Software Implementations of Elliptic Curve Cryptography

Zhijie Jerry Shi and Hai Yan
(Corresponding author: Zhijie Jerry Shi)

Computer Science and Engineering Department, University of Connecticut
371 Fairfield Way, Unit 2155, Storrs, CT 06269, USA (Email: zshi@engr.uconn.edu)

(Received Sep. 13, 2006; revised and accepted Nov. 24, 2006)

Abstract

Elliptic Curve Cryptography (ECC) is a promising alternative for public-key algorithms in resource-constrained systems because it provides a similar level of security with much shorter keys than conventional integer-based public-key algorithms. 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 used in small computing devices such as sensor nodes. In this paper, we studied software implementations of ECC. We first investigated whether some architectural parameters such as word size may affect the choice of algorithms when implementing ECC with software. We identified a set of algorithms for ECC implementation for low-end processors. We also examined several improvements to the instruction set architecture of an 8-bit processor and studied their impact on the performance of ECC.

Keywords: Binary field arithmetic, ECC, instruction set architecture, sensor networks, software implementation

1 Introduction

Elliptic Curve Cryptography (ECC), proposed independently in 1985 by Neal Koblitz [10] and Victor Miller [14], 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) [15]. 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 algorithms, especially in resource-constrained systems such as sensor nodes and mobile devices.

Considerable work on ECC has been focused on mathematical methods and algorithms, hardware implementations, and extensions of instruction set architecture [3, 7, 8, 11, 12, 13]. Hankerson et al. discussed the software implementations of in ECC in [8], in which they focused on 32-bit processors. Gura et al. compared the performance of ECC and RSA on 8-bit processors [7]. But the elliptic curves they studied are in $GF(p)$. Malan recently investigated the feasibility of implementing ECC in sensor nodes [13]. Their implementation was not optimized well. Given that the performance of ECC on low-end processors is far from being satisfactory, many protocols designed for wireless sensor networks tend to use symmetric-key algorithms only [9, 16, 20].

In this paper, we try to identify the problems in the software implementations of ECC and explore techniques that can accelerate the software implementations. We focus on ECC over $GF(2^m)$. Since each operation in ECC has many different ways to implement, we first investigated whether processor word size may affect our choice of algorithms. We selected a set of efficient algorithms and studied their performance on processors of different word sizes.

We also studied how Instruction Set Architecture (ISA) improvements affect the performance of ECC on low-end 8-bit processors. We examined three instructions, binary field multiplications, shift operations, and most significant 1, which target the time consuming operations in ECC. Although some instructions, such as binary field multiplication instructions, can achieve a large speedup on 64-bit processors [6], they are not as effective on low-end processors if affine coordinates are adopted. The architectural supports also result in different selections of algorithms. For example, when binary field multiplication instructions are supported, it is preferable to adopt projective coordinates to achieve better performances. On 8-bit microcontrollers running at 16 MHz, we can perform a 163-bit scalar point multiplication in 0.85 seconds.

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 pro-

cessors and examine the impact of word size on ECC algorithms. In Section 4, we discuss the performance of ECC on 8-bit processors. Section 5 discusses three ISA improvements for accelerating ECC. Section 6 concludes the paper.

2 Overview of ECC

Elliptic curves can be defined in many fields including prime fields $GF(p)$ and finite fields $GF(2^m)$ of characteristic two, which are also called binary fields [2]. The elliptic curves over binary field are of special interest because the operations in a binary field are faster and easier to implement. In this paper, we focus on ECC defined in binary field. More specifically, we focus on the five binary field elliptic curves specified in the Elliptic Curve Digital Signature Algorithm (ECDSA) [15], which are defined in $GF(2^{163}), GF(2^{233}), GF(2^{283}), GF(2^{409})$, and $GF(2^{571})$.

