Fast Implementation of Elliptic Curve Arithmetic in GF(p^n)

Chae Hoon Lim and Hyo Sun Hwang

Information and Communications Research Center, Future Systems, Inc.
372-2, Yang Jae-Dong, Seo Cho-Gu, Seoul, 137-130, KOREA
{chlim,hyosun}@future.co.kr

Abstract. Elliptic curve cryptosystems have attracted much attention in recent years and one of major interests in ECC is to develop fast algorithms for field/elliptic curve arithmetic. In this paper we present various improvement techniques for field arithmetic in GF(p^n)(p a prime), in particular, fast field multiplication and inversion algorithms, and provide our implementation results on Pentium II and Alpha 21164 microprocessors.

1 Introduction

Elliptic curve cryptosystems, first introduced by Koblitz [10] and Miller [19], have been investigated by many other researchers in recent years. In particular, much research has been conducted on fast algorithms and implementation techniques of elliptic curve arithmetic over various finite fields [12,22,24,23,8,7,2,9].

Since elliptic curve groups can provide a higher level of security with smaller key sizes, there is increasing interest also in the industry and thus a lot of active standardization processes are going on, for example, IEEE P1363 [26], ISO/IEC CD 14883-3 and DIS 11770-3, ANSI X.9.62/9.63 [27,28], etc. Thus we can expect that elliptic curve cryptosystems will be widely used for many security applications in the near future. In this regard, it is also expected that there will be a strong demand on efficient algorithms for fast implementation of elliptic curve cryptosystems.

An elliptic curve over GF(2^n) is best suited for hardware implementations, but in software an elliptic curve over GF(p^n) is more attractive since better performances can be achieved with a suitable choice of parameters [2]. This paper is devoted to devising and implementing efficient methods for speeding up field arithmetic in GF(p^n). In particular, we present a fast field inversion algorithm in GF(p^n), which only requires one subfield inversion and thus runs significantly (more than 2 to 4 times) faster than the extended Euclidean algorithm for most sizes of p interested to us (i.e., $|p| \approx 32$ or 64). Using the speedup techniques presented in this paper, we have implemented elliptic curve arithmetic with various choices of field parameters for GF(p^n). Our implementation shows that scalar multiplication can be performed more than 5 times faster than modular exponentiation for parameters of a comparable security level.

This paper is organized as follows. Section 2 briefly summarizes elliptic curve arithmetic in GF(p^n) and Section 3 describes various algorithms and speed-up techniques for field multiplication and inversion. We then present our implementation results in Section 5 and conclude in Section 6.

2 Elliptic Curve Arithmetic in GF(p^n)

2.1 Elliptic Addition/Doubling in Affine Coordinates

A non-supersingular elliptic curve defined over a finite field GF(p^n) is a set of points (x, y) given by the cubic equation

$$y^2 = x^3 + ax + b \ (a, b \in \mathrm{GF}(p^n),\ 4a^2 + 27b^3 \neq 0),$$

together with 'point at infinity' as an identity element. Addition formulas in this affine coordinate system are defined as:

– Addition: $(x_2, y_2) = (x_0, y_0) + (x_1, y_1)$,

$$\begin{array}{l} x_2 = \lambda^2 - (x_0 + x_1) \\ y_2 = \lambda(x_0 - x_2) - y_0 \end{array}, \text{ where } \lambda = \frac{y_1 - y_0}{x_1 - x_0}$$

– Doubling: $(x_2, y_2) = 2(x_0, y_0)$

$$\begin{array}{l} x_2 = \lambda^2 - 2x_0 \\ y_2 = \lambda(x_0 - x_2) - y_0 \end{array}, \text{ where } \lambda = \frac{3x_0^2 + a}{2y_0}$$

2.2 Elliptic Addition/Doubling in Projective Coordinates

A big disadvantage of using affine representation of elliptic curve points is that addition/doubling requires a very expensive field inversion. There is another way to represent elliptic curve points, the so-called (weighted) projective representation, which eliminates the expensive field inversion at the cost of more field multiplications. The addition/doubling formulas described here are similar to the ones in the IEEE P1363 Draft [26].

We will use the following transformation for coordinate conversions:

$$x = \frac{X}{Z^2}, \quad y = \frac{Y}{2Z^3}.$$

So, the affine coordinates (x, y) should be transformed into the corresponding projective coordinates $(X, Y, Z) = (x, 2y, 1)$. The factor 2 in y is included here to eliminate the modular division by 2 appearing in the addition formula when using $y = \frac{Y}{Z^3}$ (see A.10.5 in [26]). This also reduces the number of field additions/subtractions required in the doubling formula. Note that the addition/subtraction time in GF(p^n) is not negligible (about 15% of field multiplication on P6 and Alpha; see Sect.4).

– Addition: $(X_2, Y_2, Z_2) = (X_0, Y_0, Z_0) + (X_1, Y_1, Z_1)$

$$\begin{aligned} A &= X_0Z_1^2 + X_1Z_0^2, \quad B = X_0Z_1^2 - X_1Z_0^2, \\ C &= Y_0Z_1^3 + Y_1Z_0^3, \quad D = Y_0Z_1^3 - Y_1Z_0^3, \quad E = 2B, \\ Z_2 &= EZ_0Z_1, \quad X_2 = D^2 - AE^2, \quad Y_2 = D(AE^2 - 2X_2) - E^2BC. \end{aligned}$$

Special case of $Z_1 = 1$:

$$\begin{aligned} A &= X_0 + X_1Z_0^2, \quad B = X_0 - X_1Z_0^2, \\ C &= Y_0 + Y_1Z_0^3, \quad D = Y_0 - Y_1Z_0^3, \quad E = 2B, \\ Z_2 &= EZ_0, \quad X_2 = D^2 - AE^2, \quad Y_2 = D(AE^2 - 2X_2) - E^2BC. \end{aligned}$$

– Doubling: $(X_2, Y_2, Z_2) = 2(X_0, Y_0, Z_0)$

$$\begin{aligned} A &= 3X_0^2 + aZ_0^4, \quad B = 2X_0Y_0^2, \quad C = Y_0^4, \\ Z_2 &= Y_0Z_0, \quad X_2 = A^2 - B, \quad Y_2 = A(B - 2X_2) - C. \end{aligned}$$

The above formulas show that elliptic addition requires 12 multiplications, 4 squarings and 9 additions in GF(p^n) (8 multiplications, 3 squarings and 9 additions if $Z_1 = 1$), while elliptic doubling requires 4 multiplications, 6 squarings and 7 additions. Note however that k repeated doublings can save 2 squarings in $k - 1$ doublings by using one more chaining variable for aZ_0^4 (i.e., compute $W = aZ_0^4$ at the first doubling and then update W by $W \leftarrow CW$ in each iteration except for the final doubling) [7]. Also note that if $a = -3$, we can compute A in doubling as $A = 3(X_0 + Z_0^2)(X_0 - Z_0^2)$.

