TelosB Implementation of Elliptic Curve Cryptography over Primary Field
WM-CS Technical Report (WM-CS-2005-12)

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

10/18/2005

Abstract

Even though symmetric-key scheme, which has been investigated extensively for sensor networks, can fulfill many security requirements, public-key cryptography is more flexible and simple rendering a clean interface for the security component. Against the popular belief that public key scheme is not practical for sensor networks, this paper describes a public-key implementation in a sensor network. We detail the implementation of Elliptic Curve Cryptography (ECC) over primary field, a public-key cryptography scheme, on TelosB, which is the latest research platform in Berkeley Motes family. We evaluate the performance of our implementation and compare with other implementations we have ported to TelosB. We have achieved 3.3 seconds for public key signature and 6.7 seconds for verification in our implementation even though the hardware multiplier is disabled.

1 Introduction

Public-key cryptography has been used extensively in data encryption, digital signature, user authentication, etc. Compared with the popular symmetric key cryptography widely used in sensor network, public-key cryptography provides a more flexible and simple interface requiring no key predistribution, no pair-wise key sharing, no complicated one-way key chain scheme. 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 energy budget. The nascent exploration seems to disabuse of the misconception. The recent progress in 160-bit Elliptic Curve Cryptography (ECC) implementation on Atmel ATmega128, a CPU of 8Hz and 8 bits[8], shows that an ECC point multiplication takes less than one second, which proves public-key cryptography is feasible for sensor network security related applications.

This paper describes our implementation of ECC over primary field on TelosB, a sensor network research platform. To compare with similar implementations, we also port the other two implementations (NCSU[13] and Harvard[14]) to TelosB and measure the performance of each scheme respectively. Our experiments demonstrate that our implementation outperforms the other two implementations. This work is especially important since TelosB mote platform is becoming a popular testbed for sensor network research: a public-key cryptography implementation and performance evaluation and comparison are conducive to the research progress of the community.

Our implementations are conducted on TelosB mote (TPR2400), which is the latest product in the Mote family designed by University of California at Berkeley for experimentation in sensor network

research. It is of the size of two AA batteries integrating USB programming capability, an IEEE 802.15.4 radio with integrated antenna, a low-power MCU with extended memory, and an optional sensor suite (TPR2420). Its detailed features include: IEEE 802.15.4/ZigBee compliant RF transceiver, 2.4 to 2.4835 GHz (a globally compatible ISM band), 250 kbps data rate, integrated onboard antenna, 16 bit, 8MHz TI MSP430 microcontroller with 10kB RAM, low current consumption, 1MB external flash for data logging, programming and data collection via USB, optional sensor suite including integrated light, temperature and humidity sensor (TPR2420), supported by Berkeley's TinyOS operating system[1] and the NesC programming language[7].

Our experiments show that it takes 3.3s and 6.7s to conduct a public key signature and verification respectively even though the hardware multiplier is disabled. It is possible to further reduce the running time by using the hardware multiplier, profiling the running time of each component, and analyzing the critical path of ECC computation. Our experiment results demonstrate that public-key cryptography is feasible for sensor network security applications.

The rest of the paper is organized as follows. Section 3 gives an introduction to ECC. Section 4 shows implementation details of ECC on TelosB mote. Section 5 evaluates the performance of our implementation and compares with other implementations. Section 6 concludes the paper.

2 Related Work

NIST and SECG have specified example elliptic curves domain parameters at required security levels [19, 18]. As other implementations, we follow the recommended parameters in our implementation.

In [9], Gura *et al.* implemented ECC and RSA on 8-bit microcontroller and compared their performance. In their ECC implementation, the elliptic curves are defined over standardized prime integer field $GF(p)$. Some optimization techniques are applied for point multiplication, such as projective coordinates, non-adjacent forms, and curve-specific optimization. Additionally, the paper focuses on the optimization of modular multiplication of large integers, which is a critical operation for ECC. The authors compared row-wise and column-wise multiplications, and further proposed a new hybrid strategy to improve the performance. The experiments showed that 160-bit ECC execution time was reduced to less than one second and much faster than RSA-1024 operations.

Shantz[20] presented an efficient technique to calculate modular division, which is an important arithmetic operation in ECC and other cryptography system. The idea is to compute $\frac{y}{x}$ in one operation, instead of the previous method which first compute $\frac{1}{x}$ and then multiply it with y. Thus, this scheme reduces one multiplication in the modular division operation. The new algorithm can be applied in both $GF(p)$ and $GF(2^m)$ fields.