2.1 ECC Operations and Parameters

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

$$y^2 + x \bullet y = x^3 + a \bullet x^2 + b,$$

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:

$$P + O = O + P = P,$$
$$P + (-P) = O, where - P = (x, x + y).$$

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

$$\lambda = \begin{cases} \frac{y_2+y_1}{x_2+x_1}, & if P \neq Q \\ \frac{y_1}{x_1} + x_1, & if P = Q. \end{cases} \quad (1)$$

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

$$Q = kP = \underbrace{P + \ldots + P}_{k\ times}.$$

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

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

The scalar point multiplication can be done with the basic binary algorithm. For example, in the left-to-right binary algorithm shown in Figure 1, 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 of ECC algorithms.

Algorithm 1: Left-to-right binary algorithm for point multiplication
INPUT: $P \in E$, and a positive integer $k = (k_{l-1}\ k_{l-2} \ldots k_1 k_0)_2$
OUTPUT: $A = kP$
1. $A \leftarrow O$
2. For $i = l-1$ down to 0 do
   2.1 $A \leftarrow 2A$
   2.2 If $k_i = 1$ then $A \leftarrow A + P$
3. Return A

Figure 1: Basic left-to-right binary algorithm

The scalar point multiplication is the main workload of ECC algorithms. The binary algorithm for scalar point multiplications is similar to the algorithm for exponentiations in $GF(p)$. Therefore, the techniques, such as precomputations and sliding windows that accelerate integer exponentiations, can also be applied in similar ways to accelerating scalar point multiplications. Figure 2 shows the m-ary method for the scalar point multiplication of ECC, where $m = 2^r$ for some integer $r \geq 1$, and $d = \lceil l/r \rceil$, where $\lceil \cdot \rceil$ denotes the ceiling function. The binary method can be treated as a special case of m-ary method with $r = 1$. The total number of doublings does not change much because Step 4.1 of Algorithm 2 requires r point doublings. Nevertheless, the number of point additions is reduced from l to d (for the worst cases) at the cost of more memory space to store the precomputed results. If the point P is the same for many multiplications, the precomputations need to be done only once.

Algorithm 2: m-ary method for point multiplication
INPUT: $P \in G$, and a positive integer $k = (k_{l-1}\ k_{l-2} \ldots k_1 k_0)_2$
OUTPUT: $A = kP$
Precomputation for window size of r and $m = 2^r$
1. $P_0 \leftarrow O$
2. For $i = 1$ to $m - 1$ do
   2.1 $P_i \leftarrow P_{i-1} + P$
3. $A \leftarrow O$
Main loop
4. For $j = d - 1$ down to 0 do
   4.1 $A \leftarrow mA$
   4.2 $A \leftarrow A + P_i$ where $i = (k_{r(j+1)-1} \ldots k_{rj})_2$
5. Return A

Figure 2: m-ary method for point multiplication

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 exclusive-or (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 as shown in Figure 3, which is similar to the method for normal binary multiplications. The difference is that the add operations are in $GF(2^m)$. When multiplying two polynomials $a(x)$ and $b(x)$, we can first set the partial product $c(x)$ to 0 if $a_0 = 0$ or to $b(x)$ if $a_0 = 1$. Then we scan the bits in $a(x)$ from a_1 to $a_m - 1$. For each bit, $b(x)$ is first shifted to the left by one. If the scanned bit in $a(x)$ is 1, the new value of $b(x)$ is also added to the partial product C. The modular reduction can be integrated into the shift-and-add multiplication. After each shift operation, the degree of $b(x)$ is checked. If $b(x)$ has a degree of m, it can be reduced $tob(x) + f(x)$, where $f(x)$ is the irreducible polynomial. This method is suitable for hardware implementations where the shift operation can be performed in parallel. However, it is less desirable for software implementations because shifting a polynomial stored in multiple words is a slow operation that requires many memory accesses.

