Studying Software Implementations of Elliptic Curve Cryptography

Hai Yan and Zhijie Jerry Shi
Department of Computer Science and Engineering, University of Connecticut

Abstract

Elliptic Curve Cryptography (ECC) provides a similar level of security to conventional integer-based public-key algorithms, but with much shorter keys. ECC over binary field is of special interest because the operations in binary field are thought more space and time efficient. However, the software implementations of ECC over binary field are still slow, especially on low-end processors that are used in small computing devices such as sensor nodes. In this paper, we study the software implementations of ECC on processors with different word sizes. With a set of algorithms that we identified, we can perform 163-bit ECC in 13.9 seconds on an 8-bit processor at a clock rate of 8 MHz.

1. Introduction

Elliptic Curve Cryptography (ECC), proposed independently in 1985 by Neal Koblitz [1] and Victor Miller [2], has been used in cryptographic algorithms for a variety of security purposes such as key exchange and digital signature. Compared to traditional integer-based public-key algorithms, ECC algorithms can achieve the same level of security with much shorter keys. For example, 160-bit Elliptic-curve Digital Signature Algorithm (ECDSA) has a security level equivalent to 1024-bit Digital Signature Algorithm (DSA) [3]. Because of the shorter key length, ECC algorithms run faster, require less space, and consume less energy. These advantages make ECC a better choice of public-key cryptography, especially in resource constrained systems such as sensor nodes and mobile devices for pervasive computing.

Considerable work on ECC has been focused on mathematical methods and algorithms, hardware implementations, and extensions of instruction set architecture [4][5][6][7][8][9]. Hankerson et al. discussed the software implementations of in ECC in [6], in which they focused on 32-bit processors. Gura et al. compared the performance of ECC and RSA on 8-bit processors [8]. But the elliptic curves they studied are in GF(p). Malan recently studied the feasibility of implementing ECC in sensor nodes [5]. Compared to their results, our implementation is faster and requires less memory.

The performance of ECC on low-end processors is far from being satisfactory. Consequently, many protocols designed for wireless sensor networks tend to use symmetric-key algorithms only [10][11][[12]. In this paper, we try to identify problems in the software implementations of ECC. We select a set of efficient ECC algorithms and study their performance on processors of different word sizes. On 8-bit processors at a clock rate of 8 MHz, our implementation achieves a performance of 13.9 seconds per multiplication for 163-bit ECC.

The rest of the paper is organized as follows. In Section 2, we briefly describe ECC and relevant algorithms. In Section 3, we compare ECC algorithms on 32-bit processors. In Section 4, we discuss the implementation of ECC on an 8-bit processor. Section 5 concludes the paper.

2. Overview of ECC

Commonly-used elliptic curves are defined in either a prime field GF(p) or a finite field of characteristic two GF(2^m), which is also called a binary field [13]. The elliptic curves over binary field are of special interest to cryptography because the operations in a binary field are faster and easier to implement than those in prime fields. We will focus on ECC over binary field in this paper.

2.1 ECC operations and parameters

When defined in a binary field, an elliptic curve can be represented by

$$y^2 + x \bullet y = x^3 + a \bullet x^2 + b, \quad (1)$$

where a and b are constants in GF(2^m) and $b \neq 0$. The set E(GF(2^m)) includes all the points on the curve and a special point O, which is defined as the identity element. For any point $P = (x, y)$ in E, we have:

$$P + O = O + P = P,$$
$$P + (-P) = O, \text{ where } -P = (x, x + y).$$

The addition of two points on the curve, $P = (x_1, y_1)$ and $Q = (x_2, y_2)$, are defined as $P + Q = (x_3, y_3)$, where

$$x_3 = \lambda^2 + \lambda + x_1 + x_2 + a,$$

$$y_3 = \lambda (x_1 + x_3) + x_3 + y_1,$$

and

$$\lambda = \begin{cases} \dfrac{y_2 + y_1}{x_2 + x_1}, \text{ if } P \neq Q \\ \dfrac{y_1}{x_1} + x_1, \text{ if } P = Q. \end{cases} \quad (2)$$

The multiplication of a point P and an integer k is defined as adding k copies of P together.

$$Q = kP = \underbrace{P + \cdots + P}_{k \text{ times}} \cdot \quad (3)$$

If k has l bits and k_j represents bit j of k, the scalar point multiplication kP can be computed as:

$$kP = \sum_{j=0}^{l-1} k_j 2^j P \quad (4)$$

The multiplication can be done with the basic binary algorithm. For example, in the left-to-right binary algorithm, the bits in k are scanned from the left to the right. For each bit, the partial product is doubled, and then P is added to it if the bit is 1. The expected running time of this method is approximately $l/2$ point additions and l point doublings. Typically, $l = m$, which is also the key length in ECC.

