Speeding up Elliptic Scalar Multiplication with Precomputation

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
E-mail: {chlim,hyosun}@future.co.kr

Abstract. It is often required in many elliptic curve cryptosystems to compute kG for a fixed point G and a random integer k. In this paper we present improved algorithms for such elliptic scalar multiplication. Implementation results on Pentium II and Alpha 21164 microprocessors are also provided to demonstrate the presented improvements in actual implementations.

1 Introduction

Let E be an elliptic curve defined over a finite field F ($F = \mathrm{GF}(2^n)$ or $\mathrm{GF}(p^n)$ for a prime p). Let G be a point of prime order in E. Elliptic scalar multiplication is to compute kG for random k. The performance of elliptic curve cryptosystems mainly depends on how efficiently this scalar multiplication can be performed. If G is random, the signed window algorithm is the most preferred algorithm for general scalar multiplication (e.g., see [6,7]). If E is defined over a small subfield such as $\mathrm{GF}(2^r)$ with $r|n$ or $\mathrm{GF}(p)$ for $n > 1$, then general scalar multiplication can be performed much faster using Frobenius expansion [10,15,18,16,4,9].

On the other hand, it is often required in elliptic curve cryptosystems to compute kG for a fixed point G. Since G is now fixed, we can substantially speed up the computation of kG using a precomputed table. Several methods have been developed for fast exponentiation using precomputation over a generic group [3,17,11], which can thus be applied equally well to the elliptic curve group. Among them, the Lim-Lee algorithm (LL algorithm, for short) is known to provide higher efficiency and flexibility in time-storage tradeoffs.

In this paper we investigate further improvements of the LL algorithm for elliptic scalar multiplication. Note that field inversion is most expensive among field operations required for elliptic curve arithmetic in most interesting fields. So, we tried to reduce the number of field inversions, at the cost of more field multiplications, utilizing the parallelizability of the LL algorithm and the simultaneous inversion technique [5, Algorithm 10.3.4]. Obviously, the amount of improvement that can be achieved with the resulting algorithm, Algorithm LL-SA, depends on the cost ratio of field inversion to multiplication. Our implementations on Pentium II and Alpha 21164 show that Algorithm LL-SA achieves about 20% speed-up over Algorithm LL in most interesting fields. Further improvement

can be obtained by computing many scalar multiples in parallel. This simultaneous scalar multiplication algorithm, Algorithm LL-SM, may be useful for heavy loaded security servers, which often need to process hundreds of transactions (requiring scalar multiplications) at a time. We also show that these algorithms can be used to speed up general scalar multiplication using Frobenius expansion.

This paper is organized as follows. In section 2 we briefly summarize elliptic curve arithmetic (with some improvements) in GF(2^n) and GF(p^n). We then present improvements of the LL algorithm using simultaneous elliptic addition (Algorithm LL-SA) and simultaneous elliptic scalar multiplication (Algorithm LL-SM) in sections 3 and 4, respectively. Section 5 deals with application of LL algorithms to speed up general scalar multiplication using Frobenius expansion in GF(p^n). Finally we present our implementation results in section 6 and conclude in section 7.

2 Elliptic Curve Arithmetic in Finite Fields

2.1 Affine Coordinates

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

$$\begin{aligned} y^2 + xy &= x^3 + ax^2 + b \ (a, b \in F,\ b \neq 0) \qquad \text{if } \mathrm{char}(F) = 2, \\ y^2 &= x^3 + ax + b \ (a, b \in F,\ 4a^2 + 27b^3 \neq 0) \ \text{ if } \mathrm{char}(F) > 3, \end{aligned}$$

together with a 'point at infinity'. Addition/doubling formulas in this affine representation are summarized in Table 1.

field	operation	λ	addition formulas
GF(2^n)	addition (A_e) ($x_0 \neq x_1$)	$\lambda = \frac{y_1+y_0}{x_1+x_0}$	$x_2 = \lambda^2 + \lambda + x_0 + x_1 + a$ $y_2 = \lambda(x_0 + x_2) + x_2 + y_0$
	doubling (D_e) ($x_0 = x_1$)	$\lambda = x_0 + \frac{y_0}{x_0}$	$x_2 = \lambda^2 + \lambda + a$ $y_2 = \lambda(x_0 + x_2) + x_2 + y_0$
GF(p^n)	addition (A_e) ($x_0 \neq x_1$)	$\lambda = \frac{y_1-y_0}{x_1-x_0}$	$x_2 = \lambda^2 - (x_0 + x_1)$ $y_2 = \lambda(x_0 - x_2) - y_0$
	doubling (D_e) ($x_0 = x_1$)	$\lambda = \frac{3x_0^2+a}{2y_0}$	$x_2 = \lambda^2 - 2x_0$ $y_2 = \lambda(x_0 - x_2) - y_0$

Table 1. Addition formulas in affine coordinates: $(x_2, y_2) = (x_0, y_0) + (x_1, y_1)$

2.2 Projective Coordinates

There is another representation of points, the so-called (weighted) projective representation, which eliminates the expensive field inversion at the cost of more field multiplications.

GF(2^n) For conversions between affine and projective coordinates, we used the transformation in [14]: $x = \frac{X}{Z}, \quad y = \frac{Y}{Z^2}$. To the best of our knowledge, this is the best known conversion rule for GF(2^n). The resulting formulas for elliptic addition and doubling are given below.[1]

- Addition formula: $(X_2, Y_2, Z_2) = (X_0, Y_0, Z_0) + (X_1, Y_1, 1)$

$$\begin{array}{l} A = X_0 + X_1 Z_0 \\ B = Y_0 + Y_1 Z_0^2,\ C = AZ_0 \end{array} \Longrightarrow \begin{array}{l} Z_2 = C^2,\ X_2 = B^2 + A^2(C + aZ_0^2) + BC \\ Y_2 = (BC + Z_2)(X_2 + X_1 Z_2) + (X_1 + Y_1)Z_2^2 \end{array}$$

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

$$Z_2 = X_0^2 Z_0^2, \quad X_2 = X_0^4 + bZ_0^4, \quad Y_2 = bZ_0^4(X_2 + Z_2) + X_2(aZ_2 + Y_0^2).$$

The addition formula requires 9 (8 general, 1 constant) multiplications and 5 squarings, while the doubling formula requires 5 (3 general, 2 constant) multiplications and 5 squarings. Note that the above addition formula requires one less multiplications than the formula given in [14]. If $a = 0$, then we can further reduce one multiplication in each formula.