2.3 Performance Comparison

Table 1 summarizes the number of field operations required for elliptic addition and doubling in each coordinate system, where D_e and A_e respectively denote elliptic doubling and addition, and I, M, S and A respectively denote field operations of inversion, multiplication, squaring and addition. The number of squarings in elliptic addition can be reduced by 2 with the special choice of $a = -3$. So, for better performances we used $a = -3$ in all our implementations. Note that the fixed value for a does not much restrict the choice of elliptic curves, since the proportion of elliptic curves that can be rescaled to have $a = -3$ is approximately 1/2 or 1/4, depending on the residue of p mod 4, for GF(p^n) (see Appendix A in [26]).

Which representation of elliptic curve points gives rise to a better performance can be determined by the speed ratio of field inversion to field multiplication (I/M). Obviously, arithmetic in projective coordinates with affine representation of precomputed points (i.e., $Z_1 = 1$) almost always outperforms arithmetic using pure projective representation (i.e., $Z_1 \neq 1$). The ratio I/M at the break-even point between performances with affine and projective ($Z_1 = 1$) representations can be shown to lie between 3.6 and 7.6, assuming that $a = -3$ and $1S = 0.8M$ [15]. For example, we have $I/M = 4.4$ for the signed window algorithm for scalar multiplication (window size=4), so it is always preferable to use projective coordinates with $Z_1 = 1$ if $I > 4.4M$.

a	EC oper.	Affine	Projective ($Z_1 \neq 1$)	Projective ($Z_1 = 1$)
random	D_e	$1I + 2M + 2S + 7A$	$4M + 4S + 7A$	$4M + 4S + 7A$
	A_e	$1I + 2M + S + 6A$	$12M + 4S + 9A$	$8M + 5S + 9A$
$a = -3$	D_e	$1I + 2M + 2S + 6A$	$4M + 4S + 9A$	$4M + 4S + 9A$
	A_e	$1I + 2M + S + 6A$	$12M + 4S + 9A$	$8M + 3S + 9A$

Table 1. The number of field operations for elliptic curve doubling and addition

2.4 Elliptic Scalar Multiplication

Elliptic scalar multiplication is to compute kP for a given point P on an elliptic curve and a random integer k (let $|k| = l$). The best known method for general elliptic scalar multiplication is the sliding window algorithm using addition/subtraction chains [12,6,23]. For this, we need to precompute and store odd multiples of P, $P_i = iP$ for odd i's less than 2^w, which requires $2^{w-1} - 1$ elliptic additions and one elliptic doubling. Note that the precomputation should be done in affine coordinates for better performances, so we may use Montgomery's simultaneous inversion technique [5, Algorithm 10.3.4] to reduce the number of field inversions at the cost of more field multiplications (see also [7]). In this case, the cost for precomputation is given by $wI+(5\cdot 2^{w-1}+2w-10)M+(2^{w-1}+2w-3)S$. Since the average interval between two consecutive windows is equal to 2 for an optimal signed encoding (e.g., see [6]), the total computational cost on average for computing kP using the signed window algorithm with window size w is approximately given by

$$\begin{aligned} T_W &= \big(wI + (5\cdot 2^{w-1} + 2w - 10)M + (2^{w-1} + 2w - 3)S\big) \\ &\quad + \Big((l - w + 1)D_e + (\frac{l}{w+2} - 1)A_e\Big) \\ &= wI + \left(\frac{8l}{w+2} + 4l + 5\cdot 2^{w-1} - 2w - 14\right) M \\ &\quad + \left(\frac{3l}{w+2} + 4l + 2^{w-1} - 2w - 2\right) S, \end{aligned} \tag{1}$$

where we assumed to use projective representation with $Z_1 = 1$.

A different approach for computing kP was introduced by Montgomery, based on the observation that the x-coordinate of the sum of two points can be computed only using the x-coordinates of the two points if their difference is known [20] (see also [1,17]). Let $k = \sum_{i=0}^{l-1} k_i 2^i$ be the binary representation of the multiplier k (assume that $k_{l-1} = 1$). Montgomery's method successively updates a pair of points (x-coordinates only) S_i, T_i ($S_0 = P, T_0 = 2P$), while maintaining the invariant relationship $T_i - S_i = P$, by computing, for each k_i, $(S_{i+1} = 2S_i, T_{i+1} = S_i + T_i)$ if $k_i = 0$ and $(S_{i+1} = S_i + T_i, T_{i+1} = 2T_i)$ if $k_i = 1$ $(i = l-2, \cdots, 1, 0)$. Thus, we need one elliptic addition and one elliptic doubling for each bit of k, regardless of the Hamming weight of k. Each of elliptic addition and doubling can be done in 3 multiplications and 2 squarings in projective coordinates using Montgomery's alternative parameterization of elliptic curves

($By^2 = x^3 + Ax^2 + x$ for some A and B). So, the total cost for computing the x-coordinate of kP is given by

$$T_M = (l-1)(A_e + D_e) = (l-1)(6M + 4S). \tag{2}$$

Let us compare the performance of the signed window algorithm and Montgomery's method. With optimal window size $w = 4$ for most interesting values of l, we have $T_W = 4I + (5.33l + 18)M + (4.5l - 2)S = 4I + (8.93l + 16.4)M$ from equation (1), assuming that $1S = 0.8M$. Since $T_M = 9.2(l-1)M$ from equation (2), we can see that the signed window algorithm is asymptotically faster than Montgomery's method. For a typical value of $l = 160$, we have $T_W = 4I + 1445M$ and $T_M = 1463M$, so Montgomery's method may be a little bit faster. However, Montgomery's method is not a general algorithm for elliptic scalar multiplication in GF(p^n), since it can't compute the y-coordinate of kP (note however that this is not the case in GF(2^n); see [17]). Of course, this may not be a problem in many applications, since most elliptic curve variants of key exchange and digital signature schemes only make use of x coordinates (e.g., see [26,27,28]). We thus use the signed window algorithm for scalar multiplication in this paper. Finally, it is worth noting that there are several advantages in Montgomery's method: it does not require precomputation, which may be desired for implementations in a limited computing environment (e.g., an implementation on low-cost smart cards), addition and doubling can be performed in parallel on multi-processor architectures or in hardware implementation, and the execution time does not depend on the Hamming weight of multipliers, which helps to prevent timing attacks.