The scalar point multiplication is the main operation in ECC. The binary algorithm for scalar point multiplications is similar to that for exponentiations in integer-based public-key algorithms. Therefore, the techniques, such as precomputations and sliding windows, that accelerate integer exponentiations can also be applied in similar ways to scalar point multiplications.

NIST has recommended five elliptic curves over binary field for the Elliptic Curve Digital Signature Algorithm (ECDSA) [3]. The five curves are defined in binary fields GF(2^{163}), GF(2^{233}), GF(2^{283}), GF(2^{409}), and GF(2^{571}). In this paper, we will focus on the five elliptic curves specified in ECDSA.

2.2 Binary field arithmetic

On programmable processors, an element in GF(2^m) is often represented with a polynomial whose coefficients belong to {0, 1}. The coefficients can be packed into words, with each bit representing a coefficient. For example, an element in GF(2^{31}), $x^{29} + x + 1$, can be represented with a word 0x20000003 on a 32-bit processor. If a polynomial has a degree of m and w is the word size, the polynomial can be represented with $\lceil (m + 1)/w \rceil$ words.

Addition. The addition of two polynomials a and b in GF(2^m) is just bitwise *xor* of the words representing a and b.

Multiplication. The multiplication of two polynomials of degree $(m - 1)$ results in a polynomial of degree $2m - 2$. The product needs to be reduced with respect to an irreducible polynomial $f(x)$ of degree m. An irreducible polynomial with a few terms can be chosen to facilitate fast reductions. In ECDSA, for example, the irreducible polynomial $f(x)$ in GF(2^{163}) has only five terms: $f(x) = x^{163} + x^7 + x^6 + x^3 + 1$. The modular reduction can be done during or after the polynomial multiplication.

The most straight-forward algorithm for polynomial multiplications is the *shift-and-add* method, similar to the method for normal binary multiplications. The difference is that the add operations are in GF(2^m). When adding two polynomials a and b, we can first set the partial product c to 0 if $a_0 = 0$ or to b if $a_0 = 1$. Then we scan the bits in a from a_1 to a_{m-1}. For each bit, b is first shifted to the left by one and if the bit in a is 1, we also add the new value of b to the partial product c. The modular reduction can be integrated into the shift-and-add multiplication. After each shift operation, the degree of b is checked. If b has a degree of m, it can be reduced to $b + f$, where f is the irreducible polynomial. This method is suited for hardware implementations where the shift operation can be performed in one clock cycle. However, it is less desirable for software implementations because shifting a polynomial stored in multiple words is a slow operation that incurs many memory accesses.

The comb algorithms are normally used to perform fast polynomial multiplications [6]. Suppose a and b are two polynomials stored in t words and each word consists of w bits. Unlike in the shift-and-add algorithm, the bits in a are not tested one by one sequentially. The comb algorithm first tests bit 0 of all the words in a, from $a[0]$ to $a[t - 1]$, i.e., bits a_0, a_w, a_{2w}, and so on. Then it tests bit 1 in all the words, then bit 2 in all the words, and so on. Compared to the shift-and-add method, the comb algorithm reduced the number of shift operations significantly, from $m - 1$ to $w - 1$.

The left-to-right comb algorithm is similar to the right-to-left comb algorithm, but it tests the bits in a from the left to the right, i.e., from the most significant bit to the least significant bit. The shift operations are performed on partial product c, not on b. Because c has twice as many words as b, the left-to-right comb algorithm is a little bit slower.

The left-to-right comb algorithm does have some advantages though. In addition to keeping the input polynomials unchanged, it can employ the sliding window technique to reduce the number of shift operations [6]. By scanning the bits in a with a window of a fixed size, the algorithm can multiply more than one bit with b at a time. The partial product c is then shifted left by the window size. The products of b and every possible value in the window are precomputed and stored in a table. The sliding window method reduces the number of shifts at the cost of storage. A larger window size leads to fewer shift operations but requires more space to save the precomputed results.

The squaring of a polynomial can be performed much faster than normal multiplications, taking advantage of the fact that the representation of a^2 can be obtained by inserting 0's between consecutive bits in a's binary representation [14].