GF(p^n) The addition/doubling formulas described here are essentially the same as those of the IEEE P1363 Draft [20]. The coordinates conversion is done by $x = \frac{X}{Z^2}, \quad y = \frac{Y}{2Z^3}$.[2] So the affine coordinate (x, y) should be mapped to the projective coordinate $(X, Y, Z) = (x, 2y, 1)$. The resulting formulas for elliptic addition/doubling are described below, where we only consider the special case of $Z_1 = 1$ as before.

- Addition formula: $(X_2, Y_2, Z_2) = (X_0, Y_0, Z_0) + (X_1, Y_1, 1)$

$$\begin{array}{l} A = X_0 + X_1 Z_0^2,\ B = X_0 - X_1 Z_0^2 \\ C = Y_0 + Y_1 Z_0^3, \quad D = Y_0 - Y_1 Z_0^3,\ E = 2B \end{array} \Longrightarrow \begin{array}{l} Z_2 = Z_0 E, \quad X_2 = D^2 - AE^2 \\ Y_2 = D(AE^2 - 2X_2) - E^2 BC \end{array}$$

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

$$\begin{array}{l} A = 3X_0^2 + aZ_0^4 \\ B = 2X_0 Y_0^2,\ C = Y_0^4 \end{array} \Longrightarrow \begin{array}{l} Z_2 = Y_0 Z_0,\ X_2 = A^2 - B \\ Y_2 = A(B - 2X_2) - C \end{array}$$

The above formulas show that elliptic addition requires 8 multiplications and 3 squarings, while elliptic doubling requires 4 (3 general, 1 constant) multiplications and 6 squarings. If $a = -3$, then the variable A in doubling can be computed by $A = 3(X_0 + Z_0^2)(X_0 - Z_0^2)$, so one can save 2 squarings in this case.

[1] Here we only describe the special case of $Z_1 = 1$ for elliptic addition, which corresponds to the case where precomputation is done in affine coordinates in double-and-add algorithms for scalar multiplication. This special case gives better performances in almost all cases.

[2] The factor 2 in y is included to eliminate the modular division by 2 appearing in the addition formula when using $y = \frac{Y}{Z^3}$ (see A.10.5 in [20]). 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 (see Sect.6).

2.3 Performance and Preferred Coordinates

In Table 2 we summarized the number of field operations for elliptic curve arithmetic in affine and projective coordinates. Here the capital letters I, M, S and A denote field operations of inversion, multiplication, squaring and addition, respectively. We assumed fixed values for constant a for performance reason: $a = 0$ for GF(2^n) and $a = -3$ for GF(p^n). It should be noted that these special values for constant a do not place much restriction in the choice of elliptic curves, since the proportion of elliptic curves that can be rescaled to have the above values for constant a is approximately 1/2 for GF(2^n) and 1/2 or 1/4, depending on the residue of p mod 4, for GF(p^n) (see Appendix A in [20]).

field	coordinates	doubling (D_e)	addition (A_e)
GF(2^n)	Affine	$1I + 2M + 1S + 5A$	$1I + 2M + 1S + 7A$
($a = 0$)	Proj.($Z_1 = 1$)	$4M + 5S + 3A$	$8M + 5S + 8A$
GF(p^n)	Affine	$1I + 2M + 2S + 7A$	$1I + 2M + 1S + 6A$
($a = -3$)	Proj.($Z_1 = 1$)	$4M + 4S + 9A$	$8M + 3S + 9A$

Table 2. The number of field operations for elliptic addition/doubling

To simplify performance comparisons, we will use the following assumptions on speed ratios between field operations throughout this paper: $1S = 0.15M$, constant multiplication $= 0.5M$ for GF(2^n) and $1S = 0.8M, 1A = 0.15M$ for GF(p^n) (addition times in GF(2^n) neglected). Of course, these ratios may vary from implementation to implementation, but our optimized implementations on P6 and Alpha microprocessors (see Sect.6) show that in most interesting fields the above assumptions are reasonable enough for theoretical comparison of computational complexity.

The cost ratio of field inversion to multiplication (I/M) is a key factor in determining a preferred coordinate system. So, let us find the I/M value at the break-even point between affine representation and projective representation (with $Z_1 = 1$). For this, suppose that $r = N_{D_e}/N_{A_e}$ (i.e., r elliptic doublings are required for each elliptic addition in a scalar multiplication algorithm). For example, we have $r = 6$ for the signed window algorithm with window size 4 and $r < 1$ for the LL algorithm. From Table 2 and the assumptions on speed ratios between field operations, we can obtain the following relations at the break-even point:

$$I/M = \begin{cases} 2.60 + \frac{4.00}{r+1} & \text{for GF}(2^n), \\ 3.90 + \frac{4.15}{r+1} & \text{for GF}(p^n). \end{cases}$$

Thus, for large r, it is almost always preferable to do elliptic scalar multiplication in projective coordinates. However, in the LL algorithm, we have $0.2 < r < 0.7$ for most interesting parameters, so $4.95 < I/M < 5.93$ for GF(2^n) and $6.34 < I/M < 7.36$ for GF(p^n). Thus, as we will see later, affine coordinates may yield better performances than projective coordinates in the case of GF(p^n).

3 The Improved LL Algorithm for Scalar Multiplication

3.1 The Original LL Algorithm

We briefly describe the Lim-Lee algorithm for elliptic scalar multiplication kG for a fixed point G and analyze its performance. First, the multiplier k of l bits is divided into hv subblocks of b bits as follows (see Figure 1):

$$k = \sum_{u=0}^{l-1} 2^u e_u = \sum_{i=0}^{h-1} \left(\sum_{j=0}^{v-1} k_{i,j} 2^{bj} \right) 2^{ia}, \text{ where}$$

$$a = \lceil \frac{l}{h} \rceil, \quad b = \lceil \frac{a}{v} \rceil, \quad k_{i,j} = \sum_{t=0}^{b-1} 2^t e_{ia+jb+t}.$$

$k_{0,v-1}$	$\cdots$	$k_{0,1}$	$k_{0,0}$
$k_{1,v-1}$	$\cdots$	$k_{1,1}$	$k_{1,0}$
$\vdots$	$\cdots$	$\vdots$	$\vdots$
$k_{h-1,v-1}$	$\cdots$	$k_{h-1,0}$	$k_{h-1,0}$