Woodbury *et al.* [6] introduced another ECC system over Optimal Extension Fields(OEFs) [2] $GF(p^m)$, where p is chosen as the form of $2^n \pm c$. The authors present the implementation of a specific 134-bit ECC in details. The experiments indicate that point multiplication can be performed within 2 seconds.

Cohen *et al.* analyzed the impact of coordinates system in ECC implementation in [4]. The authors measured the performance of point addition and doubling of different coordinates systems, and proposed a new modified Jacobian coordinates which achieves the fastest doubling operation. Moreover, they introduced a mixed coordinates system, which divides exponentiation into sub-operations and chose the best coordinates representation for each sub-operation. Thus, this scheme combines the advantages of different coordinates system and improves the computation time significantly.

In [10], an implementation of 160-bit ECC cryptographic library for ECDSA over prime field $GF(p)$ on a CISC microcontroller(MC16) was given. Experiments obtain a speed of 150ms for signature generation and 630ms for verification.

EccM[15] is an ECC system implemented over binary field $GF(2^p)$. The average time for public key generation is claimed as 34 seconds. Sizzle[8] is another application of ECC system developed recently. The standard Internet security protocol(SSL) is efficiently implemented in sensor motes by using ECC.

3 ECC Introduction

In this section, we briefly give a background introduction about elliptic curve cryptography, and corresponding elliptic curve Diffie-Hellman and Digital Signature Algorithm.

3.1 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 p) 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 $GF(p)$. Since the field $GF(p)$ (p is a prime) is generally used in cryptographic applications, (1) can be simplified to:

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

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

The elliptic curve group operation is closed 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 computationally intractable to find a number k such that $Q = kP$.

3.2 Elliptic Curve Diffie-Hellman (ECDH) and Digital Signature Algorithm (ECDSA)

The original Diffie-Hellman secret sharing protocol [5] requires a key of at least 1024 bits to achieve sufficient security. Unfortunately, low-power architecture, such as MSP430 and ATMega128, cannot afford the large memory overhead. Diffie-Hellman scheme based on ECC, however, can achieve the same security level with only 160 bit key size. A typical ECDH scheme is shown in Fig. 1. Initially, Alice and Bob agree on system base point P, and generate their own public keys Q_A and Q_B. To share a secret, Alice and Bob exchange their public keys, and then use their own private key to multiply the other's

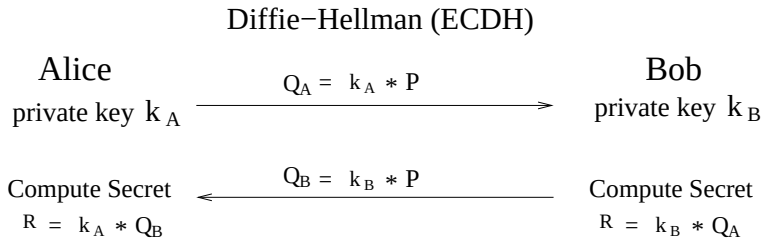

Figure 1: An example of ECC version of Diffie-Hellman Protocol.

public key. The result point R will be the secret. Eve, an eavesdropper, may overhear the communication and learn the public keys from Alice and Bob. However, with the knowledge of P, Q_A, and Q_B, it is computationally intractable for Eve to get Alice and Bob's private keys. As the result, she can not figure out secret R.

ECC can also be used for Digital Signature Algorithm. Similarly, Alice and Bob have to agree on a particular curve with base point P. We assume the field is $GF(p)$, 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 $Q = kP$).

1. Choose a random key k in $[1, q-1]$;
2. Compute kP, results a point with coordinate (x_1, y_1). Check $x \pmod q$, go back to the first step if the result is zero;
3. Compute $k^{-1} \pmod q$;
4. Compute $s = Hash(m) + xr$, where $Hash$ is an 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 $u_1 P + u_2 Q$, get the result point (x_2, y_2);
4. The signature is verified if $x_2 = r$.

4 ECC Implementation

We implement ECC cryptosystem on Telos-B mote powered by MSP430 microcontroller. The MSP430 incorporates an 8MHz, 16-bit RISC CPU, 48K bytes flash memory (ROM) and 10K bytes RAM. This architecture provides 27 instructions and 7 addressing modes. The CPU also provides sixteen 16-bit registers. The first four are dedicated for special-purpose, such as program counter, stack pointer and status register. The rest of twelve are available for general use. Besides, the MSP430 also provides a peripheral hardware multiplier, which is capable of conducting 16×16 bits multiplication.