Algorithm 3: Right-to-left shift-and-add field multiplication

INPUT: Binary polynomials $a(x)$ and $b(x)$ of degree at most m–1
OUTPUT: $c(x) = a(x) \bullet b(x)$ *mod* $f(x)$

1. If $a_0 = 1$ then $c(x) \leftarrow b(x)$; else $c(x) \leftarrow 0$
2. For $i = 1$ to $m - 1$ do
   2.1 $b(x) \leftarrow b(x) \bullet x \bmod f(x)$
   2.1 If $a_i = 1$ then $c(x) \leftarrow c(x) + b(x)$
3. Return $c(x)$

Figure 3: Shift-and-add algorithm for field multiplication

The comb algorithms are normally used for fast polynomial multiplications [8]. Algorithm 4 in Figure 4 illustrates the right-to-left comb algorithm. Suppose $a(x)$ and $b(x)$ are two polynomials stored in t words and each word consists of w bits. Unlike the shift-and-add algorithm, which scans the bits in $a(x)$ one by one sequentially, the comb algorithm first tests bit 0 of all the words in $a(x)$, 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. Note that $c(x) + b(x) \bullet x^{jw}$ in Step 2.1 can be performed by aligning $b(x)$ with proper words in $c(x)$. The only step that needs shift operations is Step 2.2. Compared to the shift-and-add method, the comb algorithm reduces the number of shift operations from $m-1$ to $w-1$.

Algorithm 4: Right-to-left comb method for polynomial multiplication

INPUT: Binary polynomials $a(x)$ and $b(x)$ of degree at most m–1
OUTPUT: $c(x) = a(x) \bullet b(x)$

1. $c(x) \leftarrow 0$
2. For $k = 0$ to $w - 1$ do
   2.1 For $j = 0$ to $t - 1$ do
      If bit k of $a[j]$ is 1 then $c(x) \leftarrow c(x) + b(x) \bullet x^{jw}$
   2.2 If $k \neq w - 1$ then $b(x) \leftarrow b(x) \bullet x$
3. Return $c(x) \bmod f(x)$

Figure 4: Right-to-left comb method for field multiplication

The left-to-right comb algorithm, as shown in Figure 5, is similar to the right-to-left comb algorithm, but it tests the bits in the words of $a(x)$ 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 $c(x)$ in Step 2.2, not on $b(x)$. Because $c(x)$ is twice as long as $b(x)$, the left-to-right comb algorithm is a little bit slower.

Algorithm 5: Left-to-right comb method for polynomial multiplication

INPUT: Binary polynomials $a(x)$ and $b(x)$ of degree at most m–1
OUTPUT: $c(x) = a(x) \bullet b(x)$

1. $c(x) \leftarrow 0$
2. For $k = w - 1$ down to 0 do
   2.1 For $j = 0$ to $t - 1$ do
      If bit k of $a[j]$ is 1 then $c(x) \leftarrow c(x) + b(x) \bullet x^{jw}$
   2.1 If $k \neq 0$ then $c(x) \leftarrow c(x) \bullet x$
3. Return $c(x) \bmod f(x)$

Figure 5: Left-to-right comb method for field multiplication

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 [8]. By scanning the bits in $a(x)$ with a window of a fixed size, the algorithm can multiply more than one bit with $b(x)$ at a time. The partial product $c(x)$ is then shifted to left by the window size. The products of $b(x)$ and every possible value of bits in a window are precomputed and stored in a table. Algorithm 6 shows the left-to-right comb method with windows of width $s = 4$, where the number of windows in one word is $d = \lceil w/s \rceil$. The sliding window method reduces the number of shifts at the cost of storage overhead. A larger window size leads to fewer shift operations but requires more space to save the precomputed results.