$\leftarrow b \rightarrow$

$\longleftarrow a \longrightarrow$

Fig. 1. Partition of an l-bit multiplier k for the LL Algorithm

In the (off-line) precomputation stage, we compute and store the point $GG[I][j]$ as follows:

$$G_{i,j} = 2^{ia+jb} G \quad \text{for } 0 \le i < h \text{ and } 0 \le j < v,$$
$$GG[I][j] = \sum_{i=0}^{h-1} e_i G_{i,j} \quad \text{for } 0 \le j < v, \quad \text{where } I = \sum_{i=0}^{h-1} 2^i e_i. \tag{1}$$

Using these precomputed values, we can express kG as

$$kG = \sum_{j=0}^{v-1} \sum_{i=0}^{h-1} k_{i,j} G_{i,j} = \sum_{t=0}^{b-1} 2^t \left(\sum_{j=0}^{v-1} \sum_{i=0}^{h-1} e_{ia+jb+t} G_{i,j} \right),$$
$$= \sum_{t=0}^{b-1} 2^t \left(\sum_{j=0}^{v-1} GG[I_{j,t}][j] \right), \text{ where } I_{j,t} = \sum_{i=0}^{h-1} 2^i e_{ia+jb+t}. \tag{2}$$

Note that $I_{j,t}$ corresponds to the t-th bit column of the j-th block column in Figure 1. Now, we can compute kG for each new value of k using equation (2) as shown in Algorithm LL.

Algorithm LL
$T := \sum_{j=0}^{v-1} GG[I_{j,b-1}][j]$;
for $t := b-2$ **to** 0 **step** -1
$\quad T := 2T$;
$\quad T := T + \sum_{j=0}^{v-1} GG[I_{j,t}][j]$;
return T;

Let us count the number of additions/doublings required by Algorithm LL. Obviously, we only need $(b-1)$ doublings. For the number of additions required, we note that the number of $GG[I_{j,t}][j]$ to be added is at most a. Therefore, we can see that the total cost for the worst case is given by

$$C_{LLw}(l,h,v) = (a-1)A_e + (b-1)D_e.$$

Let q be the probability of a bit being zero (so the probability of $I_{j,t}$ being zero is q^h). Then we can easily derive the expected number of additions/doublings as

$$C_{LLa}(l,h,v) = (a - q^h(a + (ah-l)(q^{-1}-1)) - 1)A_e + (b-1)D_e. \quad (3)$$

For random k, we may assume that $q = 1/2$. In this cse, equation (6) becomes

$$C_{LLa}(l,h,v) = \left(a - 1 - \frac{a + (ah-l)}{2^h}\right) A_e + (b-1)D_e. \quad (4)$$

It is also easy to see that Algorithm LL requires the storage for $(2^h - 1)v$ precomputed points and that the cost for precomputation is given by

$$C_{LLp}(l,h,v) = v(2^h - h - 1)A_e + b(hv-1)D_e.$$

Table 3 shows the average number of field inversions and multiplications, $(N_I + N_M)$, given by $C_{LLa}(160,h,v)$ for some selected parameters h and v, where we used the assumptions in Sect.2.3 to compute the equivalent number of field multiplications required for elliptic addition/doubling. In the case of projective coordinates, we also included the cost for coordinates conversion back to affine coordinates. The last two columns of Table 3 show the I/M ratios at the break-even point between computations in affine and projective coordinates. The ratios range from 5 to 6 in GF(2^n) and from 6.5 to 8 in GF(p^n). Our implementations on P6 and Alpha (see Sect.6) show that actual I/M ratios are larger than 10 for elliptic curves in GF(2^n) and in GF(p^n) with small n, so projective coordinates are preferred for these cases.

config.	storage	Affine $(N_I + N_M)$		Proj.		I/M at B.E.P.	
$h \times v$	$(2^h - 1)v$	GF(2^n)	GF(p^n)	GF(2^n)	GF(p^n)	GF(2^n)	GF(p^n)
2×2	6	98.0+210.7	98.0+399.6	1+684.1	1+1031	4.88	6.51
2×4	12	78.0+167.7	78.0+306.6	1+599.1	1+859.8	5.60	7.18
3×2	14	72.0+154.8	72.0+291.1	1+515.1	1+766.9	5.08	6.70
3×4	28	59.0+126.8	59.0+230.6	1+459.9	1+655.8	5.74	7.33
4×2	30	55.5+119.3	55.5+223.4	1+402.3	1+595.4	5.19	6.83
4×4	60	45.5+ 97.8	45.5+176.9	1+359.8	1+509.9	5.89	7.48
5×2	62	45.0+ 96.8	45.0+180.8	1+328.4	1+484.9	5.26	6.91
5×4	124	37.0+ 79.5	37.0+143.5	1+294.4	1+416.5	5.97	7.58
6×2	126	38.5+ 82.9	38.5+155.0	1+280.9	1+415.4	5.27	6.94
6×4	252	31.5+ 67.8	31.5+122.4	1+251.2	1+355.6	6.00	7.63
7×2	254	32.8+ 70.5	32.8+131.9	1+239.8	1+354.4	5.32	7.00
7×4	508	26.8+ 57.6	26.8+104.0	1+214.3	1+303.1	6.07	7.72
8×2	510	27.9+ 60.0	27.9+111.9	1+206.0	1+303.4	5.42	7.11
8×4	1020	22.9+ 49.3	22.9+ 88.6	1+184.7	1+260.6	6.18	7.85

Table 3. Average performance of Algorithm LL for computing kG with $|k| = 160$

3.2 The Improved LL Algorithm

Computation of multiple inverses modulo the same modulus can be substantially speeded up using Montgomery's trick to parallel inversion [5, Algorithm 10.3.4]. For example, to compute inverses of A and B modulo p, we first compute $C = (AB)^{-1} \bmod p$ and then $A^{-1} = CB \bmod p$ and $B^{-1} = CA \bmod p$. In general, this simultaneous inversion algorithm requires 1 inversion and $3(t-1)$ multiplications mod p for t-simultaneous inversion. Therefore, from Table 2, we can see that t-simultaneous elliptic addition in GF(p^n) requires the computational cost of $(I - 3M) + t(5M + S + 6A)$. Similarly, t-simultaneous elliptic addition in GF(2^n) requires $(I - 3M) + t(5M + S + 7A)$. This technique thus enables us to replace one field inversion by about 3 field multiplications for large t. The resulting cost savings are substantial, since field inversion costs more than 3 field multiplications in most interesting fields.

