multiplication and modular reduction, can be more effectively optimized if pseudo-Mersenne primes are picked for elliptic curves compared with those of binary field [3], we limit our discussion in prime number finite field $GF(p)$ in this paper. Without further clarification, our discussion of ECC implementation is based on SECG recommended 160-bit elliptic curve: secp160r1. We first describe the optimized large integer operation modules. Then we focus on the protocol related optimizations specifically for ECC operation.

3.1 Large Integer Operations

We implement a suite of large integer arithmetic operations, including addition, subtraction, shift, multiplication, division and modular reduction. Due to space limitation, we only present three of most important functions: multiplication (with squaring), division and modular reduction.

3.1.1 Multiplication

The multiplication is the key component in ECC implementation because the exponentiation is basically computed by multiplications and squaring. We have compared three different multiplication implementations [3, 7, 6], and finally decided to use hybrid multiplication proposed in [3]. To ease our explanation, we use three large integers as the examples for our following discussion: $A(a_{n-1}, a_{n-2}, \cdots, a_1, a_0)$, $B(b_{n-1}, b_{n-2}, \cdots, b_1, b_0)$, and $C(n_{2n-1}, c_{2n-2}, \cdots, c_1, c_0)$, where $C = A * B$. A and B both have length of n words, each word has k-bit size. The product C has $2n$ words.

The hybrid multiplication is the combination of Row-wise multiplication and Column-wise multiplication. The Row-wise method fixes the multiplier b_i $(0 \leq i \leq n)$, and multiplies it with every word of multiplicand A. Partial results are stored in $n+1$ accumulator registers. Every time one row is finished, the last accumulator register can be stored to memory as the part of final results. On average, one memory load is required for each $k \times k$ multiplication. When integer size n is increased, the required number registers

increase linearly in Row-wise method. For 160-bit ECC, a typical multiplication is between two 20-byte large integers. Given only 32 registers in ATmega128, Row-wise multiplication can not be directly applied.

The column-wise method, on the other side, computes the partial results of $a_i * b_j$ (where $i + j = l$) for column l. After one column finishes, the last word of accumulator registers is stored as the part of final result. The column-wise method only requires three accumulator registers and two more for operands. However, two memory load operations are required for each $k \times k$ multiplication.

The hybrid method takes advantages of row-wise and column-wise strategies. To optimize the memory operation, the hybrid method merges a number (d) of columns together, and then conducts row-wise multiplication in each merged column. When d equals to 1, the hybrid method becomes the column-wise multiplication. When d equals to n, it becomes row-wise method. A larger d leads to fewer memory operations, but requires more registers. A small d, however, requires more memory operations and consumes more CPU cycles. Balancing the advantages and disadvantages, we implement the Hybrid multiplication with column width $d = 4$, which requires 9 accumulator registers, 5 operand registers, 6 pointer registers (point to A, B and C), and others for temporary storage and loop control.

We implement the Hybrid multiplication in assembly language. For the comparison purpose, we also implement a standard multi-precision multiplication program in C language. Our experiments show the standard C program needs $122.2ms$ to finish the multiplication between two 128-byte integers, while it only takes $17.6ms$ for our Hybrid multiplication to do the same computation, which is more than 7 times faster.

The squaring is a special case of the multiplication, which has the same the multiplicand and the multiplier. Given an m-bit large integer $A = (A_1, A_0)$, where A_1, A_0 are two halves, $A^2 = A_1A_1 \times 2^m + 2A_1A_0 \times 2^{m/2} + A_0A_0$. Therefore, we can take advantage of the fact that A_1A_0 only needs to be calculated once. Compared with the multiplication, the optimized squaring can reduce the computational complexity up to 25%.