Modular reduction. A modular reduction is needed in the multiplication and squaring algorithms, to reduce the degree of the product below m. The modular reduction is done with polynomial long division. Let c be a polynomial of degree i where $i \le 2m - 2$, and f be the irreducible polynomial of degree m. We can reduce the degree of c by eliminating the highest term x^i.

$$c = c + f \bullet x^{i-m}. \quad (5)$$

We can repeat this process until the degree of c is smaller than m.

If the irreducible polynomial f is stored in memory like regular polynomials, i.e., its coefficients are packed into words, Formula (5) can be done by first shifting f to the left by $(i - m)$ bits, and then adding the result to c. A polynomial shift is needed for every term that has a degree of m or larger. If memory space is not a concern, the number of shift operations can be reduced by precomputing $f \bullet x^j$ for $j = 0, 1, \ldots, w-1$, where w is the number of bits in a word [6].

If an irreducible polynomial has only a few terms, it can be represented more efficiently with an array that stores the position of the terms. To perform the operations in (5), we can calculate the position of the terms after the shift left operation and flip the corresponding bits in c.

A faster modular reduction algorithm was described in [6]. It works for irreducible polynomials in which the highest degree of the terms is much larger than the second highest degree, and the difference between two degrees should be larger than the word size. An example of such polynomials is $f(x) = x^{163} + x^7 + x^6 + x^3 + 1$, in which the highest degree 163 is much larger than the second highest degree 7. With these types of irreducible polynomials, multiple terms in c can be eliminated at a time. In typical software implementation, the number of terms that can be eliminated at the same time is the word size.

Inversion The calculation of λ in Formula (2) needs a division in GF(2^m), which is normally done with an inversion followed by a multiplication. To divide a by b, we first obtain b^{-1}, the inverse of b, and then compute $a \bullet b^{-1}$. The classic algorithm for computing the multiplicative inverse is Extended Euclidean Algorithm (EEA) [6].

The Almost Inverse Algorithm (AIA) in [14] is based on EEA. But unlike EEA, AIA eliminates the 1 bits in polynomials from the right to the left. AIA is expected to take fewer iterations than EEA. However, AIA does not give the inverse directly and needs an additional reduction to generate the inverse. Modified Almost Inverse Algorithm (MAIA), a modification of AIA, gives the inverse directly [6]. We do not list the details of these algorithms due to the space limit. The reader is referred to references [6] and [14].

3. ECC on General-Purpose Processors

In this Section, we compare the performance of ECC algorithms on general-purpose processors. The algorithms are implemented with C on a Debian Linux system. All the code is compiled with gcc 3.3.5 with `-O3` option. Table 1 summarizes the performance of the field arithmetic operations on a Pentium 4 processor at 3 GHz. We used the five elliptic curves that NIST recommended for ECDSA [3].

Table 1. Execution time of field operations*

	***m* =**	**163**	**233**	**283**	**409**	**571**
Add	2 polys	31.6	40.7	43.6	65.7	81.6
	3 polys	37.5	48.0	49.2	71.0	86.7
Mod	By bit	8467.0	8384.4	11016.1	12802.1	21668.9
	By word	285.8	226.3	399.5	343.0	749.8
	Fixed poly	40.9	46.7	60.5	69.9	107.7
Mul	R-to-L comb	5870.5	8251.4	9891.7	16413.6	29180.9
	L-to-R comb 4-bit window	3118.4	4172.4	5235.5	8737.0	13055.2
	Squaring	489.0	495.4	671.3	736.9	1281.5
Inv	EEA	20016.8	32221.7	40065.7	71128.8	119246.4
	MAIA	34388.5	48682.6	68747.4	110253.6	184128.1
	EEA(opt.)	14989.6	25139.5	32898.0	60997.9	106894.7

* The numbers were measured on a Pentium 4 system. The unit is ns.

3.1 Addition

As expected, the addition of polynomials is the fastest field operation. For each field, Table 1 shows two algorithms, one for the addition of two polynomials and the other for three polynomials. As we can see, adding three polynomials is faster than invoking the addition of two polynomials twice. The addition of three polynomials reduces the number of memory accesses because the results from the first addition do not have to be saved to memory and then loaded back for the second addition. Combining two additions in one loop also reduces the control overhead. We did similar optimizations to the shift left and add operations.

3.2 Modular reduction

Since all the irreducible polynomials in ECDSA have either three or five terms, we use an array to store the position of the terms.