Now, let us consider how to achieve a maximal improvement of Algorithm LL using the simultaneous addition algorithm. First note that in Algorithm LL we may precompute and store the following b points ahead of time:

$$GGG[t] = \sum_{j=0}^{v-1} GG[I_{j,t}][j] \quad \text{for } 0 \le t \le b-1. \tag{5}$$

Then, we can just add one point $GGG[t]$ to T in the t-th iteration of the **for**-loop (see Algorithm LL-SA). Thanks to the high degree of parallelism existing in equation (5), we can take much advantage of simultaneous inversion in this on-line precomputation stage.

A naive way to evaluate equation (5) is to iterate b-simultaneous elliptic addition $v-1$ times (so, $v-1$ inversions required). However, the number of

Algorithm LL-SA

for $t := 0$ **to** $b-1$ **step** 1
  $GGG[t] := \sum_{j=0}^{v-1} GG[I_{j,t}][j]$;
$T := GGG[b-1]$;
for $t := b-2$ **to** 0 **step** -1
  $T := 2T$;
  $T := T + GGG[t]$;
return T;

inversions required can be further reduced by performing b-simultaneous elliptic additions in parallel (based on the binary tree structure). E.g., if $v = 4$, we do the computation as follows:

1. $GGG[t] = GG[I_{0,t}][0] + GG[I_{1,t}][1]$ for $0 \le t \le b-1$ and
   $TTT[t] = GG[I_{2,t}][2] + GG[I_{3,t}][3]$ for $0 \le t \le b-1$.
2. $GGG[t] = GGG[t] + TTT[t]$ for $0 \le t \le b-1$.

This way we can reduce the number of inversions from $v-1$ to $\lceil \log_2 v \rceil$. This method of course increases the requirement for temporary storage from b to $\lfloor \frac{v}{2} \rfloor b$.

Suppose that c elliptic additions are required for the on-line precomputation of $GGG[t]$'s. This requires field operations given by

$$C_{sa}(c) = \begin{cases} \lceil \log_2 v \rceil (I - 3M) + c(5M + S + 7A) & \text{for GF}(2^n), \\ \lceil \log_2 v \rceil (I - 3M) + c(5M + S + 6A) & \text{for GF}(p^n). \end{cases} \quad (6)$$

Since c elliptic additions in Algorithm LL are now replaced by $C_{sa}(c)$ in Algorithm LL-SA, we can obtain the cost of Algorithm LL-SA as

$$C_{LL\text{-}SA}(l,h,v) = C_{LL}(l,h,v) - \Delta C(c), \text{ where } \Delta C(c) = cA_e - C_{sa}(c). \quad (7)$$

Thus we only need to find the average and worst case values of c, c_a and c_w.

In the worst case, we need $a-b$ additions in the precomputation stage, so we have $c_w = a-b$. Considering the probability of $GG[I][j]$ being 'point at infinity', we can find the expected value c_a as

$$\begin{aligned} c_a &= a - b - q^h(a + (ah - l)(q^{-1} - 1)) + \delta, \text{ where} \\ \delta &= q^{hv}(b + (ah - l)(q^{-1} - 1) + (bv - a)(q^{-h} - 1)). \end{aligned} \quad (8)$$

As before, assuming that $q = 1/2$, we can simplify equation (8) to

$$c_a = a - b - \frac{a + (ah - l)}{2^h} + \delta, \text{ where } \delta = \frac{b + (ah - l) + (bv - a)(2^h - 1)}{2^{hv}}. \quad (9)$$

The cost advantage ΔC of Algorithm LL-SA over Algorithm LL can be expressed in terms of field operations as follows.

$$\begin{aligned} &\text{Affine}: \quad \Delta C(c) = (c - \lceil \log_2 v \rceil)(I - 3M), \\ &\text{Proj.}: \quad \Delta C(c) = \begin{cases} 3(c + \lceil \log_2 v \rceil)M + 4cS + cA - \lceil \log_2 v \rceil I & \text{for GF}(2^n), \\ 3(c + \lceil \log_2 v \rceil)M + 2cS + 3cA - \lceil \log_2 v \rceil I & \text{for GF}(p^n). \end{cases} \end{aligned} \quad (10)$$

We evaluated the average performance of Algorithm LL-SA using equation (10) and Table 3. The result is shown in Table 4. Obviously, the amount of improvement of Algorithm LL-SA over Algorithm LL depends on the I/M ratio and becomes larger as the I/M ratio increases, since the improvement comes from the replacement of field inversions in Algorithm LL with about 3 field multiplications in Algorithm LL-SA. The I/M ratios at the break-even point shown in the last two columns can be used to determine which coordinates are preferred for the implementation of Algorithm LL-SA. From Tables 3 and 4 and the measured I/M ratios (Table 8 in Sect.6), we can see that the amount of improvement can be about 10 to 25% for projective coordinates and about 5 to 40% for affine coordinates.

config.	storage	Affine $(N_I + N_M)$		Proj.		I/M at B.E.P.	
$h \times v$	(temp.)	GF(2^n)	GF(p^n)	GF(2^n)	GF(p^n)	GF(2^n)	GF(p^n)
2×2	6(40)	76.5+275.2	76.5+464.1	2+600.1	2+914.2	4.36	6.04
2×4	12(40)	39.7+282.6	39.7+421.6	3+448.0	3+650.2	4.51	6.23
3×2	14(27)	52.1+214.5	52.1+350.8	2+436.9	2+658.3	4.44	6.14
3×4	28(28)	27.8+220.6	27.8+324.4	3+334.3	3+481.9	4.59	6.36
4×2	30(20)	38.7+169.8	38.7+273.8	2+335.1	2+502.5	4.51	6.23
4×4	60(20)	20.0+174.4	20.0+253.5	3+254.6	3+364.9	4.73	6.56
5×2	62(16)	30.9+139.1	30.9+223.1	2+270.9	2+405.5	4.57	6.31
5×4	124(16)	16.0+142.6	16.0+206.6	3+205.5	3+294.2	4.85	6.75
6×2	126(14)	26.7+118.4	26.7+190.5	2+231.7	2+347.5	4.59	6.36
6×4	252(14)	13.9+120.7	13.9+175.3	3+174.6	3+250.5	4.93	6.88
7×2	254(12)	22.7+100.8	22.7+162.1	2+196.9	2+295.5	4.63	6.43
7×4	508(12)	11.9+102.3	11.9+148.6	3+147.5	3+211.9	5.06	7.09
8×2	510(10)	19.0+ 86.8	19.0+138.7	2+167.2	2+250.2	4.73	6.57
8×4	1020(10)	10.0+ 88.1	10.0+127.4	3+125.0	3+179.3	5.28	7.41

Table 4. Average performance of Algorithm LL-SA for computing kG with $|k| = 160$

4 Simultaneous Scalar Multiplication

A central security server often needs to handle thousands of transactions, e.g., involving Diffie-Hellman key exchanges or digital signatures, at a peak time. For such a heavy loaded application, further speed up can be obtained by computing many scalar multiples simultaneously.