3.1.2 Modular Division

Modular division is another expensive operation in ECC. In Affine coordinate, each ECC operation of point addition and doubling requires a modular inversion. The integer inversion is also required for ECC digital signature generation and verification. In our implementation, we adopt the Great Divide scheme proposed in [9]. We briefly explain the algorithm in the followings. Given an denominator x and numerator y, we want to compute the modular division $\frac{y}{x}$ over $GF(p)$. This is equivalent to find r, so that

$$r \equiv \frac{y}{x} (\text{mod q}). \tag{10}$$

To find r efficiently, the algorithm maintains following two invariant relationship:

$$A * y \equiv U * x, \text{ and } B * y \equiv V * x, \tag{11}$$

where $A, B, U,$and V are four auxiliary variables and initialized with values $x, q, y,$ and 0, respectively. Note the two relationship is true with the initial values. The algorithm intuition is to reduce the value of A to 1, so that the first relationship in (11) will become $y \equiv U * x$, and U will be the result. The procedure is conducted in following way. When A is even, we can divide A by 2. Correspondingly, U has to be divided by 2 to keep the relation true. If U is not even at that time, we can make it become even by adding U with the modulus. When A is odd, we use the 2nd relationship to help to reduce A. If B is even, we keep dividing B by 2 similarly to make B odd. Then we add the two relation together and the divide the result value by 2 at the both sides. By repeating this process, it is guaranteed that either value of A or B reduces one bit

in one iteration. The procedure stops when $A = B = 1$, the first equation becomes $y \equiv U * x$. The value of U is our final result. If we initialize U with 1, this routine can be used to calculate an inversion of x. This algorithm works when x and q are relatively prime. The Great Divide finishes division or inversion operation in $2(log(x) - 1)$ steps. Great Divide is much faster than the long division method because Great Divide only needs addition operations in each iteration, while long division method requires multiplications.

3.1.3 Modular Reduction

The modular reduction operation is another important module because each multiplication or squaring must be followed by a reduction operation. The classic reduction method is using long division. Although the long division method is a general method for calculating the modular reduction, it is also the slowest method. In ECC cryptosystem, the modular reduction operation is as important as modular multiplication. Each multiplication must be followed by a reduction operation. Since we choose to use pseudo-Mersenne primes as specified in NIST/SECG curves, the modular reduction can be optimized by conducting a fixed number of integer additions. Because the optimization is curve specific, we will explain in more details in the next subsection of ECC operation.

Now, we discuss the modular reductions in ECC digital signature generation and verification. In most cases, the modulus is not a pseudo-Mersenne prime, the optimization cannot be applied for those reduction calculations. We choose the classic long division method to implement this operation. Fortunately, the number of this type of modular reduction is very limited, it does not affect the overall performance much. We briefly describe the long division method as in Algorithm 1. The long division producer reduces the

Algorithm 1 Reduction by using long division.

1: Input: x, n;
2: Output: $r = x \bmod n$;
3: **while** $x \geq n$ **do**
4: Align the most significant byte (MSB) of modulus n to the MSB of x, the lower bytes of n can be filled with zeros;
5: Starting with the MSB of x, divide the first two MSBs of x by the MSB of modulus n, and get the quotient;
6: Multiply the quotient with the modulus and get a subproduct;
7: If the subproduct is greater than the remainder of x (over estimation), subtract the modulus from the subproduct;
8: Then subtract the subproduct from the remainder of x;
9: The procedure continues and goes back to step 2 if the MSB of the remainder becomes zero;
10: If the MSB of the remainder is not zero (under estimation), subtract the modulus from the remainder, and then go back to step 2;
11: The procedure stops when the remainder is less than modulus n;
12: **end while**
13: return x;

remainder of x by one byte in each iteration.

3.2 Optimization for ECC Operation

We first discuss ECC point addition and doubling. We then introduce an optimized modular reduction for curve secp160r1. Finally, we explain several different optimizations for point multiplication.