On the other hand, we may use an elliptic curve defined over GF(p) as an elliptic curve over GF(p^n) (let us call such a curve as a subfield curve). A big advantage of using such a subfield curve is that elliptic scalar multiplication on a subfield curve can be substantially speeded up using Frobenius expansion [9,15] (see also [11,18,23,21,4]). Though subfield curves allow much faster implementations and easy parameter generation, we should be careful for security concerns related to the special structure (e.g., see [14,25]). We do not consider such a special curve in this paper, but for comparison we provide the implementation result from [15] in Sect.4.

3 Speeding up Field Arithmetic

Optimization of field arithmetic is much more critical to the overall performance of elliptic scalar multiplication than optimization in group operations. This section describes various algorithms and techniques for speeding up field multiplication and inversion in GF(p^n).

3.1 Field Construction

The performance of field arithmetic in GF(p^n) ($n > 1$) heavily depends on the choice of parameters for field extension (a prime p and an irreducible polynomial

$f(x)$). For fast field arithmetic in GF(p^n), Bailey and Paar [2] proposed to choose the parameters p, n and $f(x)$ such that $p = 2^m - c$ with small c and m around a target computer word size and $f(x) = x^n - \omega$ with small ω (called an Optimal Extension Field (OEF)).

For further optimization of field arithmetic, we placed another restriction on the size of p: choose p, whenever possible, so that multiplication results of two $|p|$-bit numbers can be accumulated as many as possible without overflow in computer's word boundary. Then, with such a p one can reduce the number of reductions mod p required for field multiplication from n^2 to n. This will result in a substantial improvement in the overall performance, since modular reduction is still quite expensive even with the special choice of p in typical microprocessors, such as P6 and Alpha.

The field parameters selected for use in our implementations are summarized in Table 2. Of course, there are other possible choices worth considering, such as ($p = 2^{31} - 1, f(x) = x^6 - 5$), ($p = 2^{61} - 1, f(x) = x^3 - 37$), ($p = 2^{84} - 35$ or $2^{85} - 19, f(x) = x^2 - 2$) and $p = 2^{166} - 5$. The three field parameters with degree of n^* ($n = 7, 11, 13$) were included for use in building subfield curves. The figures of the 'order' column in Table 2 denote the largest possible prime orders in $E/\mathrm{GF}(p^n)$.

n	order (bits)	$p = 2^m - c$	$f(x)$
13*	168	$2^{14} - 3$	$x^{13} - 2$
12	168	$2^{14} - 3$	$x^{12} - 2$
11*	160	$2^{16} - 437$	$x^{11} - 2$
10	160	$2^{16} - 165$	$x^{10} - 2$
7*	168	$2^{28} - 57$	$x^7 - 2$
6	168	$2^{28} - 165$	$x^6 - 2$
5	160	$2^{32} - 5$	$x^5 - 2$
3	171	$2^{57} - 13$	$x^3 - 2$
2	178	$2^{89} - 1$	$x^2 - 3$
1	160	$p = 2^{160} - 2933$	

Table 2. Selected parameters for extension field GF(p^n)

3.2 Field Multiplication and Squaring

Let $A(x)$ and $B(x)$ be polynomials of degree $n - 1$ over GF(p), i.e.,

$$A(x) = \sum_{i=0}^{n-1} A_i x^i, \quad B(x) = \sum_{i=0}^{n-1} B_i x^i,$$

where $A_i, B_i \in \mathrm{GF}(p)$. Field multiplication in GF(p^n) is to compute $C(x) = A(x)B(x) \bmod f(x)$.

First, consider methods for speeding up polynomial multiplication. There exists a well-known divide-and-conquer technique, called the Karatsuba-Ofman algorithm, to reduce the number of subfield multiplications required for polynomial multiplication [13, Sect.4.3.3]. We can use Karatsuba-Ofman's algorithm in two directions. For example, for $n = 6$, let $A(x) = A_h(x)x^3 + A_l(x), B(x) = B_h(x)x^3 + B_l(x)$, where

$$\begin{aligned} A_h(x) = A_5x^2 + A_4x + A_3, \quad A_l(x) &= A_2x^2 + A_1x + A_0, \\ B_h(x) = B_5x^2 + B_4x + B_3, \quad B_l(x) &= B_2x^2 + B_1x + B_0. \end{aligned}$$

Then we can compute $C(x) = A(x)B(x)$ as $C(x) = D_h(x)x^6 + D_m(x)x^3 + D_l(x)$, where

$$\begin{aligned} D_h(x) &= A_h(x)B_h(x), \quad D_l(x) = A_l(x)B_l(x), \\ D_m(x) &= (A_h(x) + A_l(x))(B_h(x) + B_l(x)) - (D_h(x) + D_l(x)). \end{aligned}$$

This high-level application of Karatsuba-Ofman's algorithm reduces the number of subfield multiplications from $n^2 = 36$ to $\frac{3}{4}n^2 = 27$ at the cost of more subfield additions. Our implementation shows that this technique is only effective on Alpha 21164 for n up to 7.

We may apply Karatsuba-Ofman's algorithm once again to polynomial multiplication of degree 3. However, for small n, it is better to use Karatsuba-Ofman's algorithm at the lowest word level. For example, for $n = 3$, $C(x) = (A_0 + A_1x + A_2x^2)(B_0 + B_1x + B_2x^2)$ can be computed as

$$C(x) = D_0 + (D_3 - D_0 - D_1)x + (D_4 + D_1 - D_2 - D_0)x^2 + (D_5 - D_2 - D_1)x^3 + D_2x^4,$$

where

$$\begin{aligned} D_0 = A_0B_0, \quad D_1 &= A_1B_1, \quad D_2 = A_2B_2, \quad D_3 = (A_0 + A_1)(B_0 + B_1), \\ D_4 = (A_0 + A_2)(B_0 + B_2), \quad D_5 &= (A_1 + A_2)(B_1 + B_2). \end{aligned}$$

We can thus reduce the number of subfield multiplications from 9 to 6 (reduction rate of 2/3). Of course, this low-level Karatsuba-Ofman's algorithm can be applied to polynomial multiplication of any degree n. In this case, it is easy to see that the number of subfield multiplications required can be reduced to $\frac{n(n+1)}{2}$. However, as n becomes larger, the increase in the addition (and memory access) complexity may be larger than the reduction in the multiplication complexity. So, the effectiveness of this method varies from architecture to architecture, depending on the relative speed of basic operations involved and on the number of general-purpose registers available. For example, our implementation shows that this technique is only useful for $n = 2$ or 3 on Pentium II and for n up to 7 on Alpha 21164.