Suppose that we want to evaluate t scalar multiplications, k_iG ($0 \leq i < t$, $|k_i| = l$), at a time. We can then perform t instances of Algorithm LL-SA simultaneously, one for each k_iG. Let us call this algorithm as Algorithm LL-SM. Then the simultaneous inversion technique can be applied even to double-and-add parts of concurrent Algorithm LL-SA instances. We thus perform all

elliptic curve arithmetic in affine coordinates, but the number of field inversions required for t scalar multiplications is reduced to about $\lceil \log_2 v \rceil + 2b - 2$.

From the analysis of Sect.3, we can easily see that the average performance of Algorithm LL-SM (assuming that $q = 1/2$) is given by

$$C_{LL-SMa}(t,l,h,v) = (\lceil \log_2 v \rceil + 2b - \delta - 2)(I - 3M) + t\left(a - 1 - \frac{a + (ah - l)}{2^h}\right)\tilde{A}_e + t(b-1)\tilde{D}_e, \quad (11)$$

where δ is the same as before (equation (9)) and $\tilde{A}_e$ and $\tilde{D}_e$ are given by

$$\begin{aligned} \tilde{A}_e &= \begin{cases} 5M + 1S + 7A & \text{for GF}(2^n), \\ 5M + 1S + 6A & \text{for GF}(p^n). \end{cases} \\ \tilde{D}_e &= \begin{cases} 5M + 1S + 5A & \text{for GF}(2^n), \\ 5M + 2S + 7A & \text{for GF}(p^n). \end{cases} \end{aligned} \quad (12)$$

Therefore, for large t, the cost of Algorithm LL-SM per scalar multiplication, i.e., $C_{LL-SMa}(t,l,h,v)/t$, is almost the same as the cost of Algorithm LL in affine coordinates with field inversion replaced by 3 field multiplications. This would be the best achievable performance per scalar multiplication, as far as field inversion is more expensive than 3 field multiplications. For example, we tabulated in Table 5 the cost of Algorithm LL-SM per scalar multiplication for $t = 100$. As can be seen from the table, the number of field inversions required per scalar multiplication is less than 1 for large t. We can also see that Algorithm LL-SM improves over Algorithm LL-SA by more than 15% under the reasonable assumption of I/M ratios (see Table 8 in Sect.6).

5 Speeding up Scalar Multiplication Using ϕ-Expansion

We can view an elliptic curve E defined over GF(p) as an elliptic curve defined over GF(p^n). For such a subfield curve we can achieve a much higher performance using Frobenius expansion [9]. Let $P = (x, y)$ be a GF(p^n)-point on E. The Frobenius map ϕ is defined as $\phi : (x, y) \to (x^p, y^p)$ and satisfies the equation

$$\phi^2 - t\phi + p = 0 \text{ and } \phi^n = 1, \quad -2\sqrt{p} \le t \le 2\sqrt{p}. \quad (13)$$

This map ϕ can be evaluated only using $2(n-1)$ multiplications mod p (see [9]).

To compute kP, we first express the multiplier k using equation (13) as

$$k = \sum_{i=0}^{n-1} k_i \phi^i, \quad \text{where } |k_i| < \frac{p}{2}, \quad (14)$$

precompute n points $P_i = \phi^i(P)$ for $0 \le i < n$ and then compute kP as

$$kP = \sum_{i=0}^{n-1} k_i P_i. \quad (15)$$

configuration		storage		$(N_I + N_M)/t$	
$h \times v$	(a, b)	perm/temp	c_a	GF(2^n)	GF(p^n)
2×2	(80,40)	6/4000	22.5	0.77+502.4	0.77+691.4
2×4	(80,20)	12/4000	40.3	0.40+400.5	0.40+539.5
3×2	(54,27)	14/2700	20.9	0.52+369.2	0.52+505.5
3×4	(54,14)	28/2800	33.2	0.28+303.0	0.28+406.8
4×2	(40,20)	30/2000	17.8	0.39+284.7	0.39+388.7
4×4	(40,10)	60/2000	27.5	0.20+233.7	0.20+312.8
5×2	(32,16)	62/1600	15.1	0.31+230.8	0.31+314.8
5×4	(32, 8)	124/1600	23.0	0.16+190.1	0.16+254.1
6×2	(27,14)	126/1400	12.9	0.27+197.7	0.27+269.8
6×4	(27, 7)	252/1400	19.6	0.14+162.0	0.14+216.6
7×2	(23,12)	254/1200	11.1	0.23+168.3	0.23+229.6
7×4	(23, 6)	508/1200	16.9	0.12+137.7	0.12+184.0
8×2	(20,10)	510/1000	9.9	0.19+143.2	0.19+195.1
8×4	(20, 5)	1020/1000	14.9	0.10+117.7	0.10+157.1

Table 5. Average performance of Algorithm LL-SM for $t = 100$ and $|k_i| = 160$

Note that the bit-length of k_i's in equation (14) can always be made one bit less than the bit-length m of p ($|p| = m$) and that the negative signs can be absorbed into the precomputed points P_i's. So, we may assume that the coefficients k_i's in equation (15) are always positive integers of bit-length $m - 1$.

The main source of efficiency in this scalar multiplication using base-ϕ expansion is that the intermediate points P_i's can be evaluated almost free, only using $2(n-1)^2$ subfield multiplications, and thus about $\frac{n-1}{n}|k|$ elliptic doublings can be saved, compared to general scalar multiplication. The cost we have to pay for this improvement is a small amount of on-line precomputation (i.e., base-ϕ expansion of k and $(n-1)$ evaluations of ϕ), which costs less than a few elliptic additions.