3.2.1 ECC addition and doubling

The fundamental ECC operation is point addition and point doubling. The point multiplication can be decomposed to a series of addition and doubling operations. As discussed in previous section, point addition and doubling in Affine coordinate require integer inversion, which is considered much slower than integer multiplication. Cohen *et al.* showed that these operations in Projective coordinate and Jacobian coordinate yield better performance [1]. They further found addition and doubling in mixed coordinate, with the combination of Modified Jacobian coordinate and Affine coordinate, lead to the best performance [2]. Consider an ECC point in Modified Jacobian coordinate, $P_1(X_1, Y_1, Z_1, aZ_1^4)$, and a point in Affine coordinate, $P_2(x_2, y_2)$, their addition results in the third point $P_3 = (X_3, Y_3, Z_3, aZ_3^4)$ in Modified Jacobian coordinate. The result is given by following equations.

$$\begin{aligned} X_3 &= -H^3 - 2X_1H^2 + r^2, \\ Y_3 &= -Y_1H^3 + r(X_1H^2 - X_3), \\ Z_3 &= Z_1H, \\ aZ_3^4 &= aZ_3^4, \end{aligned} \tag{12}$$

where $H = x_2Z_1^2 - X_1$, and $r = y_2Z_1^3 - Y_1$. The result of point doubling for $P_3 = 2P_1$ is given by following formula.

$$\begin{aligned} X_3 &= T, \\ Y_3 &= M(S-T) - U, \\ Z_3 &= 2Y_1Z_1, \\ aZ_3 &= 2U(aZ_1^4) \end{aligned} \tag{13}$$

To estimate the computational complexity, we only consider large integer multiplication and squaring operations, and ignore those addition and subtraction since they are much faster. According to Eq.12 and Eq.13, point addition requires 9 large integer multiplications and 5 squaring, and point doubling requires 4 multiplications and 5 squaring.

The basic point operations can be further optimized for specific elliptic curves. In our case, the curve parameter a of secp160r1 equals to -3. For point doubling, M can be further reduced to

$$M = 3X_1^3 - 3Z_1^4 = 3(X_1 + Z_1^2)(X_1 - Z_1^2). \tag{14}$$

As the result, point doubling operation reduces to 4 multiplications and 4 squaring. Actually, aZ_3^4 does not have to be calculated in point addition, so the computational complexity reduces to 8 multiplications and 3 squaring. Our observation supports the choice of mixed coordinate, the performance of point multiplication improves around 6% compared with our previous implementation in Jacobian coordinate.

3.2.2 Modular Reduction on ECC Curve

Recall that modular reduction has to be applied after every large integer multiplication, it is also a performance critical operation. By taking advantage of pseudo-Mersenne primes specified in SECG curves, the

complexity of the modular reduction operation can be reduced to a negligible amount. In this section, we use curve secp160r1 as the example to show how to do efficient reduction.

Suppose we use the 8-bit architecture, the multiplication result of two 160-bit integers can be represented by

$$C(c_{39},\cdots,c_{20},c_{19},\cdots,c_1,c_0),$$

where c_i ($0 \leq i \leq 39$) is a word with 8 bits, and c_{39} is the most significant word. The 40-word integer can also be written as:

$$C = (c_{39},\cdots,c_{20}) * 2^{160} + (c_{19},\cdots,c_1,c_0) \tag{15}$$

Given the field of curve secp160r1 $q = 2^{160} - 2^{31} - 1$, we can have $2^{160} \equiv 2^{31} + 1$. Therefore,

$$\begin{aligned} C &\equiv (c_{39},\cdots,c_{20}) * (2^{31}+1) + (c_{19},\cdots,c_1,c_0) \\ &\equiv (c_{39},\cdots,c_{20}) * 2^{31} + (c_{39},\cdots,c_{20}) + (c_{19},\cdots,c_1,c_0) \end{aligned} \tag{16}$$

Since each word has 8 bits, the first term in the result of Eq. 16 can be further reduced to

$$\begin{aligned} &(c_{39},\cdots,c_{20}) * 2^{31} \equiv (c_{39},c_{38},c_{37}) * 2^{167} + c_{36} * 2^{159} + (c_{35},\cdots,c_{20}) * 2^{31} \\ &\equiv (c_{39},c_{38},c_{37}) * 2^{38} + (c_{39},c_{38},c_{37}) * 2^7 + (d_7 d_6 \cdots d_0) * 2^{31} + (d_7 d_6 \cdots d_0) + (d_0) * 2^{159} \end{aligned} \tag{17}$$

where $(d_7,\cdots,d_1,d_0)$ are 8 bits of c_{36}. Now, all terms in Eq.16 and 17 have at most 159 bit length, the reduction result is simply the addition of these terms.