Algorithm 6: Left-to-right comb method with windows of width $s = 4$

INPUT: Binary polynomials $a(x)$ and $b(x)$ of degree at most m–1
OUTPUT: $c(x) = a(x) \bullet b(x)$
1. Compute $B_u = u(x) \bullet b(x)$ for all polynomials $u(x)$ of degree ≤ 3
2. $c(x) \leftarrow 0$
3. For $k = d - 1$ down to 0 do
  3.1 For $j = 0$ to $t - 1$ do
    Let $u = (u_3, u_2, u_1, u_0)$, where u_i is bit $(4k + i)$ of $a[j]$
    $c(x) \leftarrow c(x) + B_u \bullet x^{jw}$
  3.2 If $k \neq 0$ then $c(x) \leftarrow c(x) \bullet x^4$
4. Return $c(x) \bmod f(x)$

Figure 6: Left-to-right comb method with windows

The squaring of a polynomial is much efficient than normal multiplications, taking advantage of the fact that the representation of $a(x)^2$ can be obtained by inserting 0's between consecutive bits in $a(x)$'s binary representation [17]. For example, if $a(x)$ is represented with 0xFFFF, $a(x)^2 = 0x55555555 \bmod f(x)$.

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 $c(x)$ be a polynomial of degree i where $i \leq 2m - 2$, and $f(x)$ be the irreducible polynomial of degree m. We can reduce the degree of $c(x)$ by eliminating the highest term x^i.

$$c(x) = c(x) + f(x) \bullet x^{i-m}. \quad (2)$$

The process is repeated until the degree of $c(x)$ is smaller than m.

If the irreducible polynomial $f(x)$ is stored in memory like regular polynomials, i.e., its coefficients are packed into words, Formula (2) can be done by first shifting $f(x)$ to the left by $(i - m)$ bits, and then adding the result to $c(x)$. 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 $f(x) \bullet x^j$ for $j = 0, 1 \ldots w - 1$, where w is the number of bits in a word [8]. This technique is adopted in Algorithm 7 shown in Figure 7. In the figure, Step 1 performs the precomputation. If $f(x)$ is the same for many reductions, Step 1 does not have to be done for every reduction. In Step 2, bits in $c(x)$ are scanned one by one from the highest 1. The 1's are eliminated by adding $f(x)$ to $c(x)$. Because of the precomputation, Step 2.1 does not require shift operations on $f(x)$.

If $f(x)$ 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 Formula (2), we can calculate the position of the terms after the shift left operation and flip the corresponding bits in $c(x)$.

A faster modular reduction algorithm was described in [8]. It works for irreducible polynomials in which the difference between the degrees of the first two terms is larger than the word size. An example of such polynomials is $f(x) = x^{163} + x^7 + x^6 + x^3 + 1$, in which

Algorithm 7: Modular reduction (one bit at a time)

INPUT: Binary polynomials $c(x)$ of degree at most $2m$–2
OUTPUT: $c(x) \bmod f(x)$
1. Compute $u_k(x) = f(x) \bullet x^k$, for $0 \ ? \ k \ ? \ w - 1$
2. For $i = 2m - 2$ down to m do
  2.1 If $c_i = 1$ then
    Let $j = (i - m) / w$ and $k = (i - m) - w\,j$
    $c(x) \leftarrow c(x) + u_k(x) \bullet x^{jw}$
3. Return $(c[t–1], \ldots, c[1], c[0])$

Figure 7: Modular reduction one bit per time

the degree of the leading term is much larger than that of the second term. With these types of irreducible polynomials, multiple terms in $c(x)$ can be eliminated at a time. In typical software implementations, w terms can be eliminated at the same time, where w is the word size.

Inversion. The calculation of λ in Formula (1) needs a division in $GF(2/6m)$, which is normally done with an inversion followed by a multiplication. To divide $a(x)$ by $b(x)$, we first obtain $b(x)^{-1}$, the inverse of $b(x)$, and then compute $a(x) \bullet b(x)^{-1}$. The classic algorithm for computing the multiplicative inverse is Extended Euclidean Algorithm (EEA) [8].

The Almost Inverse Algorithm (AIA) in [17] is based on EEA. But unlike EEA, AIA eliminates the 1 bit 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 [8]. We do not list the details of these algorithms. The reader is referred to references [8] and [17].

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 binary elliptic curves that NIST recommended for ECDSA [15].

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.

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.

3.2 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 irreducible 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 larger.