On the other hand, we can reduce the number of reductions mod p by accumulating individual product terms, A_iB_j's, as many as possible, and then reducing the accumulated sum mod p only once. To see this, let us express

$C(x) = A(x)B(x) \bmod f(x)$, using the identity $x^n = \omega \bmod f(x)$, as

$$\begin{aligned} C(x) &= \sum_{i=0}^{n-1} A_i x^i \cdot \sum_{j=0}^{n-1} B_j x^j \bmod f(x) \\ &= \sum_{i+j=k<n} A_i B_j x^k + \omega \sum_{i+j=k\geq n} A_i B_j x^{k-n} \\ &= \sum_{k=0}^{n-1} \left(\sum_{i=0}^{k} A_i B_{k-i} + \omega \sum_{i=k+1}^{n-1} A_i B_{n+k-i} \right) x^k, \end{aligned}$$

so the coefficient C_k can be computed by

$$C_k = \left(\sum_{i=0}^{k} A_i B_{k-i} + \omega \sum_{i=k+1}^{n-1} A_i B_{n+k-i} \right) \bmod p. \tag{3}$$

Therefore, with this accumulation-and-then-reduction technique, we can reduce the number of reductions mod p from n^2 to n. We still need n^2 multiplications of $|p|$-bit integers, assuming ω is very small (typically 2 or 3), so that multiplication by ω can be done by a few additions (this is possible in all our field constructions, as can be seen in Table 2). However, our experiments on P6 and Alpha show that modular reduction is more expensive than multiplication in most interesting fields (of course, except for large p). So, the above accumulation-and-then-reduction technique actually contributes to the overall performance more than any other optimization technique.

Since C_k is at most $2m + \lceil \log_2(\omega(n-1)+1) \rceil$ bits long, we can do modular reduction using at most 2 multiplications by c, as long as $\lceil \log_2(\omega(n-1)+1) \rceil + 2|c| \leq m$, which is the case for most choices of p and $f(x)$. Equation (3) suggests that it is preferable to choose p such that the largest partial product sum, C_0, do not produce an additional carry in word boundary of a target computer, as in the parameters given in Table 2. For example, we chose $p = 2^{28} - 165$ and $f(x) = x^6 - 2$ for GF(p^6), instead of $p = 2^{31} - 1$ and $f(x) = x^6 - 5$.

Finally, note that field squaring in GF(p^n) only requires $\frac{n(n+1)}{2}$ multiplications of $|p|$-bit numbers, while the number of modular reductions remains the same. Thus, though all optimization techniques described above can be applied equally well to field squaring, depending on p field squaring may not be improved as much as expected, compared to field multiplication.

3.3 Field Inversion

Field inversion in GF(p^n) $(n > 1)$ corresponds to computing a multiplicative inverse of a polynomial modulo an irreducible polynomial $f(x)$ of degree n. Inversion in GF(2^n) can be best performed by the *almost inverse algorithm (AIA)* [22]. However, the AIA is not effective at all for polynomial inversion in GF(p^n). Rather, the extended Euclidean algorithm runs a little bit faster. In this section we present fast algorithms for polynomial inversion in GF(p^n) and compare their computational complexity with other known methods in [9,3].

Variant of Extended Euclidean Algorithm We start with a polynomial version of Extended Euclidean algorithm shown in Table 3 and improves it in step-by-step. Let deg(A) be the degree of $A(x)$.

```
Algorithm IE
Input: $A(x)$ and $f(x)$ such that $\deg(A) < n$ and $\deg(f) = n$.
Output: $B(x)$ such that $A(x)B(x) = 1 \bmod f(x)$.
1. set $B \leftarrow 0,\ C \leftarrow 1,\ F \leftarrow f(x)$ and $G \leftarrow A(x)$.
2. repeat the following steps while $\deg(F) \neq 0$:
   2-1. if $\deg(F) < \deg(G)$, then exchange $F, B$ with $G, C$, respectively.
   2-2. update $F$ and $B$ as follows (let $j = \deg(F) - \deg(G)$):
            $\alpha \leftarrow F_{\deg(F)} G^{-1}_{\deg(G)},\quad F \leftarrow F - \alpha x^j G,\quad B \leftarrow B - \alpha x^j C.$
3. return $B \leftarrow F_0^{-1} B$.
```

Table 3. Polynomial version of Extended Euclidean algorithm

Algorithm IE reduces the degree of the larger out of $F(x)$ and $G(x)$ by at least one in each iteration of step 2. Thus, we need at most $2n - 2$ iterations of step 2 in total. The most time-consuming operation in Algorithm IE is subfield inversion for most preferable choices of p, which is much slower than even a field multiplication on Pentium II. So we first try to reduce the number of subfield inversions required by using some parallelism in step 2 of Algorithm IE. The idea is to reduce the degree of $F(x)$ or $G(x)$ by two or more at a time as shown in Table 4.

```
Algorithm IP
Input: $A(x)$ and $f(x)$ such that $\deg(A) < n$ and $\deg(f) = n$.
Output: $B(x)$ such that $A(x)B(x) = 1 \bmod f(x)$.
1. set $B \leftarrow 0,\ C \leftarrow 1,\ F \leftarrow f(x)$ and $G \leftarrow A(x)$.
2. repeat the following steps while $deg(F) \neq 0$:
   2-1. if $\deg(F) < \deg(G)$, then exchange $F, B$ with $G, C$, respectively.
   2-2. update $F$ and $B$ as follows ($j = \deg(F) - \deg(G)$):
            $\alpha \leftarrow F_{\deg(F)} G^{-1}_{\deg(G)},\quad \beta \leftarrow (F_{\deg(F)-1} - \alpha G_{\deg(G)-1}) G^{-1}_{\deg(G)},$
            $F \leftarrow F - (\alpha x^j + \beta x^{j-1}) G,\quad B \leftarrow B - (\alpha x^j + \beta x^{j-1}) C.$
   2-3. if $\deg(F) = \deg(G)$, then execute the following:
            $\alpha \leftarrow F_{\deg(F)} G^{-1}_{\deg(G)},\quad F \leftarrow F - \alpha G,\quad B \leftarrow B - \alpha C.$
3. return $B \leftarrow F_0^{-1} B$.
```