3.2.3 Further Optimization

Examining the computational complexity, we notice that point addition is more expensive than point doubling. As we have discussed, point multiplication can be decomposed to a series of point addition and doubling, we would rather use more point doubling than point addition to compute the point multiplication. Morain *et al.* found Non-adjacent forms (NAFs) is an effective way to achieve the lightest Hamming weight for scalar k in point multiplication $k * P$, which results to use the least number of point additions to calculate $k * P$ [8]. For example, $255 * P$, or $(11111111) * P$, requires 7 point additions. But if we transform it to $(10000000 - 1) * P$, which is $256 * P - P$, only one addition is required. Note the point subtraction can be replaced by point addition because the inverse of an Affine point $P = (x,y)$ is $-P = (x,-y)$. We implement NAFs technique in random point multiplication. According to our experiments, point multiplication with NAFs contributes at least 5% performance improvement.

Recall in the digital signature procedure in ECDSA, component r is generated by a point multiplication with the fixed base point of a selected elliptic curve. To further reduce the execution time, we precompute some partial results and apply sliding window method [5] to speed up fixed point multiplication. Different from NAFs, sliding window scheme groups scalar k into a number of $s-bit$ bit-clusters, where s is also called window size. So, k can be represented by $k_m * 2^{sm} + k_{m-1} * 2^{s(m-1)} + \cdots + k_0$, where k_i is a bit-cluster. If we precompute the point multiplication with every possible value of k_i, the number of point addition is bounded by $\lceil \frac{160}{s} \rceil - 1$. Note the sliding window method does not reduce the number of point doubling operations. Obviously, this scheme requires extra memory space for storing partial results. In practice, we select window size $s = 4$. Correspondingly, there are 16 entries in the partial result table. Our experiments show sliding window method is more effective than NAFs for fixed point multiplication, the performance of sliding window method is more than 10% better than that of NAFs.

Our initial experimental results indicated that it took double amount of time to perform an ECDSA verification than to do an ECDSA signature: signature is 1.35s, while verification is 2.85s. The reason

is that the verification requires two ECC point multiplications (while the signature only needs one point multiplication); the verifier has to perform $u_1P + u_2Q$ as shown in Section 2.2.2. To speed up the verification time, we adopt Shamir's trick [4] to do multiple point multiplication simultaneously. The idea of Shamir's trick is similar to the sliding window method discussed previously. Given t-bit u_1 and u_2, we use the window size ω and precompute the values $iP + jQ$ for $0 \leq i, j \leq 2^{\omega}$. At each of $\lceil t/\omega \rceil$ steps, we perform ω doubling and the (precomputed) additions determined by the window contents. The larger the window size (ω) is, the more memory is required for storing the precomputed values. In practice, we choose the single bit window size, $\omega = 1$. Therefore, only the value of $P + Q$ needs to be precomputed and stored. As the result, the performance of ECDSA verification has been improved more than 30%, from 2.85s to 1.96s. There is still further improvement space if multi-bit window size is used, but the trade-off is more memory overhead.

4 Experiments and Performance Evaluation

We have implemented the 160-bit ECC security primitive on MICAz mote, and revamped our previous ECC implementation [13] on TelosB and Tmote Sky motes. MICAz is powered by ATmega128 microcontroller. ATmega incorporates an 8MHz, 8-bit RISC CPU, 128K bytes flash memory (ROM) and 4KB RAM. The RF transceiver on MICAz is IEEE 802.15.4/ZigBee compliant, and can have 250kbps data rate. TelosB and Tmote Sky share the same platform with TI MSP430 16-bit processor, 48K bytes programming memory and 10KB RAM. The only difference between TelosB and Tmote Sky is the CPU frequency. Tmote Sky can work at 8MHz, while TelosB only works at 4Mhz.

4.1 ECC Evaluation

In this subsection, we first present the performance of our implementation on three sensor platforms. Then we use the MICAz mote as an example to give an overall analysis to quantify the computation complexity.