Given the limited processor resources, we concentrate most of our efforts on computation optimization. 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 [9], we limit our discussion in prime number finite field $GF(p)$ in this paper. Without further clarification, our following discussion is based on SECG recommended 160-bit elliptic curve: secp160r1.

4.1 Large Integer Operations

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

4.1.1 Multiplication

The efficiency of large integer multiplication dominates the overall performance of ECC operation. Gura *et al.* show that as much as 85% of execution time is spent on multiplication for a typical point multiplication in ECC. That means the optimization on multiplication is critical for overall performance of our implementation. We have compared three different multiplication implementations [9, 16, 13], and finally decided to use Hybrid Multiplication proposed in [9]. 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 part of final results. On average, one memory load is required for each $k \times k$ multiplication. When integer size n is increased (integer size is 10 for curve secp160r1), the required number registers increase linearly in Row-wise method.

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. Considering a large number of data in ECC operations, unnecessary memory operations would lower the performance.

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, then it equals to Row-wise method. Therefore, a single memory load operation can be used for several multiplications. A larger d leads to fewer memory operations, but requires more registers. Since the MSP430 microcontroller only has 12 general registers, we can only implement the Hybrid method with column size $d = 2$, which requires 5 accumulator registers, 3 operand register and other 4 registers for pointer, temporary storage and loop control.

To achieve better performance and enable flexible control over registers, we implement the Hybrid multiplication in assembly language. Our experiments show that the performance of point multiplication improves about 5% with the Hybrid multiplication compared with the Column-wise method, and improves another 5% with assembly language compared with original implementation with C.

4.1.2 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 [20]. 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

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

To find r efficiently, we maintain 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 registers and assigned with initial values x, q, y, and 0, respectively. Note the second invariant relationship is true even for $v = 0$ because algebraically the value of modulus is equivalent to zero in finite field. The division procedure repeatedly reduces the values of A and B by following way. In each iteration, if either A or B is even, we divide by 2 both sides of the equation. If U or V is not even at that time, we can make it even by adding modulus p. If both A and B are odd, we add two equations together and then divide 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

$$y \equiv U * x. \tag{12}$$

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. Otherwise, the routine would return the greatest common divisor of A and B. The Great Divide finishes division or inversion operation in $2(log(x) - 1)$ steps.

4.1.3 Reduction

The modular reduction operation is as important as modular multiplication. Each multiplication must be followed by a reduction operation. Note the Great Divide algorithm does not work for modular reduction. 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 section of ECC operation. Now, we discuss the modular reductions in digital signature generation and verification. In most cases, the order of an elliptic curve is not a pseudo-Mersenne prime, the optimization cannot be applied for those reduction calculation. We choose the classic long division method to implement this operation. It may not be the most efficient algorithm, but it does not affect the overall performance much because only very limited number of modular reductions are required in digital signature algorithm. We briefly describe the long division method as follows.

Given an integer x, we want to calculate

$$r \equiv x \bmod \mathrm{p}, \tag{13}$$

where p is a prime.

1. Align the most significant byte (MSB) of modulus p to the MSB of x, the lower bytes of p can be filled with zeros;
2. Starting with the MSB of x, divide the first two MSBs of x by the MSB of modulus p, and get the quotient;
3. Multiply the quotient with the modulus and get a subproduct;
4. If the subproduct is greater than the remainder of x (over estimation), subtract the modulus from the subproduct;
5. Then subtract the subproduct from the remainder of x;
6. The procedure continues and goes back to step 2 if the MSB of the remainder becomes zero;
7. If the MSB of the remainder is not zero (under estimation), subtract the modulus from the remainder, and then go back to step 2;
8. The procedure stops when the remainder is less than modulus p.

The long division producer reduces the remainder of x by one byte in each iteration.

4.2 ECC Operations

In this section, we present our 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.

4.2.1 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 [3]. They further found addition and doubling in mixed coordinate, with the combination of Modified Jacobian coordinate and Affine coordinate, lead to the best performance [4]. 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.

$$\begin{aligned} X_3 &= -H^3 - 2X_1H^2 + r^2, \\ Y_3 &= -Y_1H^3 + r(X_1H^2 - X_3), \\ Z_3 &= Z_1H, \\ aZ_3^4 &= aZ_3^4, \end{aligned} \tag{14}$$

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.

$$\begin{aligned} X_3 &= T, \\ Y_3 &= M(S - T) - U, \\ Z_3 &= 2Y_1Z_1, \\ aZ_3 &= 2U(aZ_1^4) \end{aligned} \tag{15}$$

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.14 and Eq.15, point addition requires 9 large integer multiplications and 5 squarings, and point doubling requires 4 multiplications and 5 squarings.