The right-hand side of equation (15) can be efficiently evaluated using the signed binary algorithm with optimal signed encoding of k_i's [18] (this is actually the same as Type-II expansion in [9]). Since an optimal signed encoding of a t-bit integer can produce an integer of bit-length at most $t + 1$ and probability of a bit being zero 2/3, this computation can be done in $(m - 1)$ elliptic doublings and $(\frac{nm}{3} - 1)$ elliptic additions on average.

Further speed-up can be achieved using Algorithms LL/LL-SA, as can be expected from equation (15). Figure 2 shows some possible (actually best on average) arrangements of k_i's for using Algorithms LL/LL-SA. Here we only consider three field extensions of degree 7, 11 and 13, since they are most interesting in practice. Unlike Algorithms LL/LL-SA in Sect.3, we now have to do the precomputation required on-line. It is easy to see that the costs of on-line precomputation for the configurations shown in Figure 2 are given by $C_{LLp}(7m, 4, 2) = 15A_e$, $C_{LLp}(11m, 3, 4) = 13A_e$, and $C_{LLp}(13m, 3, 5) = 14A_e$,

$n = 7$

k_1	k_0
k_3	k_2
k_5	k_4
	k_6

$n = 11$

k_3	k_2	k_1	k_0
k_7	k_6	k_5	k_4
	k_{10}	k_9	k_8

$n = 13$

k_4	k_3	k_2	k_1	k_0
k_9	k_8	k_7	k_6	k_5
		k_{12}	k_{11}	k_{10}

Fig. 2. Arrangements of k_i's for base-ϕ scalar multiplication using Algorithm LL

respectively. Since it is preferable to do the precomputation in affine coordinates, we can obtain some speed-up with simultaneous inversion. In this case, the costs are given by $C_{LLp}(7m, 4, 2) = 2I + 94.5M$, $C_{LLp}(11m, 3, 4) = 2I + 81.1M$, and $C_{LLp}(13m, 3, 5) = 2I + 87.8M$, respectively.

Since the average Hamming weight of k_i's can also be reduced to approximately $\frac{1}{3}$ with some clever weight minimization strategy, we can obtain average performances for the evaluation of equation (15) using Algorithms LL/LL-SA by substituting $a = vm, b = m, l = nm$ and $q = 2/3$ in equations (3) and (8):

$$C_{LLa}(nm, h, v) = C_{LLp}(nm, h, v) + \left(\left(v - \left(\frac{2}{3}\right)^h \left(v + \frac{hv - n}{2}\right)\right) m - 1\right) A_e + (m-1) D_e,$$

$$C_{LL\text{-}SAa}(nm, h, v) = C_{LLa}(nm, h, v) - \Delta C(c_a), \text{ where}$$

$$c_a = \left(v - 1 - \left(\frac{2}{3}\right)^h \left(v + \frac{hv - n}{2}\right) + \left(\frac{2}{3}\right)^{hv} \left(1 + \frac{hv - n}{2}\right)\right) m.$$

Table 6 shows the number of elliptic additions and field inversions required for three methods of evaluating equation (15), where the computational costs for base-ϕ expansion and ϕ evaluations are not included.

coord.	n	$m = \lvert p \rvert$	signed binary	Algorithm LL	Algorithm LL-SA
Affine	7	28	83.0+307.1	70.2+372.4	55.4+416.8
	11	16	68.0+251.6	58.7+305.0	33.8+379.6
	13	14	67.9+251.2	59.1+311.5	30.9+396.0
Proj.	7	28	1.0+978.2	3.0+812.1	4.0+729.2
	11	16	1.0+802.0	3.0+701.9	5.0+560.3
	13	14	1.0+800.8	3.0+720.2	6.0+553.8

Table 6. Average performances ($N_I + N_M$) of three algorithms for base-ϕ scalar multiplication

Finally, it is worth noting that though we can obtain much higher efficiencies using Frobenius expansion with subfield curves, we should be careful for their security consequences. The structure allowing faster implementations may also allow faster attacks (e.g., see [19]). $\#E/\mathrm{GF}(p^m)$ (the order of $E/\mathrm{GF}(p^m)$) divides $\#E/\mathrm{GF}(p^n)$ if m divides n, so $\#E/\mathrm{GF}(p^n)$ contains at least small prime factors

of size$\#E/\mathrm{GF}(p)$. Since we have to use a prime order subgroup for ECC, this may increase the order of subfield curves more than necessary. Furthermore, the small prime factors in $\#E/\mathrm{GF}(p^n)$ may considerably weaken the resulting cryptosystem in many applications if proper precautions are not taken (by the small order subgroup attack in [12]).

6 Implementation and Discussion

We have implemented Algorithms LL, LL-SA and LL-SM on two different architectures: 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). Table 7 summarizes the parameters used for field constructions. 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 7 denote the largest possible prime orders (in bits) in $E/\mathrm{GF}(p^n)$. See [13] for details on selection criteria of field parameters and timings for field/EC arithmetic.

field	n	order	p	irred. poly.
$\mathrm{GF}(2^{162})$	162	162		$x^{162} + x^{27} + 1$
$\mathrm{GF}(p^n)$	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 7. Field constructions for elliptic curves

For better understanding of this presentation, we provided Table 8 summarizing various speed ratios between field operations and elliptic doubling to addition.[3] From Table 8, we can see that our assumptions given in Sect.2.3 are quite reasonable at least on P6 and Alpha family microprocessors, i.e.,$A/M \approx 0.15, S/M \approx 0.8$ in $\mathrm{GF}(p^n)$ and $S/M \approx 0.15$ in $\mathrm{GF}(2^n)$. Also note that I/M ranges from 5 to 7 for most fields (except for $\mathrm{GF}(p)$, $\mathrm{GF}(p^2)$ and $\mathrm{GF}(2^n)$).

[3] The figures in Table 8 are different from the figures in Tables 9-11 in [13]. At the time of writing [13], we didn't implement the field inversion method using exponentiation from [2] (Algorithm BP, for short). Though the multiplicative complexity of Algorithm BP seems higher than that of Algorithm IM in [13], our actual implementations show that Algorithm BP runs about 20 to 30% faster than Algorithm IM due to smaller overheads in other simple operations and loop controls.