4.1.1 The performance of ECC Implementation

In experiments, we measure execution time and code size of our implementation. We choose secp160r1 as the elliptic curve in all experiments. We use the embedded system clock (921.6kHz for MICAz and 32.6kHz for TelosB/Tmote Sky) to measure the execution time of major operations in ECC, such as point multiplication, point addition and point doubling.

We first test point multiplication operation, which is comprised of point addition and doubling. We consider two cases in point multiplication. One is multiplying large integer with a fixed point(base point), and the other one is with a random point. Fixed point multiplication allows for optimization by precomputing. We apply sliding window technique[5] and set window size to 4, i.e., precomputing $2^4 - 1 = 15$ points. In experiments, we randomly generate 20 large integers to multiply with the point and take the average execution time as the result.

Since ECC point multiplication consists of addition and doubling operations, we further evaluate these two operations individually. We generate random points and large integers for tests. Since a single operation takes very little time, to reduce the error of clock inaccuracy, we measure 100 operations every round and take the average value.

We summarize the performance in Table 1, including ECC fix point multiplication (with size-4 sliding-window optimization) (FPM), random point multiplication (RPM), point addition (PAdd), point doubling (PDbl), ECDSA signature (SIGN) and verification (VERIFY). It clearly shows that the performance of ECC operation on MICAz is slightly better than that on TelosB, even though TelosB is equipped with an 8MHz,

	FPM	RPM	PAdd	PDbl	SIGN	VERIFY
MICAz	1.24s	1.35s	6.2ms	5.8ms	1.35s	1.96s
Tmote	0.74s	0.77s	3.7ms	3.5ms	0.77s	1.12s
TelosB	1.44s	1.55s	7.3ms	7.0ms	1.55s	2.25s

Table 1: The comparison of ECC execution Time on three mote platforms, including fixed point multiplication (FPM), random point multiplication (RPM), point addition (PAdd) and point doubling (PDbl) and ECDSA signature generation (SIGN), verification (VERIFY) time.

16-bit CPU. After a careful investigation, we found the performance degradation on TelosB is due to the following two reasons. First, the 8MHz CPU (MSP430) frequency on TelosB is just a nominal value. The maximum CPU clock rate is actually 4MHz. Second, the hardware multiplier in MSP430 CPU uses a group of special peripheral registers which are located outside of MSP430 CPU. As the result, it takes MSP430 eight CPU cycles to perform an unsigned multiplication, while it at most takes four cycles to do the same operation in ATmega CPU. The above two reasons explain why TelosB cannot perform better than MICAz.

Tmote Sky is capable of running at 8MHz CPU frequency instead of 4MHz on TelosB because it is equipped with an external resistor on the ROSC pin of MSP430 that enables the DCO to operate at a higher frequency. We simply enable the external resistor on Tmote and achieve the ECC performance twice faster than that on TelosB. As shown in Table. 1, it only takes 0.77s to finish a signature generation and 1.12s to verify it.

	ECC library		ECDSA		UART Comm.	
	ROM	RAM	ROM	RAM	ROM	RAM
MICAz	10,360	978	8,244	202	3,452	147
TelosB/Tmote	7,018	1,012	4,420	164	3,202	233

Table 2: ECC implementation code size.

Table 2 presents code sizes and data sizes of the ECC implementations. For TelosB and Tmote Sky platforms, the ECC library uses 7,018 byte ROM (for code) and 1,012 byte RAM (for data). Note more than 60% of data size is used to store the 15 elliptic points which are used in sliding-window optimization. When the data size budget is tight, the sliding-window optimization can be removed to have more data space. ECDSA module accounts for 4,420 bytes on TelosB and Tmote Sky. The reason is the included SHA1 module consumes around 3KB code size. Finally, for the IO purpose, we also have the UART communication module, which uses 3,202 bytes for code and 233 bytes for data. The total code size of our test program is 19,290 bytes.

Compared to TelosB and Tmote Sky, our ECC package is more space demanding on MICAz platform. The ECC library requires 10,360 bytes in code size for MICAz, 46% more than that on TelosB/Tmote. This is due to our assembly codes for optimizing the large number integer operations. Since the CPU register number in MICAz is twice the amount that in TelosB/Tmote, more instructions are needed to handle the extra register operations. For the same reason, the code size of ECDSA requires 8,244 bytes. Overall, the test program on MICAz uses 24,258 bytes for code and 1507 bytes for data.