Three reduction algorithms discussed in Section 2.2 are compared in Table 1. The first algorithm (by bit) eliminates one term at a time while the second one (by word) eliminates up to 32 terms each time as the word size on Pentium 4 is 32. The second algorithm is about 27x to 37x faster than the first algorithm. The third algorithm also eliminates 32 terms at a time, but is optimized specifically for a fixed irreducible polynomial. The third algorithm is the fastest of the three for all the key sizes. However, it requires different implementations for different polynomials and for processors of different word sizes.

Generally, field operations become slower as the operand size increases. However, we noticed that the modular operations may be faster on larger operands. The reason is that the irreducible polynomials have different numbers of terms. The irreducible polynomials in $GF(2^{233})$ and $GF(2^{409})$ have three terms while the others have five terms.

3.3 Multiplication

Three multiplication algorithms are compared in Table 1. The first one is the right-to-left comb algorithm. As we discussed in Section 2, it is slightly faster than the left-to-right comb algorithm because the shift operations are not performed on the partial product. The second multiplication algorithm is the left-to-right comb algorithm with a window size of four bits. With the space overhead for storing 16 precomputed products, the second algorithm is 1.8 to 2.2 times faster than the right-to-left comb algorithm.

The third multiplication algorithm presented in Table 1 is the squaring algorithm. Our implementation utilizes a table of 16 bytes to insert 0's into a group of four bits. As we can see, squaring is much faster than regular multiplications. When m = 163, the squaring is about six times faster than the comb algorithm with a window of 4 bits. When m increases, the ratio becomes even large.

A modular reduction is needed in the multiplication algorithms. We used the second modular reduction algorithm (by word) that eliminates 32 terms at a time.

3.4 Inversion

The inverse operation is the most time-consuming field operation. Table 1 compares three inverse algorithms. The first one is EEA and the second one is MAIA. Although MAIA may take fewer iterations than EEA, our implementations show that MAIA is about 50% to 70% slower than EEA for all the key sizes. Our results confirm the findings in [6], although they are contrary to those in [14][15].

By profiling EEA, we noticed that most of the execution time was spent on two functions, `b_length` and `b_shiftleft_xor`. `b_length` searches for the highest 1 in a polynomial. It is used to compute the degree of polynomials. `b_shiftleft_xor` shifts a polynomial to the left and adds the result to another polynomial. It is used to perform the shift and add operations in EEA. When m = 163, `b_length` is accounted for 55.6% of the total execution time, and `b_shiftleft_xor` for 25.5%.

To accelerate EEA, we tried to minimize the overhead on determining the degree of polynomials. In addition to optimizing `b_length`, we also reduced the number of times we needed to call `b_length`. After the improvements, the time spent on `b_shiftleft_xor` increased to 41.0%, and that on `b_length` decreased to around 23.8%. When m = 163, the optimized EEA has a speedup of 1.3 over the original implementation.

The inversion is still the slowest field operation even after our improvements. It has been proposed to utilize projective coordinate systems to avoid expensive inversions [6][9][16]. In the affine coordinate system we currently use, a point addition needs one inversion and at most two multiplications. In the project coordinate system, however, a point addition needs more than 13 multiplications. Unless an inversion is 11 times slower than a multiplication, it is not worth to adopt the projective coordinate system. In our implementation, the optimized EEA is about 2.5 times slower than the right-to-left comb multiplication in a 32-bit system for m=163.

3.5 Field operations with different word sizes

Table 2 lists the performance of 163-bit binary operations with different word sizes. Generally, as the word size decrease, the time needed for each operation is increased. Modular reduction by bit is an exception. In this algorithm, the function that searches for the highest 1 in a word is accounted for a significant portion of the total execution time and the function is slower on 32-bit words than on 8-bit words. Another exception is that right-to-left comb algorithm is faster with 16-bit words than with 32-bit words. This is because when the word size decreases, the number of shift operations in the comb algorithm deceases although each shift operation takes longer time. The performance of inversion degrades faster than that of multiplication. With 32-bit words, the inversion is only three times slower than the right-to-left comb algorithm. When the word size changes to 8 bits, the inversion is about five times slower.