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 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 [8], although they are contrary to those in [17, 18].

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 obtain 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` accounts 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 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 after the improvements. Projective coordinate systems are options to avoid expensive inversions with the cost of more multiplication operations [4, 8, 11]. One of the fastest projective coordinate systems is the one proposed by Lopez et al. in [11]. In this system, a projective point (X, Y, Z), $Z \neq 0$ corresponds to the affine point $(X/Z, Y/Z^2)$ and the equation of the elliptic curve is changed to

$$Y^2 + X \bullet Y \bullet Z = X^3 \bullet Z + a \bullet X^2 \bullet Z^2 + b \bullet Z^4.$$

To compare different coordinate systems, we look at how many multiplications and inversions need to be done in a point doubling or addition operation. We do not count squaring operations because they are much faster than multiplications. In the affine coordinate system we have discussed so far, a point addition needs one inversion and two multiplications. In the projective coordinate system, a point addition does not require inversion; however, it needs 14 multiplications, as shown in Table 2 [8]. In the table, I and M denote inversion and multiplication, respectively. In the left-to-right binary algorithm, point additions are done with mixed coordinates as the intermediate results are represented with projective coordinates

Table 2: Cost of point additions and doublings[8]

Coordinate system	General addition	General addtion (mixed coordinates)	Doubling
Affine	$1I, 2M$	-	$1I, 2M$
Projectuve $(X/Z, Y, Z^2)$	$14M$	$9M$	$4M$

while the base point is with the affine coordinates. Assume that half of the scalar bits are 1's. The projective coordinate can achieve a better performance only when a field inversion is at least 3.7 times slower than a multiplication. When the right-to-left binary algorithm is adopted, however, both the intermediate point and the base point have to be represented with projective coordinates. Hence, the projective coordinates outperform the affine coordinates only when a field inversion is 5.3 times slower than a multiplication. In our implementations, the optimized EEA is about 2.6 times slower than the right-to-left comb multiplication in a 32-bit system for $m = 163$. When $m = 571$, the cost ratio of inversion and multiplication is about 3.66. Therefore, adopting projective coordinates does not provide performance advantages in these cases.

3.5 Field Operations with Different Word Sizes

We now examine how the word size of processors affects the choice of algorithms. Table 3 lists the performance of 163-bit binary field operations with different word sizes.

Table 3. 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

Generally, as the word size decreases, the time needed for each operation increases. Modular reduction by bit is an exception. In this algorithm, the function that searches for the highest 1 in a word accounts for a significant portion of the total execution time. As a result, 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. Overall, the difference between the right-to-left comb algorithm and the left-to-right comb algorithm with 4-bit windows becomes smaller as the word size reduces to 8.

We also noticed that the performance of inversion degrades faster than that of multiplication. With 32-bit words, the inversion is only 2.6 times slower than the right-to-left comb algorithm. When the word size changes to 8 bits, the inversion is about 4.7 times slower. As a result, adopting projective coordinates would provide better performance although more storage space is needed.

3.6 Performance of Point Multiplication

Table 4 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 4. 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. Although the physical word size is always 32-bit on Pentium 4, we use only 8 bits (or 16 bits) of the datapath when the specified

word size is 8 bits or 16 bits. For each word size, the first two rows are the execution time of a scalar point multiplication in 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. We can see that the optimized EEA accelerates the scalar point multiplication significantly. The speedups range from 1.48 to 1.75 on 32-bit systems. The speedups are smaller when the word size is 8, 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

This section evaluates the ECC implementations on an 8-bit processor Atmega 128 [1]. 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 execute most instructions in one cycle. Atmega 128 and other microcontrollers in its family have been used in many wireless sensor networks [1, 5].

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 5. 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.

We can improve the performance of ECC by replacing the general modular reduction with the fastest algorithm 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.