Table 4. Parallel version of Algorithm IE

In Algorithm IP, each iteration of step 2 reduces the degree of $F(x)$ by at least two by subtracting a suitable multiple of $G(x)$. Thus, the numbers of subfield inversions and modular reductions are now reduced by half, compared to Algorithm IE, though we need the same number of $|p|$-bit multiplications. This improvement is crucial to the overall performance in actual implementations, since both subfield inversion and modular reduction are more expensive than multiplication for most sizes of p interested to us (i.e., $|p|$ around 32 or 64). Note that we don't have to compute the first two highest coefficients of $F(x)$ in step 2-2, since they must be zero. Also note that if $|\deg(F) - \deg(G)|$ is equal to 1 at the begining of step 2, this difference is maintained throughout the iteration with probability of $1/p$. Thus, step 2-3 will not be executed in most cases.

Field Inversion Using Multiplication Since subfield inversion becomes more and more expensive as the size of p increases, a natural question to ask is how to minimize the number of subfield inversions in a field inversion algorithm. Fortunately, we are able to modify Algorithm IE/IP into algorithms requiring only one subfield inversion at the final stage. First note that Algorithm IE maintains the following relationships throughout its internal processing:

$$A(x)B(x) + U(x)f(x) = F(x), \quad A(x)C(x) + V(x)f(x) = G(x),$$

for some polynomials $U(x)$ and $V(x)$ (not interested to us). Therefore, we can see that these relations still hold even if we multiply both $F(x)$ and $B(x)$ (both $G(x)$ and $C(x)$, respectively) by the same constant. This observation shows that the following field inversion algorithm actually works, where we only describe the variant of Algorithm IP (A referee kindly pointed out that Algorithm IM can be constructed from algorithms in [13, Sect.4.6.1]):

```
Algorithm IM
Input: $A(x)$ and $f(x)$ such that $\deg(A) < n$ and $\deg(f) = n$.
Output: $B(x)$ such that $A(x)B(x) = 1 \bmod f(x)$.

1. set $B \leftarrow 0,\ C \leftarrow 1,\ F \leftarrow f(x)$ and $G \leftarrow A(x)$.
2. repeat the following steps while $\deg(F) \neq 0$:
   2-1. if $\deg(F) < \deg(G)$, then exchange $F, B$ with $G, C$, respectively.
   2-2. update $F$ and $B$ as follows (let $j = \deg(F) - \deg(G)$):
            $\alpha \leftarrow G^2_{\deg(G)},\quad \beta \leftarrow F_{\deg(F)} G_{\deg(G)}$,
            $\gamma \leftarrow G_{\deg(G)} F_{\deg(F)-1} - F_{\deg(F)} G_{\deg(G)-1}$,
            $F \leftarrow \alpha F - (\beta x^j + \gamma x^{j-1})G,\quad B \leftarrow \alpha B - (\beta x^j + \gamma x^{j-1})C$.
   2-3. if $\deg(F) = \deg(G)$, then execute the following:
            $F \leftarrow G_{\deg(F)} F - F_{\deg(F)} G,\quad B \leftarrow G_{\deg(F)} B - F_{\deg(F)} C$.
3. return $B \leftarrow F_0^{-1} B$.
```

Table 5. Field inversion using multiplication

Algorithm IM requires just one subfield inversion at the final step. Comparing steps 2-2 in Algorithm IM and Algorithm IP, one can see that one subfield inversion in Algorithm IP was replaced with one multiplication by constant in both $F(x)$ and $B(x)$ in Algorithm IM (or equivalently about n $|p|$-bit multiplications). Consequently, Algorithm IM will be more efficient than Algorithm IP if subfield inversion is more expensive than n $|p|$-bit multiplications.

To see how $F(x)$ is updated in step 2, let us suppose that the degrees of the current $F(x)$ and $G(x)$ are d and $d-1$, respectively. Then, the coefficient update in $F(x)$ and $B(x)$ in step 2 can be described by

$$F_i \leftarrow \begin{cases} \alpha F_i - \gamma G_i \bmod p & \text{for } i = 0, \\ \alpha F_i - (\beta G_{i-1} + \gamma G_i) \bmod p & \text{for } 1 \leq i \leq d-2, \end{cases}$$

$$B_i \leftarrow \begin{cases} \alpha B_i - (\beta C_{i-1} + \gamma C_i) \bmod p & \text{for } 0 \leq i \leq n-d-1, \\ \alpha B_i - \gamma C_i \bmod p & \text{for } i = n-d. \end{cases}$$

If the degree of $f(x)$ is 2 or 3, then we can derive simple inversion formulas from Algorithm IM, which will be more efficient than direct execution of Algorithm IM since we can do more intelligent manual optimization.

- GF(p^2): Let $A(x) = A_1x + A_0$ ($A_i \in$ GF(p)) and $f(x) = x^2 - \omega$. Then, $B(x) = A(x)^{-1} \bmod f(x)$ can be computed by

$$B(x) = F_0^{-1}(A_1x - A_0), \text{ where } F_0 = \omega A_1^2 - A_0^2.$$

- GF(p^3): Let $A(x) = A_2x^2 + A_1x + A_0$ ($A_i \in$ GF(p)) and $f(x) = x^3 - \omega$. Then, $B(x) = A(x)^{-1} \bmod f(x)$ can be computed by

$$B(x) = A_2F_0^{-1}(T_2x^2 + T_1x + T_0), \text{ where}$$

$$T_0 = A_0^2 - \omega A_1A_2, \quad T_1 = \omega A_2^2 - A_0A_1, \quad T_2 = A_1^2 - A_0A_2, \quad F_0 = T_1^2 - T_0T_2.$$

Comparison of Inversion Algorithms To see the relative performance of the three inversion algorithms describe above, we calculated the number of basic operations required in each algorithm. The result is summarized in Table 6. Our experiments on P6 and Alpha microprocessors show that subfield inversion (not using table lookups) is the most expensive in general, and reduction mod p is more expensive than multiplication of $|p|$-bit numbers. So, we separately counted the number of $|p|$-bit multiplications, reductions mod p and inversions mod p. Note that if we do not differentiate modular reduction from multiplication, the number of multiplications in the table corresponds to the number of subfield multiplications.