Table 2. Performance of 163-bit binary field operations

	Word size	32	16	8
Add	Two poly.	31.6	37.9	74.6
	Three poly.	37.5	59.0	78.6
Mod	By bit	8467.0	7093.2	5855.2
	By word	285.8	472.3	681.8
	Fixed poly.	40.9	63.1	129.4
Mul	R-to-L comb	5870.5	5219.4	8492.1
	L-to-R comb, 4-bit window	3118.4	3794.6	7126.7
	Squaring	489.0	770.1	804.7
Inv	EEA	20016.8	36687.2	48250.9
	MAIA	34388.5	41654.5	75933.0
	EEA(opt.)	14989.6	31408.0	39705.4

3.5 Performance of point multiplication

Table 3 summarizes the performance of scalar point multiplications. The algorithm we implemented here is the basic binary algorithm that requires about *m*/2 additions and *m* doublings. The binary field arithmetic algorithms we used include the right-to-left algorithm for filed multiplications and the “by word” modular reduction. As for inverse, the results of both the original and the optimized EEA are presented. Please note that all the algorithms we chose do not require precomputations. This is critical for systems with a small amount of memory. Also, the algorithms are not specific for processors of a particular word size.

Table 3. Performance of scalar point multiplications*

Word size	Algorithm	*M*				
		163	233	283	409	571
8-bit	EEA	18.6	45.6	74.8	195.3	490.5
	EEA(opt.)	14.2	35.1	58.1	154.7	397.7
	Speedup	1.31	1.30	1.29	1.26	1.23
16-bit	EEA	14.3	33.6	53.2	147.3	358.7
	EEA(opt.)	10.6	25.7	41.0	118.0	294.5
	Speedup	1.35	1.31	1.30	1.25	1.22
32-bit	EEA	10.7	21.6	30.5	75.4	169.0
	EEA(opt.)	6.1	13.2	19.2	49.9	114.5
	Speedup	1.75	1.64	1.59	1.51	1.48

* The unit of time is ms.

We compiled the same set of algorithms for three different word sizes, 8, 16, and 32 bits, and measured the performance on a Pentium 4 at 3 GHz. For each word size, the first two rows are the execution time of a scalar point multiplication, and the unit is millisecond. The first row is for the original EEA and the second row for the optimized EEA. The third row is the speedup of the optimized EEA over the original EEA. We can see that the optimized EEA accelerates the scalar point multiplication significantly. The speedups with 32-bit words range from 1.48 to 1.75, and those with 8-bit words are less, ranging from 1.23 and 1.31.

As expected, ECC has a better performance with a larger word size, and the performance decreases as the key size increases. However, the performance degradation is worse with small word sizes. A 571-bit multiplication is 18.8 times slower than a 163-bit multiplication with 32-bit words while the ratio is 28 times with 8-bit words.

4. ECC on an 8-bit Processor

In this section, we evaluate our ECC implementations on an 8-bit processor Atmega 128 [17]. Atmega 128 is a low-power microcontroller based on the AVR architecture. It contains 128 KB of FLASH program memory and 4 KB of data memory. ATmega128 can be operated at frequencies up to 16 MHz and can finish most instructions in one cycle. Atmega 128 and other microcontrollers in its family have been used in many wireless sensor networks [17][18].

Our ECC implementation can be compiled in 8-bit mode for the AVR architecture with avr-gcc 3.3.2. We simulated the binary code with Avrora [19], a set of simulation and analysis tools developed at UCLA for the AVR microcontrollers. On Atmega 128, it takes about 151 million cycles to perform a 163-bit scalar multiplication with the original EEA, as shown in Table 4. Assuming a clock rate of 8 MHz, it is about 19.0 seconds per multiplication. The optimized EEA reduces the number of cycles to 133 million and takes 16.7 seconds.

Table 4. Performance of 163-bit ECC on Atmega 128

	Number of cycles	Time (sec.)
EEA	151,796,532	19.0
EEA(opt.)	133,475,171	16.7
Fast modular reduction	111,183,513	13.9

We then tried to improve the performance of ECC by replacing the general modular reduction with the fastest algorithm that is fixed for the particular irreducible polynomial. The substitution results in a speedup of 1.2 over the optimized EEA. The scalar point multiplication can now be done in 13.9 seconds.

The memory usage of our implementations is summarized in Table 5. The implementation reported in [5] takes 34 seconds and needs 34k bytes of memory. Our implementation runs more than twice faster and needs approximately a third of the memory.

Table 5. Memory usage of ECC

Data type	Size (byte)
.data	396
.text	11,592
.bss	424
Total	12,412

Table 6 lists the execution time percentage of the three most time-consuming operations in 163-bit ECC. 69.5% of the execution time is on inversion, more than twice as much as multiplication (25.7%) and squaring (3.6%) combined.