4.1.2 A Performance Anatomy of ECC Point Multiplication on MICAz

Since ECC point multiplication dominates the computational complexity in ECC signature and verification, we are curious to learn the performance anatomy in ECC point multiplication.

This analysis is based on 160-bit ECC curves. We use secp160r1 as the example. We also assume 4-bit sliding window method is used, and partial results are precomputed. The computational cost for each window unit is 4 point doubling and 1 point addition. Given a 161 bit private key, there are 41 window units. Totally , 164 point doubling and 41 point additions are required to finish 1 point multiplication.

Large (160-bit) integer multiplication, squaring and reduction are the most expensive operations in point doubling and point addition. To learn the amount of time contributed by the above three operations in a fix point multiplication. We first individually test the performance of large integer multiplication, squaring and reduction. Our results show that it takes $0.47ms, 0.44ms$ and $0.07ms$ to perform a 160×160 multiplication, squaring and reduction, respectively. Next, we count the the number of each operation required in a point multiplication. Since we adopt the mixed coordination (the combination of Jacobian coordinate and Affine coordinate), each point addition requires 8 large integer multiplications and 3 large integer squaring, and each point doubling requires 4 large integer multiplications and 4 large integer squaring. In addition, each multiplication, squaring or shifting operation has to be followed by a modular reduction. Our program shows the point addition requires 12 modular reductions, and the point doubling requires 11 modular reductions. In total, each point multiplication costs $164 \times 4 + 41 \times 8 = 984$ large integer multiplications, $164 \times 4 + 41 \times 3 = 779$ large integer squaring and $164 \times 11 + 41 \times 12 = 2,296$ large integer modular reductions. Plugging in the results of the individual tests, we get the total amount of time consumed on the three operations is 0.97s, roughly 78.2% of the total time to do a fix point multiplication. The rest of 21.8% of the time is spent on various operations, including inversion operation (to convert the Jacobian coordinate to Affine), addition, subtraction, shifting and memory copy, etc. Based on above analysis, we believe the performance of ECC operations on MICAz can be further improved by more refined and careful programming.

4.2 Performance Comparison

In the last part of the evaluation, we compare the performance of our implementation with existing research results [3, 6, 7] and give the possible explanation of the performance gap.

	MICAz				TelosB	
	WM-ECC	Sun-ECC	TinyECC	EccM2.0	WM-ECC	TinyECC
SIGN	1.35s	0.81s	1.92s	30s	1.55s	4.36s
VERIFY	1.96s	-	2.43s	-	2.25s	5.44s

Table 3: The performance comparison of our ECC implementation, WM-ECC, with other research results, including Sun-ECC [3], TinyECC [6] and EccM2.0 [7]. We use MICAz and TelosB as the two platforms.

We first compare the computation time of ECC operations. We denote our ECC implementation as WM-ECC, and compare the ECDSA signature generation and verification time with other implementations in Table 3. Obviously, our WM-ECC is more computationally efficient than TinyECC and EccM2.0. On MICAz platform, TinyECC is 42% slower in signature generation than our implementation. On TelosB platform, the performance gap increases to 180%.

We also notice than Sun-ECC is more efficient than our WM-ECC. Their result, 0.81s for a random point multiplication, is about 40% faster than 1.35s of our result. We notice that the time for their 160×160

multiplication is 0.39ms, roughly 17% faster than our 0.47ms. In general, we believe their code is more polished and optimized in many aspects than our code. Furthermore, Our code is implemented in TinyOS, and mostly written with NesC (except several critical large integer operations), which could introduce more CPU cycles.

ECC+ECDSA	MICAz		TelosB	
	ROM	RAM	ROM	RAM
WM-ECC	18,604	1,180	11,438	1,176
TinyECC	13,858	1,440	12,564	1,526
EccM2.0	43k	820	-	-

Table 4: ECC implementation code size and data size comparison.