Table 5: 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
EEA (opt.) with fast modular reduction	111,183,513	13.9
Projective coordinates	120,072,952	15.0
Projective coordinates with fast modular reduction	97,770,030	12.2

Since the cost ratio of inversion and multiplication is around 4.7 on 8-bit processors, projective coordinates have better performance than affine coordinates. This can be seen in Table 5. The projective coordinate performed slightly (about 12%) better than affine coordinates. However, more memory space is required to store points in projective coordinates.

The memory usage of our implementations is summarized in Table 6. The implementation reported in [13] takes 34 seconds and needs 34 KB of memory. Our implementation is about 2.7 times faster and needs less than half the memory.

Table 6: Memory requirements of ECC

Sections	Size (byte)
.data	142
.text	10462
.bss	674
.stab	2448
.stabstr	2029
Total	15755

Table 7 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 7: Important operations in 163-bit ECC

Operations	Time
Multiplication	25.7%
Squaring	3.6%
Inversion	69.5%

5 Architectural Support for ECC

This section investigates several architectural supports for ECC and then discusses how these supports will affect our choice of coordinate systems.

5.1 Extension of Instruction Set Architecture

We evaluated the performance of ECC on the Atmega system with several ISA improvements, hoping to find cost-effective methods for improving ECC's performance on 8-bit processors. We compared improved systems with the baseline, the best result in Table 5. The comparison is presented in Table 8. The first thing we did is to rewrite critical routines in ECC with assembly language, by which we achieved a speedup of 1.18. The assembly code also provides a basic infrastructure for exploring new instructions. Based on the assembly code, we experimented with three instructions: binary field multiplications, shift instructions with variable shift amounts, and most-significant 1 instruction.

While integer multiplication instructions are available in most processors, the multiplications in $GF(2^m)$, which are needed in ECC over binary field, are not supported efficiently. Recently, some new instructions have been proposed to accelerate binary field multiplications in processors. For example, binary field multiplication instructions can achieve a speedup of more than 20 times on 64-bit processors [6].

Table 8: Comparison of architectural techniques

Method	Number of cycles	Speedup
Baseline	111,183,513	1
Critical routines in assembly language	94,223,316	1.18
Hardware supported $GF(2^m)$ multiplication	86,862,119	1.28
Shift by multiple bits	36,938,044	3.01
Most significant bit 1 instruction	92,652,927	1.20
All ISA improvements	28,581,879	3.89

To evaluate the effectiveness of binary field multiplication instructions on 8-bit processors, we modified the Avrora simulator and included binary filed multiplication instructions. As described in [6], the multiplication can be supported in three different ways. 1) The higher and lower halves of the product are written in parallel to two registers. 2) Two separate instructions write either the lower or the higher half of the product to two registers. 3) Only one instruction that generates the lower half of the product is defined. Thus, a full multiplication also requires a bit-level reverse instruction that reverses the order of bits in a register.

Our implementation took method 1. We included an instruction that generates both higher and lower halves of the product and places them into registers `R0` and `R1`. For example, the following instruction multiplies two 8-bit source registers `RS0` and `RS1` and stores the 16-bit product into `R0` and `R1`.

```
GFMUL  RS0, RS1
```

We found that the hardware supported binary filed multiplication is not very effective in accelerating ECC on 8-bit processors if the inverse operation is performed. It achieves a speedup of 1.28 over baseline and gains only 8% improvement over the assembly version. The reason is that the inverse operation accounts for the most of the execution time. On the other hand, the hardware supported binary field multiplications do improve the performance of polynomial multiplications and thus change the performance ratio of inversion and multiplication. As a result, projective coordinates become a better choice and provide better performance.