Table 6: Important operations in 163-bit ECC

Operations	Time
Multiplication	25.7%
Squaring	3.6%
Inversion	69.5%

5. Conclusions and Discussions

In this paper, we studied the software implementations of ECC. We first examined the algorithms for ECC over binary filed. After comparing algorithms for the major field operations that are required in ECC, we identified a set of efficient algorithms suitable for resource constrained systems. We also compared the performance of these algorithms for different word sizes. Finally, we simulated our implementation on an 8-bit microcontroller. Our implementation is more than twice faster than the previous results without instruction set architecture extensions or hardware accelerations. We can perform a 163-bit scalar point multiplication in 13.9 seconds.

We have only implemented ECC with the C language. We expect a significant improvement by simply implementing the algorithms with assembly language. For example, the shift-and-add operation, which takes 43% of the total execution time, can be performed much more efficiently with hand-coded instructions on most processors.

Accounted for about 70% of the execution time, an efficient EEA is important to ECC. In the future, we will study the architecture support for fast polynomial inversion.

References

[1] N. Koblitz, “Elliptic curve cryptosystems”, *Mathematics of Computation*, vol. 48, pp. 203-209, 1987.

[2] V. Miller, “Uses of elliptic curves in cryptography”, *Advances in Cryptology: proceedings of Crypto’85*, pp. 417-426, 1986.

[3] National Institude of Standards and Technology, *Digital Signature Standard*, FIPS Publication 186-2, February 2000.

[4] Y. Choi, H. W. Kim, and M. S. Kim, “Implementation of Elliptic Curve Cryptographic coprocessor over GF(2^{163}) for ECC protocols,” *Proceedings of the 2002 International Technical Conference on Circuits/Systesm, Computers, and Communications*, pp. 674-677, July 2002.

[5] D. J. Malan, M. Welsh, M. D. Smith, “A public-key infrastructure for key distribution in TinyOS based on elliptic curve cryptography,” *First Annual IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks*, pp. 71-80, October 2004.

[6] D. Hankerson, J. Hernandez, A. Menezes, “Software implementation of Elliptic Curve Cryptography over binary fields,” *Proceedings of Workshop on Cryptographic Hardware and Embedded System*, vol. 1965 of Lecture Notes in Computer Science, pp. 1-24, 2000.

[7] J. Lopez, R. Dahab, “High-speed software multiplication in F(2^m)”, *Proceedings of INDOCRYPTO*, pp. 203-212, 2000.

[8] N. Gura, A. Patel, A. Wander, H. Eberle, S. C. Shantz, “Comparing elliptic curve cryptography and RSA on 8-bit CPUs,” *Proceedings of Cryptographic Hardware and Embedded Systems (CHES)*, pp. 119-132, 2004.

[9] J. Lopez, R. Dahab, “Improved algorithms for elliptic curve arithmetic in GF(2n)”, *Selected Areas in Cryptography – SAC’98*, vol. 1556, pp. 201-212, 1999.

[10] S. Zhu, S. Setia, S. Jajodia, “LEAP: efficient security mechanisms for large-scale distributed sensor networks,” *Technique Report*, August 2004.

[11] C. Karlof, N. Sastry, D. Wagner, “TinySec: a link layer security architecture for wireless sensor networks,” *SenSys’04*, November 2004.

[12] A. Perrig, R. Szewczyk, et al., "SPINS: security protocols for sensor networks", *Wireless Networks*, Vol.8, No. 5, pp. 521-534, September 2002.

[13] I. Blake, G. Seroussi, N. Smart, *Elliptic Curves in Cryptography*, Cambridge University Press, 1999.

[14] R. Schroeppel, H. Orman, S. O’Malley and O. Spatscheck, “Fast key exchage with elliptic curve systems”, *Advances in Cryptogaphy – Crypto ’95*, pp. 43-56, 1995.

[15] J. Solinas, “Efficient arithmetic on Koblitz curves”, *Designs, Codes and Cryptography*, vol. 19, pp. 195-249, 2000.

[16] D. Chudnovsky, G. Chudnovsky, “Sequences of numbers generated by addition in formal groups and new primality and factoring tests”, *Advances in Applied Mathematics*, vol. 7, pp. 385-434, 1987.

[17] Atmel Corp., *8-bit Microcontroller with 128K Bytes In-System Programmable Flash: ATmega 128*, 2004.

[18] I. Crossbow Technology, "MICA2: wireless measurement system", http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/6020-0042-4_A_MICA2.pdf.

[19] Ben L. Titzer, Daniel K. Lee, Jens Palsberg, “Avrora: scalable sensor network simulation with precise timing”, *Proceedings of IPSN*, 2005.