μP	Pentium II/266MHz				Alpha 21164/533MHz			
Field	A/M	S/M	I/M	D_e/A_e	A/M	S/M	I/M	D_e/A_e
GF(2^{162})	0.03	0.13	14.0	0.47	0.05	0.16	10.5	0.48
GF(p^{13})	0.17	0.73	5.54	0.72	0.15	0.59	4.99	0.66
GF(p^{12})	0.18	0.74	6.63	0.73	0.15	0.61	6.11	0.64
GF(p^{11})	0.12	0.77	6.46	0.71	0.14	0.62	6.39	0.69
GF(p^{10})	0.14	0.78	6.04	0.77	0.18	0.66	5.98	0.70
GF(p^7)	0.11	0.79	6.05	0.73	0.14	0.87	6.17	0.72
GF(p^6)	0.13	0.82	6.41	0.73	0.18	0.87	5.79	0.70
GF(p^5)	0.18	0.82	5.89	0.74	0.12	0.81	4.60	0.76
GF(p^3)	0.18	0.80	7.59	0.75	0.15	0.89	6.88	0.74
GF(p^2)	0.16	0.86	19.9	0.73	0.13	0.90	10.4	0.75
GF(p)	0.15	0.88	42.7	0.74	0.12	0.85	31.7	0.75

Table 8. Speed ratios of field and elliptic curve operations

Timings for Algorithms LL/LL-SA/LL-SM on Pentium II/266MHz are given in Table 10, and timings on Alpha 21164/533MHz are given in Table 11. Here are some observations on the implementation results:

- As expected from the analysis in Sect.3 (compare the I/M ratios in Tables 3 and 4 with those in Table 8), Algorithms LL and LL-SA yield better performances in projective coordinates than in affine coordinates for GF(2^n) and GF(p^n) with $n \leq 3$.
- Algorithm LL-SA improves over Algorithm LL by about 10 to 25% in either coordinates, with some exceptions in GF(p), GF(p^2) and GF(2^n). The exceptions in these fields are actually expected from the speed ratio of I/M in Table 8 (i.e., much higher values of I/M).
- Compared to individual scalar multiplication using Algorithm LL-SA in preferred coordinates, simultaneous scalar multiplication using Algorithm LL-SM can significantly reduce the time per scalar multiplication (up to 40% for GF(p)).
- Algorithms LL/LL-SA/LL-SM can achieve 2 to 10 times speedup over the ordinary signed window algorithm for scalar multiplication.

Timings for elliptic scalar multiplication using Frobenius expansion are given in Table 9. We can see that Algorithm LL achieves about 15 to 20% improvement over the signed binary algorithm and that Algorithm LL-SA again improves over Algorithm LL by 5 to 15%.

7 Conclusion

Simultaneous inversion is a simple but poweful technique to speed up elliptic curve arithmetic with high degree of parallelism. Lim-Lee's algorithm for elliptic

	algorithm		w/o Frob.		binary		Alg. LL		Alg. LL-SA	
	field	$\|k\|$	A	P	A	P	A	P	A	P
Pentium II 266MHz	GF(p^{13})	178	4.19	3.48	1.66	1.86	1.43	1.58	1.34	1.36
	GF(p^{11})	160	5.17	4.07	2.03	2.11	1.78	1.81	1.62	1.58
	GF(p^{7})	168	3.03	2.42	1.40	1.44	1.24	1.24	1.19	1.16
Alpha 21164 533MHz	GF(p^{13})	178	3.19	2.51	1.27	1.41	1.08	1.17	1.03	1.04
	GF(p^{11})	160	3.15	2.23	1.27	1.24	1.09	1.05	1.01	0.95
	GF(p^{7})	168	1.75	1.32	0.79	0.77	0.69	0.67	0.66	0.63

Table 9. Timings for scalar multiplication using Frobenius expansion (in msec, A: affine, P: projective)

scalar multiplication for a fixed point (Algorithm LL) allows a very high degree of parallelism and thus can be substantially speeded up using the simultaneous inversion technique. This paper investigated such improvement on Lim-Lee's algorithm. More specifically, we presented and analyzed improved Lim-Lee's algorithms using simultaneous inversion: Algorithm LL-SA for computing a single scalar multiple and Algorithm LL-SM for computing many scalar multiples at a time. Implementation results of these algorithms on Pentium II and Alpha 21164 microprocessors were also provided to demonstrate practical performance improvement. We also showed that the presented algorithms can be used to speed up general elliptic scalar multiplication using Frobenius expansion.

References

1. 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.
2. D.V.Bailey and C.Paar, Inversion in Optimal Extension Fields, presented at *The Mathematics of Public-Key Cryptography*, Jun. 1999 (see also Elliptic curve cryptosystems over large characteristic extension fields by the same authors, preprint, 1999).
3. E.F.Brickell, D.M.Gordon, K.S.McCurley and D.Wilson, Fast exponentiation with precomputation, *Advances in Cryptology-EUROCRYPT'92*, LNCS 658, Springer-Verlag, 1993, pp.200-207.
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.
5. H.Cohen, *A course in computational number theory*, Graduate Texts in Math. 138, Springer-Verlag, 1993, Third corrected printing, 1996.
6. H.Cohen, A.Miyaji and T.Ono, Efficient elliptic curve exponentiation, *Information and Communications Security*, LNCS 1334, Springer-Verlag, 1997, pp.282-290.
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.
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.

9. T.Kobayashi, H.Morita, K.Kobayashi and F.Hoshino, Fast elliptic curve algorithm combining Frobenius map and table reference to adapt to higher characteristic, *Advances in Cryptology-EUROCRYPT'99*, LNCS 1592, Springer-Verlag, 1999, pp.176-189.
10. N.Koblitz, CM curves with good cryptographic properties, *Advances in Cryptology-CRYPTO'91*, LNCS 576, Springer-Verlag, 1992, pp.279-287.
11. C.H.Lim and P.J.Lee, More flexible exponentiation with precomputation, *Advances in Cryptology-CRYPTO'94*, LNCS 839, Springer-Verlag, 1994, pp.95-107.
12. 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.
13. C.H.Lim and H.S.Hwang, Fast implementation of elliptic curve arithmetic in GF(p^n), *Public Key Cryptography*, LNCS 1751, Springer-Verlag, 1999.
14. 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.
15. 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.
16. V.Muller, Fast multiplication on elliptic curves over small fields of characteristic two, *J. of Cryptology*, vol.11, no.4, 1998, pp.219-234.
17. P.de Rooij, Efficient exponentiation using precomputation and vector addition chains, *Advances in Cryptology-EUROCRYPT'94*, LNCS 950, Springer-Verlag, 1995, pp.389-399.
18. J.A.Solinas, An improved algorithm for arithmetic on a family of elliptic curves, *Advances in Cryptology-CRYPTO'97*, LNCS 1294, Springer-Verlag, 1997, pp.357-371.
19. 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.
20. IEEE P1363: Standard Specifications for Public Key Cryptography, *Working Draft*, Aug. 1999.