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

$$M = 3X_1^3 - 3Z_1^4 = 3(X_1 + Z_1^2)(X_1 - Z_1^2). \tag{16}$$

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

4.2.2 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 present two implementations to do the reduction. Although the theory behind these two methods are exactly the same, the second methods performs 10 times better than the first one. Again, our discussion is based on curve secp160r1.

Suppose we use the 16-bit architecture, the multiplication result can be represented by

$$C(c_{19}, \cdots, c_{10}, c_9, \cdots, c_1, c_0),$$

where c_i $(0 \leq i \leq 19)$ is a word with 16 bits, and c_{19} is the most significant word. The 20-word integer can also be written as:

$$C = (c_{19}, \cdots, c_{10}) * 2^{160} + (c_{10}, c_9, \cdots, c_1, c_0) \tag{17}$$

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

$$\begin{aligned} C &\equiv (c_{19}, \cdots, c_{10}) * (2^{31} + 1) + (c_{10}, c_9, \cdots, c_1, c_0) \\ &\equiv (c_{19}, \cdots, c_{10}) * 2^{31} + (c_{19}, \cdots, c_{10}) + (c_{10}, c_9, \cdots, c_1, c_0) \end{aligned} \tag{18}$$

Since each word has 16 bits, the first term in the result of Eq. 18 can be further reduced to

$$\begin{aligned} (c_{19}, \cdots, c_{10}) * 2^{31} &\equiv c_{19} * 2^{175} + c_{18} * 2^{159} + (c_{17}, \cdots, c_{10}) * 2^{31} \\ &\equiv c_{19} * 2^{15} * (2^{31+1} + (d_{15}, \cdots, d_1, d_0) * 2^{159} \\ &\quad + (c_{17}, \cdots, c_{10}) * 2^{31} \\ &\equiv c_{19} * 2^{15} + c_{19} * 2^{46} + (d_0) * 2^{159} \\ &\quad + (d_{15}, \cdots, d_1) * (2^{31} + 1) \\ &\quad + (c_{17}, \cdots, c_{10}) * 2^{31} \\ &\equiv c_{19} * 2^{15} + c_{19} * 2^{46} + (d_0) * 2^{159} + (d_{15}, \cdots, d_1) * 2^{31} \\ &\quad + (d_{15}, \cdots, d_1) + (c_{17}, \cdots, c_{10}) * 2^{31}, \end{aligned} \tag{19}$$

where $(d_{15}, \cdots, d_1, d_0)$ are 16 bits of c_{18}. Now, all terms in Eq.18 and 19 have at most 159 bit length, the reduction result is simply the addition of these terms.

In our experiments, we found the above modular reduction method is not efficient as we thought. In fact, it costs 2.5 times of execution time of a large integer multiplication! The poor performance of the above implementation is caused by too many memory copying and shifting. To avoid expensive memory operations, we use a *while* loop to reduce $(c_{19}, \cdots, c_{10})$:

$$\begin{aligned} &while\,((c_{19}, \cdots, c_{10}) \neq 0) \\ &\quad (c_{19}, \cdots, c_{10}, c_9, \cdots, c_1, c_0) = (c_{19}, \cdots, c_{10}) * \omega + \\ &\qquad\qquad (c_9, \cdots, c_1, c_0) \\ &C = (c_9, \cdots, c_1, c_0) \bmod p, \end{aligned} \tag{20}$$

where $\omega = 2^{31} + 1$. It is easily noticed that there are at least two iterations in the loop, so it will cost at least two large integer multiplications. However, since multiplier ω only has two words (or 32 bits), the multiplication is much faster than the multiplication of two 160-bit integers. In reality, we implement a separate multiplication routine dedicated for this special operation. As the result, the performance second modular reduction method improves 10 times.

4.2.3 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 $k * P$ [17]. 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 [11] 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.

5 Experiments and Performance Analysis

In this section, we first compare the performance of our implementation with other two implementations. Then, we give an overall analysis to quantify the computation complexity.

In experiments, we measure execution time, power consumption and code size of our implementation and compare the performance with other two released codes, TinyEcc[13] and EccM[14]. We make appropriate modifications on their program to make them executable on TelosB platform. And we choose secp160r1 as the elliptic curves parameters in all experiments. Since EccM[14] is implemented for ECC curve on binary field, we choose curve secp163r2 for EccM for comparison.