In Table 6, we also included the computational complexity of Bailey and Paar's inversion algorithm in GF(p^n). This algorithm computes $A(x)^{-1}$ mod $f(x)$ as

$$A(x)^{-1} = (A(x)^r)^{-1}A(x)^{r-1} \bmod f(x), \quad \text{where } r = \frac{p^n - 1}{p - 1} = 1 + p + p^2 + \cdots + p^{n-1}.$$

Since $A(x)^r = A(x)A(x)^{r-1}$ is always an element in GF(p), this algorithm also requires only one subfield inversion. Furthermore, due to the special form of

algorithm	#multiplications	#reductions	#inversions
IE	$2n^2+n-4$	$2n^2+n-4$	$2n-1$
IP	$2n^2+n-4$	n^2-n-2	$n+1$
IM $(f(x)=x^n-\omega)$	$3n^2-5$ $(3n^2-n-7)$	n^2+4n-6 (n^2+4n-8)	1 (1)
BP [3] $f(x)=x^n-\omega$	$t(n)(n^2+2n-2)+3n-1$	$t(n)(3n-2)+2n$	1
	$t(n)=\lfloor\log_2(n-1)\rfloor+H_w(n-1)-1$		

Table 6. Complexity for computing $A(x)^{-1}$ mod $f(x)$ $(\deg(f(x))=n$, $\deg(A(x))=n-1)$

$r-1$, $A(x)^{r-1}$ can be efficiently computed using $x^n=\omega \bmod f(x)$ and some basic properties of finite fields (for details, see [3]). The complexity of this algorithm is given by

$$(\lfloor\log_2(n-1)\rfloor+H_w(n-1)-1)(n^2+2n-2)+3n-1,$$

where $H_w(x)$ denotes the Hamming weight of x. In Table 6, we assumed that general polynomial multiplication of degree n can be done in n^2 multiplications and n modular reductions using the accumulation-then-reduction technique. Note that this algorithm is only useful for a binomial $f(x)$ (thus not general) and that its complexity is higher than Algorithm IM for any n.

There is another field inversion method using matrix inversion proposed in [9]. The explicit formula for $n=3$ given in [9] is almost identical to that of Algorithm IM. However, this algorithm has a computational complexity of $O(n^3)$ and is not general either. Algorithm IM seems to be the best algorithm for field inversion in GF(p^n). It is also general (works equally well with any $f(x)$), systematic and easy to implement (independent of a specific form of $f(x)$).

4 Implementation and Discussion

We have implemented various field and elliptic curve arithmetic using the algorithms and techniques presented in Section 3 on two typical microprocessors: Pentium II/266MHz (32-bit μP; Windows 98, MSVC 5.0 with in-line assembly) and Alpha 21164/533MHz (64-bit μP; Linux, gcc 2.95 with in-line assembly).

4.1 Timings for Field Arithmetic

To see the relative speed of three inversion algorithms described in Sect.3 in actual implementations, we measured their speed on Pentium II and Alpha 21164, as shown in Tables 7 and 8. The tables show that Algorithm IP runs about $25 \sim 45\%$ faster than Algorithm IE and that Algorithm IM runs about $30 \sim 60\%$ faster than Algorithm IP for $3 \le n \le 7$. We used a table lookup method for subfield inversion in GF(p^n) for $n > 7$, so the advantage of Algorithm IM disappears

Alg.	IE	IP	IM	IP/IE	IM/IE	IM/IP
GF(p^{13})	28.71	15.82	**15.54**	0.55	0.54	0.98
GF(p^{12})	24.96	14.28	**13.79**	0.57	0.55	0.97
GF(p^{11})	32.44	**20.25**	22.44	0.62	0.69	1.12
GF(p^{10})	27.64	**16.79**	19.18	0.61	0.69	1.14
GF(p^{7})	35.90	21.79	**12.13**	0.61	0.34	0.56
GF(p^{6})	29.16	17.54	**9.48**	0.60	0.33	0.54
GF(p^{5})	25.10	15.02	**7.20**	0.60	0.29	0.48
GF(p^{3})	38.64	26.14	**9.69**	0.68	0.25	0.37

Table 7. Speed of three field inversion algorithms on Pentium II/266MHz (in μsec)

Alg.	IE	IP	IM	IP/IE	IM/IE	IM/IP
GF(p^{13})	20.05	**12.92**	12.95	0.64	0.65	1.00
GF(p^{12})	17.28	11.29	**11.13**	0.65	0.64	0.99
GF(p^{11})	21.05	14.04	**13.10**	0.67	0.62	0.93
GF(p^{10})	14.47	10.76	**10.06**	0.74	0.70	0.93
GF(p^{7})	15.98	9.78	**6.18**	0.61	0.39	0.63
GF(p^{6})	13.14	8.30	**5.16**	0.63	0.39	0.62
GF(p^{5})	11.08	7.28	**5.12**	0.66	0.46	0.70
GF(p^{3})	10.23	7.29	**2.82**	0.71	0.28	0.39

Table 8. Speed of three field inversion algorithms on Alpha 21164/533MHz (in μsec)

in these cases. For inversion in GF(p), we used a combination of the binary and integer extended Euclidean algorithms for a better performance.

Tables 9 and 10 show the speed of field arithmetic for various fields defined in Table 2. For comparison, we also included the timings for GF(2^n). The ratio A/M ranges from 0.1 to 0.2 in GF(p^n), so the addition time in GF(p^n) is not negligible. The column S/M shows that $1S \approx 0.8M$ on Pentium II. But on Alpha 21164 we have $1S \approx 0.9M$ for $n \leq 7$, which would be due to better optimization in field multiplication by the Karatsuba-Ofman algorithm. The same argument also explains the lower ratio of S/M ($1S \approx 0.6M$) for $n \geq 10$, since in this case the number of subfield multiplications required for squaring is $n(n+1)/2$, compared to n^2 for multiplication (note that the Karatsuba-Ofman algorithm was applied (effective) only up to $n = 7$ on Alpha). The I/M ratio ranges from 7 to 9 for $n > 3$ on both microprocessors (we used best field inversion algorithms for each n; see Tables 7 and 8). Note that for small n (e.g., $n < 7$), subfield inversion takes much more time than field multiplication (compare SF_Inv column with F_Mul column), which explains why Algorithm IM runs much faster than Algorithm IE or IP in these cases (in particular, as p increases). Field inversion in GF(p) is extremely expensive compared to multiplication (e.g., about 30 to 40 times slower than modular multiplication).