As mentioned in Sections 3.3 and 3.4, the shift instruction is used extensively in the implementation of ECC. However, the shift instruction in the AVR instruction set can shift only one bit each time. Thus we added a shift instruction that supports an arbitrary shift amount. This type of instructions is supported in many processors but not available in existing AVR instruction set. Because of the short instruction words, the shift amount cannot be specified in the shift instruction explicitly. In our implementation, we decided to use register `R0` to store the shift amount. For example, the following two instructions shift `Rs` to left by 5 bits and store the result in `Rd`.

```
  LDI    R0, 5
SHIFT.L  Rd, Rs
```

The shift instruction gives us the best performance improvement of the three methods we evaluated. It achieves a speedup of 3.01 as shown in Table 8. The shift instruction with arbitrary shift amounts only accelerates operations like inversion in which the shift amounts are often more than one. It does not help the multiplication operations in which shift amounts are always one. Because the inversion accounts for a large portion of the total execution time (see Table 7), the shift instruction results in a large speedup.

The third instruction we experimented is to find the index of the most significant 1 in a register. This instruction helps us determine the degree of a polynomial. As described in Section 3.4, `b_length` is an important function. The optimizations on the function have accelerated inversion by a speedup of 1.3. With the most significant 1 instruction, we can accelerate the inversion operation further. The most significant 1 instruction is defined as follows.

```
MSB  Rd, Rs
```

The instruction store in `Rd` the position of the most significant 1 in `Rs`. For example, when `Rs` $= 1$, `Rd` is 0. When `Rs` $= 8$, `Rd` is 3.

While integer multiplication instructions are available in most processors, the multiplications in $GF(2^m)$, which are needed in ECC over binary field, are not supported efficiently. Recently, some new instructions have been proposed to accelerate binary field multiplications in processors. For example, binary field multiplication instructions can achieve a speedup of more than 20 times on 64-bit processors [6].

Table 8 shows that the MSB instruction can increase the speedup over the baseline to 1.20 even if used alone. The benefit of MSB is not significant on 8-bit microprocessors because it is relatively fast to locate the most significant 1 among 8 bits. However, the instruction will have larger speedups on 32 or 64-bit microprocessors.

With all the three instructions, we reduced the scalar point multiplication of ECC on 8-bit microcontroller to 28,581,879 cycles, achieving a speedup of 3.89 over our baseline architecture.

5.2 Elliptic Curve Point Representations

The performance ratio of inversion and multiplication has changed as we added new instructions. As a result, adopting projective coordinates may become preferable. Table 9 illustrates how the performance of inversion and multiplication is affected by the new instructions.

With the new instructions, Inversion is 3.4 times faster than in the baseline architecture. Nevertheless, the multiplication operation has been accelerated even more. Compared with the baseline architecture, the new instructions result in a speedup of 9.6 for multiplications. Consequently, the performance ratio of inversion and multiplication has been increased significantly from 5.6 to 15.9.

Since the inversion is now much more expensive than the multiplication, it is desirable to adopt projective coordinates to represent the points on the curve. We adopted the projective coordinates discussed in Section 3.4 and implemented the left-to-right binary algorithm with the point additions in mixed coordinates. Table 10 shows the performance comparison between affine and projective coordinates. As expected, the projective coordinates have better performance. They are more than twice faster than affine coordinates and achieve a speedup of 8.23 over the baseline architecture. Assuming a clock rate of 16MHz, a 163-bit ECC can be performed in about 0.85 seconds.

Table 10: Performance for different coordinates

Coordinate system	Number of cycles	Speedup
Affine	28,581,879	3.89
Projective $(X/Z, Y/Z^2)$	13,515,076	8.23

∗Compared with the baseline performance in Table 8

6 Conclusions and Discussions

In this paper, we studied the software implementations of ECC and examined how some architectural features, such as word size and ISA, affect the performance 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. The change of word sizes result in different choices of algorithms. We simulated our implementations on an 8-bit microcontroller. Our implementations are more than twice faster than previous results without instruction set architecture extensions or hardware accelerations.

We also evaluated three instructions for accelerating ECC: binary field multiplications, shift with arbitrary shift amounts, and index of most significant 1. Combining all three instructions, we can achieve a speedup of 3.89. More importantly, the new instructions make the projective coordinates a better choice for point representations. The projective coordinates achieves a speedup of 8.23 over the baseline architecture. It takes about 0.85 seconds to perform a 163-bit scalar point multiplication on 8-bit AVR processors at 16MHZ.