5.1 Execution Time

We first test the performance of point multiplication operation. We consider two cases in point multiplication: one is multiplying a large integer with a fixed point(base point) and the other is with a random point. Fixed point multiplications allow for optimizations by precomputing. We apply sliding window technique[12] and set windows size to 4, i.e., precomputing $2^4 - 1 = 15$ points. In experiments, we randomly generate large integers to multiply with points and take the average execution time as the result.

Since ECC point multiplication consists of point 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 impact of clock inaccuracy, we measure 100 operations every round and take the average value of multiple rounds.

Table 1 shows the experimental results of execution time. Point addition and doubling of our implementation is superior to the other codes, which results in a faster point multiplication. Since EccM has no precomputation, there is no difference between the multiplications with base point and random point. Comparing with TinyEcc, we achieve more than 60% improvement in fixed point multiplication. And for random point multiplication, which does not involve precomputing optimization, our implementation is more than 64% faster.

	Fixed Point Mult	Random Point Mult	Point Add	Point Doubling
Our Codes	3.13s	3.51s	0.133s	0.137s
TinyEcc	7.98s	9.86s	0.315s	0.300s
EccM	88.43s	88.43s	0.262s	0.255s

Table 1: Execution Time of Point Operations

Moreover, we implement ECDSA signature scheme and compare the performance with TinyEcc. The following Table 2 is the experimental results. In fact, when signing a message, one fixed point multiplication is the dominant operation. As we can see, the performance improvement of signature time is very close to that of fixed point multiplication. On the other hand, verification of ECDSA mainly consists of one fixed point multiplication and one random point multiplication. According to Table 1, we expect to achieve at least 11 seconds improvement. However, TinyEcc implemented ECDSA in a different way with the assumption that the verifier is aware of the sender's public key. Thus, precomputation is conducted to optimize the performance for the random point multiplication. We think this assumption is impractical for a scalable sensor network and we implement pure random point multiplication in the verification process. That explains why we only gain about 9.5 seconds.

	Signature	Verification
Our Codes	3.35s	6.78s
TinyEcc	8.24s	16.26s

Table 2: Execution Time of ECDSA

5.2 Power Consumption

The power consumption is estimated by the formula $E = U \cdot I \cdot t$. Thus, it is linear to the execution time. TelosB uses an ultra-low-power microcontroller MSP430. According to the specifications of TelosB, the current is 1.8mA in active model. And the typical voltage is 3V for two AA batteries. Therefore, for our ECDSA implementation, generating a signature consumes roughly 18.09mJ energy and verification costs 36.61mJ.

5.3 Code Size

The following Table 3 compares the code size of three implementations. Our program uses 42.3k ROM and 1.6k RAM for ECDSA protocol, where SHA-1, occupying approximately 30K memory space, takes a large portion of codes. Our implementation code size is slightly larger than TinyEcc because we apply the hybrid multiplication for large integer multiplication and squaring modules, which is equivalent to loop unrolling. Meanwhile, some dedicated utility functions, which are aimed to speed up a specific operation (e.g., modular reduction), also contribute to the slight inflation of the code size. However, our ECDSA RAM size is 25% smaller than TinyEcc, which means our data usage is more efficient.

	ECC library		ECDSA	
	ROM	RAM	ROM	RAM
Our Codes	13.8k	1.3k	42.3k	1.6k
TinyEcc	12.5k	1.3k	39.2k	2.1k
EccM	17.6k	1.1k	–	–

Table 3: Code Size

5.4 Performance Analysis

Since ECC point multiplication dominates over 95% of execution time in either ECDH or ECDSA, our performance analysis focuses on this operation only. The performance analysis is based on 160-bit ECC Curve, such as secp160r1. We also assume 4-bit sliding window method is used, and partial results are precomputed. The analysis can be also applied to different ECC curve and different sliding window size.

The computational cost in each 4-bit window unit is 4 point doubling and 1 point addition. Given a 161 bit private key, there are 41 window units. Correspondingly, the point multiplication costs 164 point doublings and 41 point additions. To simplify the presentation, we denote "MUL" as large integer multiplication, and let "SQR" be large integer squarings, and let "MOD" be large integer modular reduction.