Since memory storage is extremely limited in sensor motes, the program code size and data size determine the feasibility of the ECC package. We compare our WM-ECC with TinyECC and EccM2.0. We do not compare Sun-ECC because it is not based on TinyOS so it is not comparable. To compare with the code size and data size of TinyECC that only has ECC and ECDSA modules, we combine ECC library and ECDSA of our WM-ECC, but not UART communication module. Note EccM2.0 only has the ECC module, there is no ECDSA available. Table 4 shows WM-ECC has the similar program code size and data size as TinyECC. The code and data sizes shown for Comparatively, EccM2.0 consumes much more code space. Given 128KB ROM, 4KB RAM on MICAz, and 48KB ROM, 10KB RAM on TelosB, WM-ECC can easily fit in existing applications. One may notice that WM-ECC requires extra 5KB code size than TinyECC on MICAz platform. This is due to the trade-off of the computation efficiency. We have extensively optimized the large integer operations on MICAz platform. As the result, the code size is slightly inflated due to the techniques such as loop unrolling. Considering the programming space MICAz is relatively large, 128KB, we believe this trade-off of 5KB code size is worthwhile.

5 Conclusion

In this technical report, we present how viable that WM-ECC can run on small, less-powerful sensor devices. We implement 160-bit ECC on popular sensor motes, including MICAz, Tmote Sky and TelosB. Our experiments show that WM-ECC is practical for all three sensor platforms. ECC signature of WM-ECC only takes 1.35s, 0.77s, 1.35s for MICAz, Tmote Sky and TelosB, respectively. Meanwhile, we believe there is still performance improvement space, which can be achieved by more careful programming.

Acknowledgment This project was partially supported by US National Science Foundation award CCF-0514985 and CNS-0721443.

References

[1] H. Cohen, A. Miyaji, and T. Ono. Efficient elliptic curve exponentiation. In *Advances in Crytology-Proceedings of ICICS, Lecture Notes in Computer Science*, pages 282–290, Springer-Verlag, 1997.

[2] H. Cohen, A. Miyaji, and T. Ono. Efficient elliptic curve exponentiation using mixed coordinates. In *ASIACRYPT: Advances in Cryptology*, 1998.

[3] Nils Gura, Arun Patel, Arvinderpal Wander, Hans Eberle, and Sheueling Chang Shantz. Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs. In *CHES*, Boston, Aug. 2004.

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

[5] C. K. Koc. High-Speed RSA Implementation. In *RSA Laboratories TR201*, Nov 1994.

[6] An Liu and Peng Ning. TinyECC: Elliptic Curve Cryptography for Sensor Networks. Sept 15 2005.

[7] D.J. Malan, M. Welsh, and M.D. Smith. A public-key infrastructure for key distribution in tinyos based on elliptic curve cryptography. In *The First IEEE International Conference on Sensor and Ad Hoc Communications and Networks*, Santa Clara, CA, October 2004.

[8] F. Morain and J. Olivos. Speeding up the computations on an elliptic curve using addition-subtraction chains. *Theoretical Informatics and Applications*, 24:531–543, 1990.

[9] S. Chang Shantz. From Euclid's GCD to Montgomery Multiplication to the Great Divide. In *Technical report, Sun Microsystems Laboratories TR-2001-95*, June 2001.

[10] Haodong Wang and Qun Li. Distributed user access control in sensor networks. In *IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS)*, pages 305–320, San Francisco, CA, June 2006.

[11] Haodong Wang and Qun Li. Efficient Implementation of Public Key Cryptosystems on MICAz and TelosB Motes. Technical Report WM-CS-2006-7, College of William and Mary, Computer Science, Williamsburg, VA, October 2006.

[12] Haodong Wang and Qun Li. Efficient implementation of public key cryptosystems on mote sensors (short paper). In *International Conference on Information and Communication Security (ICICS), LNCS 4307*, pages 519–528, Raleigh, NC, Dec. 2006.

[13] Haodong Wang, Bo Sheng, and Qun Li. Elliptic curve cryptography based access control in sensor networks. *International Journal of Sensor Networks*, 1(2), 2006.