We only focus on software implementation of ECC in this paper. Application-specific hardware can be integrated into processors to accelerate the multiplication and inversion operations further. When new hardware is implemented, the performance of multiplication and inversion should be evaluated to choose the best point representations for better performances.

References

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

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

[3] Y. Choi, H. W. Kim, and M. S. Kim, "Implementation of elliptic curve cryptographic coprocessor over $GF(2^{163})$ for ECC protocols," in *Proceedings of the 2002 International Technical Conference on Circuits/Systesm, Computers, and Communications*, pp. 674-677, July 2002.

[4] D. Chudnovsky and 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.

[5] I. Crossbow Technology, *MICA2: Wireless Measurement System*, http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/6020-0042-4_A_MICA2.pdf.

[6] A. M. Fiskiran and R. B. Lee, "Evaluating instruction set extensions for fast arithmetic on binary finite fields," in *Proceedings of the International Conference on Application-Specific Systems, Architectures, and Processors (ASAP'04)*, pp. 125-136, Sep. 2004.

[7] N. Gura, A. Patel, A. Wander, H. Eberle, and S. C. Shantz, "Comparing elliptic curve cryptography and RSA on 8-bit CPUs," *Proceedings of Cryptographic Hardware and Embedded Systems (CHES'04)*, pp. 119-132, 2004.

[8] D. Hankerson, J. Hernandez, and A. Menezes, "Software implementation of elliptic curve cryptography over binary fields," *Proceedings of Workshop on Cryptographic Hardware and Embedded System*, LNCS 1965, pp. 1-24, 2000.

[9] C. Karlof, N. Sastry, and D. Wagner, "TinySec: a link layer security architecture for wireless sensor networks," *SenSys'04*, pp. 162-175, Nov. 2004.

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

[11] J. Lopez and R. Dahab, "Improved algorithms for elliptic curve arithmetic in $GF(2n)$," in *Cryptography - SAC'98*, vol. 1556, pp. 201-212, 1999.

Table 9: Performance ratio of binary field inversion and multiplication

Method	Inversion (number of cycles)	Multiplication (number of cycles)	Inv./Mul. ratio
Baseline	397985	71580	5.6
Optimized	118642	7479	15.9

[12] J. Lopez and R. Dahab, "High-speed software multiplication in $F(2^m)$," *Proceedings of Indocrypto'00*, pp. 203-212, 2000.

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

[14] V. Miller, "Uses of elliptic curves in cryptography," Advances in *Cryptology: proceedings of Crypto'85*, pp. 417-426, 1986.

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

[16] A. Perrig, R. Szewczyk, et al., "SPINS: security protocols for sensor networks," *Wireless Networks*, vol.8, no. 5, pp. 521-534, Sep. 2002.

[17] R. Schroeppel, H. Orman, S. O'Malley, and O. Spatscheck, "Fast key exchage with elliptic curve systems," in *Advances in Cryptogaphy (Crypto'95)*, pp. 43-56, 1995.

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

[19] B. L. Titzer, D. K. Lee, and J. Palsberg, "Avrora: scalable sensor network simulation with precise timing," in *Proceedings of IPSN'05*, pp. 477-482, April 2005.

[20] S. Zhu, S. Setia, and S. Jajodia, *LEAP: Efficient Security Mechanisms for Large-Scale Distributed Sensor Networks*, Technique Report, Aug. 2004.

Z. Jerry Shi is an assistant professor in the Department of Computer Science and Engineering at the University of Connecticut. He received his Ph. D. in Electrical Engineering from Princeton University in June 2004. His research areas are in computer architecture, cryptography, sensor networks, and high performance, secure computer systems.

Hai Yan is a Ph. D. student at the University of Connecticut. His research interests include computer security for sensor networks and computer architecture. Hai has an M. S. in computer science from Wuhan University, Wuhan, China.