For point addition in mixed coordinates, each operation costs 8 MULs and 3 SQRs. And each point doubling (optimized for secp160r1) in Jacobian coordinate costs 4 MULs and 4 SQRs. In total, point multiplication costs 984 multiplications and 780 squarings. In addition, each multiplication or squaring operation has to be followed by a MOD. Combined with some other MODs after shifting operations, there are 12 MODs in each point addition and 11 MODs in each point doubling. After being optimized, a MOD only costs 20%-25% (by experiments) of the execution time of a MUL. So we will add another 3 MULs for each point doubling and addition. Therefore, MODs introduce 615 more MULs. Also, considering the cost of SQR is 90%-95% of MUL (by experiments), the total cost for one ECC point multiplication is 2,248 MULs. Our experiments show that 1000 MULs costs 0.95 second. Therefore, the three major operations: MUL, SQR and MOD, cost about 2 seconds in the execution of ECC point multiplication, which is consistent with our final experimental result: 3.1s (if consider the rest 1 second is spent in all other overheads combined).

The above performance is achieved without the hardware multiplier. Due to some unknown bugs in TelosB supporting software, the hardware multiplier has to be disabled in TinyOS to avoid system crash. We profiled the running time of 16 x 16 multiplication, and found it more than 10 times slower than the speed of hardware multiplier. It can be expected that our implementation will have a significant performance boost after the bugs in mspgcc are fixed, and be comparable or better than the performance achieved in [9].

6 Conclusion

In this paper, we show our implementation of ECC over primary field on TelosB platform and compare the performance with other implementations that are ported to TelosB. Our experiments show that it takes 3.3s and 6.7s to conduct a public key signature and verification respectively even though the hardware multiplier is disabled. It is possible to further reduce the running time by using the hardware multiplier, profiling the running time of each component, and analyzing the critical path of ECC computation. Our experiment results demonstrate that public-key cryptography is feasible for sensor network security applications.

Acknowledgment

We thank Nils Gura for answering some of our questions.

References

[1] Tinyos. *http : //www.tinyos.net/*.

[2] D. V. Bailey and C. Paar. Optimal extension fields for fast arithmetic in public-key algorithms. In *Advances in Cryptography — CRYPTO'98*, pages 472–485, 1998.

[3] H. Cohen, A. Miyaji, and T. Ono. Efficient elliptic curve exponentiation. In *Advances in Crytology-Proceedings of ICICS'97, Lecture Notes in Computer Science*, pages 282–290, Springer-Verlag, 1997.

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

[5] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transaction of Information and Theory*, IT-22:644–654, November 1976.

[6] A. D.Woodbury, D. V. Bailey, and C. Paar. Elliptic curve cryptography on smart cards without coprocessors. In *The Fourth Smart Card Research and Advanced Applications (CARDIS2000) Conference*, Bristol, UK, Sept. 2000.

[7] David Gay, Phil Levis, Rob von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC language: A holistic approach to networked embedded systems. In *Programming Language Design and Implementation (PLDI)*, June 2003.

[8] Vipul Gupta, Matthew Millard, Stephen Fung, Yu Zhu, Nils Gura, Hans Eberle, and Sheueling Chang Shantz. Sizzle: A standards-based end-to-end security architecture for the embedded internet. In *Third IEEE International Conference on Pervasive Computing and Communication (PerCom 2005)*, Kauai, Mar. 2005.

[9] 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.

[10] T. Hasegawa, J. Nakajima, and M. Matsui. A practical implementation of elliptic curve cryptosystems over gf (p) on a 16-bit microcomputer. In *Public Key Cryptography PKC '98*, pages 182–194, 1998.

[11] Cetin Kaya Kac. High-speed rsa implementation, rsa laboratories technical report tr-201, version 2.0. Nov 22 1994.

[12] C. K. Koc. High-speed rsa implementation. In *RSA Laboratories TR201*, Nov. 1994.

[13] An Liu and Peng Ning. Tinyecc: Elliptic curve cryptography for sensor networks. Sept 15 2005.

[14] David J. Malan, Matt Welsh, and Michael D. Smith. Eccm: A public-key infrastructure for key distribution in tinyos based on elliptic curve cryptography. 2004.

[15] David J. Malan, Matt Welsh, and Michael D. Smith. A public-key infrastructure for key distribution in tinyos based on elliptic curve cryptography. *First IEEE International Conference on Sensor and Ad Hoc Communications and Networks*, October 2004.

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

[17] 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.

[18] National Institute of Standards and Technology. Recommended elliptic curves for federal government use. Aug. 1999.

[19] Certicom Research. SEC 2: Recommended elliptic curve domain parameters. In *Standards for Efficient Cryptography Version 1.0*, Sept. 2000.

[20] 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.