Field	SF_Inv	F_Add	F_Sqr	F_Mul	F_Inv	A/M	S/M	I/M
GF(2^{162})		0.119	0.558	3.917	54.92	0.03	0.14	14.0
GF(p^{13})	0.072	0.343	1.517	2.042	15.79	0.17	0.74	7.73
GF(p^{12})	0.071	0.326	1.364	1.816	13.94	0.18	0.75	7.68
GF(p^{11})	0.072	0.310	1.966	2.530	20.54	0.12	0.78	8.12
GF(p^{10})	0.073	0.290	1.647	2.100	17.14	0.14	0.78	8.16
GF(p^7)	1.718	0.158	1.131	1.426	12.25	0.11	0.79	8.59
GF(p^6)	1.700	0.142	0.917	1.119	9.595	0.13	0.82	8.57
GF(p^5)	2.075	0.174	0.787	0.956	7.311	0.18	0.82	7.65
GF(p^3)	6.255	0.235	1.042	1.264	9.723	0.19	0.82	7.69
GF(p^2)	17.62	0.157	0.857	1.004	19.83	0.16	0.85	19.8
GF(p)		0.151	0.885	1.012	43.25	0.15	0.87	42.7

Table 9. Timings (in μsec) for field operations on Pentium II 266MHz

Field	SF_Inv	F_Add	F_Sqr	F_Mul	F_Inv	A/M	S/M	I/M
GF(2^{162})		0.058	0.198	1.093	11.73	0.05	0.18	10.7
GF(p^{13})	0.081	0.217	0.956	1.710	12.96	0.13	0.56	7.58
GF(p^{12})	0.089	0.220	0.854	1.460	11.18	0.15	0.59	7.66
GF(p^{11})	0.097	0.185	0.967	1.483	13.05	0.12	0.65	8.80
GF(p^{10})	0.104	0.166	0.837	1.272	10.01	0.13	0.66	7.87
GF(p^7)	0.706	0.111	0.673	0.719	6.177	0.15	0.94	8.59
GF(p^6)	0.704	0.073	0.462	0.608	5.170	0.12	0.76	8.50
GF(p^5)	0.794	0.081	0.650	0.731	5.142	0.11	0.89	7.03
GF(p^3)	1.591	0.064	0.383	0.423	2.822	0.15	0.91	6.67
GF(p^2)	5.354	0.063	0.541	0.582	6.173	0.11	0.93	10.6
GF(p)		0.072	0.512	0.597	18.97	0.12	0.86	31.8

Table 10. Timings (in μsec) for field operations on Alpha 533MHz

4.2 Timings for Elliptic Curve Arithmetic

Tables 11 and 12 show the speed of elliptic curve operations on Pentium II and Alpha 21164 microprocessors, respectively. The tables show that it is always preferable to use projective coordinates with affine precomputation as expected from the I/M ratios in Tables 9 and 10. Obviously, we have the best performance on Pentium II with an elliptic curve over GF(p^5) and on Alpha 21164 with an elliptic curve over GF(p^3). It is also worth noting that an elliptic curve in GF(p) gives an almost comparable speed to the best in both microprocessors. This is due to the special choice of p such that $p = 2^{160} - 2933$. Also note that the speed ratio of elliptic doubling to addition is around 0.5 in GF(2^n) and around 0.75 in GF(p^n).

For comparison, we summarized the timings for elliptic scalar multiplication using Frobenius expansion in Table 13 (from [15]). The table shows that we can

coordinates		Affine			Proj.($Z_1 = 1$)		
Field	$\|k\|$	$P+P$	$P+Q$	kP	$P+P$	$P+Q$	kP
GF(2^{162})	162	63.8	64.2	12147	16.8	34.9	3978
GF(p^{13})	178	25.8	22.9	5034	16.7	23.2	3528
GF(p^{12})	178	22.9	20.4	4530	15.0	20.9	3206
GF(p^{11})	160	32.1	29.1	5991	20.2	27.8	4104
GF(p^{10})	160	27.0	24.3	5075	17.0	22.0	3446
GF(p^7)	168	19.2	17.4	3780	11.6	16.0	2457
GF(p^6)	168	15.3	13.8	3016	9.52	13.0	1997
GF(p^5)	160	12.5	10.9	2341	8.53	11.6	1693
GF(p^3)	171	16.6	14.7	3316	11.2	15.0	2397
GF(p^2)	178	25.3	23.7	5285	9.15	12.3	2070
GF(p)	160	49.2	47.4	9137	9.04	12.1	1983

Table 11. Timings (in μsec) for elliptic curve operations on Pentium II 266MHz

achieve about 2 to 3 times speed-up with subfield curves. However, when using such a special curve, we should be careful for the security consequence related to the special structure (e.g., see [25]).

5 Conclusion

We presented various speed-up techniques for field arithmetic in GF(p^n). The main improvements presented in this paper consist of various optimizations in field multiplication and inversion and careful choices of field parameters to speed up field arithmetic. Since the presented optimization techniques are focused on more primitive field arithmetic, the performance improvement in practical implementations will be more substantial than with optimizations in elliptic curve arithmetic. We also presented implementation results on two popular microprocessors, Pentium II and Alpha 21164.

Acknowledgement: The authors would like to thank an anonymous referee for his constructive comments, which were very helpful in improving the presentation of this paper.

References

1. G.B.Agnew, R.C.Mullin and S.A.Vanstone, An implementation of elliptic curve cryptosystems over$F_{2^{155}}$, *IEEE J. Selected Areas in Commum.*, 11, 5, 1993, pp.804-813. 408
2. D.V.Bailey and C.Paar, Optimal extension field for fast arithmetic in public key algorithms, *Advances in Cryptology-CRYPTO'98*, LNCS 1462, Springer-Verlag, 1998, pp.472-485. 405, 410

coordinates		Affine			Proj.($Z_1 = 1$)		
Field	$\|k\|$	$P+P$	$P+Q$	kP	$P+P$	$P+Q$	kP
GF(2^{162})	162	15.0	14.9	2875	5.46	10.9	1253
GF(p^{13})	178	20.4	18.4	3970	12.1	17.7	2514
GF(p^{12})	178	18.0	16.0	3492	10.4	15.3	2204
GF(p^{11})	160	20.1	18.3	3733	11.1	16.2	2270
GF(p^{10})	160	16.6	14.9	3046	9.89	14.3	2002
GF(p^7)	168	10.4	9.13	2028	6.60	8.81	1364
GF(p^6)	168	8.37	7.48	1675	4.81	6.54	1008
GF(p^5)	160	9.01	7.86	1657	5.97	8.08	1204
GF(p^3)	171	5.64	4.87	1143	3.83	5.06	816
GF(p^2)	178	9.66	8.89	2009	5.31	6.95	1138
GF(p)	160	22.7	21.8	4216	5.42	7.34	1142

Table 12. Timings (in μsec) for elliptic curve operations on Alpha 533MHz

	algorithm		signed binary		Alg. LL		Alg. LL-SI	
μP	field	$\|k\|$	A	P	A	P	A	P
Pentium II 266MHz	GF(p^{13})	178	1.96	1.85	1.66	1.57	1.46	1.36
	GF(p^{11})	160	2.34	2.11	2.01	1.83	1.77	1.60
	GF(p^7)	168	1.71	1.44	1.49	1.25	1.40	1.17
Alpha 21164 533MHz	GF(p^{13})	178	1.57	1.40	1.35	1.19	1.15	1.04
	GF(p^{11})	160	1.50	1.25	1.28	1.08	1.11	0.97
	GF(p^7)	168	0.93	0.80	0.80	0.69	0.76	0.66

Table 13. Timings (in msec) for scalar multiplication kP using Frobenius expansion

3. D.V.Bailey and C.Paar, Elliptic curve cryptosystems over large characteristic extension fields, preprint, 1999. 412, 416
4. J.H.Cheon, S.M.Park, S.W.Park and D.H.Kim, Two efficient algorithms for arithmetic of elliptic curves using Frobenius map, *Public Key Cryptography*, LNCS 1431, S-V, 1999, pp.195-202. 409
5. H.Cohen, *A course in computational number theory*, Graduate Texts in Math. 138, Springer-Verlag, 1993, Third corrected printing, 1996. 408
6. H.Cohen, A.Miyaji and T.Ono, Efficient elliptic curve exponentiation, *Information and Communications Security*, LNCS 1334, Springer-Verlag, 1997, pp.282-290. 408
7. H.Cohen, A.Miyaji and T.Ono, Efficient elliptic curve exponentiation using mixed coordinates, *Advances in Cryptology-ASIACRYPT'98*, LNCS 1514, Springer-Verlag, 1998, pp.50-65. 405, 407, 408
8. J.Guajardo and C.Paar, Efficient algorithms for elliptic curve cryptosystems, *Advances in Cryptology-CRYPTO'97*, LNCS 1294, Springer-Verlag, 1997, pp.342-356. 405
9. T.Kobayashi, H.Morita, K.Kobayashi and F.Hoshino, Fast elliptic curve algorithm combining Frobenius map and table reference to adapt to higher characteris-

tic, *Advances in Cryptology-EUROCRYPT'99*, LNCS 1592, Springer-Verlag, 1999, pp.176-189. 405, 409, 412, 416

10. N.Koblitz, Elliptic curve cryptosystems, *Math. Comp.*, 48, 1987, pp.203-209. 405
11. N.Koblitz, CM curves with good cryptographic properties, *Advances in Cryptology-CRYPTO'91*, LNCS 576, Springer-Verlag, 1992, pp.279-287. 409
12. K.Koyama and Y.Tsuruoka, Speeding up elliptic cryptosystems using a signed binary method, *Advances in Cryptology-CRYPTO'92*, LNCS 740, Springer-Verlag, 1993, pp.345-357. 405, 408
13. D.E. Knuth, *The art of Computer Programming: Seminumerical Algorithms*, 3rd edition, Addison Wesley, 1998. 411, 414
14. C.H.Lim and P.J.Lee, A key recovery attack on discrete log-based schemes using a prime order subgroup, *Advances in Cryptology-CRYPTO'97*, LNCS 1294, Springer-Verlag, 1997, pp.249-263. 409
15. C.H.Lim and H.S.Hwang, Fast elliptic scalar multiplication with precomputation, preprint, 1999. 407, 409, 418
16. J.Lopez and R.Dahab, Improved algorithms for elliptic curve arithmetic in GF(2^n), *Selected Areas in Cryptography*, LNCS 1556, Springer-Verlag, 1999, pp.201-212.
17. J.Lopez and R.Dahab, Fast multiplication on elliptic curves over GF(2^m) without precomputation, *Cryptographic Hardware and Embedded Systems*, LNCS 1717, Springer-Verlag, 1999. 408, 409
18. W.Meier and O.Staffelbach, Efficient multiplication on certain non-supersingular elliptic curves, *Advances in Cryptology-CRYPTO'92*, LNCS 740, Springer-Verlag, 1993, pp.333-344. 409
19. V.S.Miller, Use of elliptic curves in cryptography, *Advances in Cryptology-CRYPTO'85*, LNCS 218, Springer-Verlag, 1986, pp.417-426. 405
20. P.L.Montgomery, Speeding the Pollard and elliptic curve methods of factorization, *Math. of Computation*, 48, 177, 1987, pp.243-264. 408
21. V.Muller, Fast multiplication on elliptic curves over small fields of characteristic two, *J. of Cryptology*, vol.11, no.4, 1998, pp.219-234. 409
22. A.Schroeppel, H.Orman, S.O'Malley and O.Spatschek, Fast key exchange with elliptic curve systems, *Advances in Cryptology-CRYPTO'95*, LNCS 963, Springer-Verlag, 1995, pp.43-56. 405, 412
23. J.A.Solinas, An improved algorithm for arithmetic on a family of elliptic curves, *Advances in Cryptology-CRYPTO'97*, LNCS 1294, S-V, 1997, pp.357-371. 405, 408, 409
24. E.De Win, A.Bosselaers and S.Vandenberghe, A fast software implementation for arithmetic operations in GF(2^n), *Advances in Cryptology-ASIACRYPT96*, LNCS 1163, Springer-Verlag, 1996, pp.65-76. 405
25. M.J.Wiener and R.J.Zuccherato, Faster attacks on elliptic curve cryptosystems, *Selected Areas in Cryptography*, LNCS 1556, Springer-Verlag, 1999, pp.190-200. 409, 419
26. IEEE P1363: Standard Specifications for Public Key Cryptography, *Working Draft*, Oct. 1998. 405, 406, 407, 409
27. ANSI X9.62: The elliptic curve digital signature algorithm, *Working Draft*, Oct. 1998. 405, 409
28. ANSI X9.63: Elliptic curve key agreement and key transport protocols, *Working Draft*, Oct. 1998. 405, 409