	field		GF(p^n)										GF(2^n)
	n		1	2	3	5	6	7	10	11	12	13	162
AFFINE	Win. Alg.		8.93	4.72	3.08	2.04	2.41	2.91	4.21	5.18	3.96	3.99	12.0
	LL	2 × 4	3.68	1.91	1.20	0.78	0.93	1.12	1.63	2.00	1.55	1.53	5.01
		3 × 4	2.79	1.44	0.91	0.59	0.70	0.85	1.22	1.52	1.16	1.15	3.77
		4 × 4	2.14	1.11	0.70	0.45	0.54	0.65	0.94	1.17	0.90	0.88	2.91
		5 × 4	1.75	0.91	0.57	0.37	0.44	0.53	0.77	0.95	0.73	0.73	2.37
		6 × 4	1.49	0.77	0.49	0.32	0.38	0.45	0.66	0.81	0.63	0.62	2.02
		7 × 4	1.27	0.66	0.41	0.27	0.32	0.39	0.56	0.69	0.53	0.53	1.72
		8 × 4	1.09	0.56	0.35	0.23	0.28	0.33	0.48	0.59	0.46	0.45	1.47
	LL | SA	2 × 4	2.19	1.29	0.99	0.69	0.80	0.97	1.39	1.69	1.28	1.35	3.39
		3 × 4	1.57	0.94	0.73	0.51	0.59	0.72	1.03	1.26	0.95	1.01	2.47
		4 × 4	1.15	0.70	0.55	0.39	0.45	0.55	0.79	0.95	0.72	0.77	1.84
		5 × 4	0.92	0.56	0.45	0.32	0.37	0.44	0.64	0.77	0.59	0.62	1.48
		6 × 4	0.81	0.49	0.39	0.27	0.32	0.38	0.55	0.67	0.51	0.54	1.28
		7 × 4	0.69	0.42	0.33	0.23	0.27	0.33	0.47	0.57	0.44	0.46	1.10
		8 × 4	0.58	0.36	0.29	0.20	0.23	0.28	0.40	0.49	0.37	0.39	0.93
	LL | SM	2 × 4	0.65	0.66	0.78	0.59	0.68	0.83	1.21	1.43	1.12	1.26	1.75
		3 × 4	0.48	0.49	0.58	0.44	0.51	0.62	0.90	1.07	0.83	0.93	1.30
		4 × 4	0.37	0.37	0.44	0.34	0.38	0.47	0.68	0.81	0.62	0.69	1.00
		5 × 4	0.30	0.30	0.36	0.27	0.31	0.38	0.55	0.65	0.50	0.56	0.81
		6 × 4	0.26	0.26	0.31	0.24	0.27	0.33	0.47	0.56	0.43	0.48	0.69
		7 × 4	0.22	0.22	0.26	0.20	0.23	0.28	0.40	0.48	0.37	0.41	0.59
		8 × 4	0.19	0.19	0.23	0.17	0.19	0.24	0.35	0.41	0.32	0.35	0.51
PROJECTIVE	Win. Alg.		1.96	1.86	2.23	1.68	1.88	2.31	3.41	4.06	3.01	3.34	3.91
	LL	2 × 4	0.93	0.92	1.12	0.86	0.96	1.19	1.74	2.06	1.54	1.70	2.45
		3 × 4	0.72	0.70	0.86	0.66	0.74	0.90	1.33	1.58	1.18	1.31	1.89
		4 × 4	0.57	0.55	0.67	0.51	0.57	0.71	1.04	1.23	0.92	1.02	1.50
		5 × 4	0.48	0.46	0.55	0.42	0.47	0.58	0.85	1.01	0.75	0.84	1.23
		6 × 4	0.42	0.40	0.47	0.36	0.41	0.50	0.73	0.87	0.65	0.72	1.06
		7 × 4	0.36	0.34	0.41	0.31	0.35	0.43	0.63	0.75	0.56	0.62	0.92
		8 × 4	0.32	0.30	0.35	0.27	0.30	0.37	0.54	0.65	0.48	0.53	0.80
	LL | SA	2 × 4	0.84	0.78	0.91	0.69	0.77	0.95	1.38	1.63	1.23	1.36	2.01
		3 × 4	0.66	0.60	0.68	0.52	0.58	0.71	1.03	1.23	0.92	1.02	1.54
		4 × 4	0.53	0.47	0.52	0.40	0.44	0.54	0.79	0.94	0.71	0.78	1.22
		5 × 4	0.45	0.39	0.43	0.32	0.36	0.45	0.65	0.78	0.58	0.64	1.01
		6 × 4	0.41	0.34	0.37	0.28	0.32	0.39	0.56	0.67	0.51	0.56	0.89
		7 × 4	0.37	0.30	0.32	0.24	0.28	0.33	0.49	0.58	0.44	0.48	0.78
		8 × 4	0.33	0.27	0.28	0.21	0.24	0.29	0.42	0.50	0.38	0.42	0.69

Table 10. Timings (in msec) for Algorithms LL, LL-SA and LL-SM for computing kG with $|k| = 160$ on Pentium II/266MHz (timings for Algorithm LL-SM denote timings per scalar multiplication for $t = 100$)

	field		GF(p^n)										GF(2^n)
	n		1	2	3	5	6	7	10	11	12	13	162
AFFINE	Win. Alg.		4.22	1.80	1.06	1.41	1.14	1.63	2.61	3.14	3.08	3.05	2.82
	LL	2 × 4	1.75	0.72	0.42	0.53	0.45	0.62	1.00	1.24	1.12	1.16	1.18
		3 × 4	1.32	0.54	0.32	0.40	0.34	0.47	0.76	0.94	0.85	0.87	0.90
		4 × 4	1.02	0.42	0.24	0.31	0.26	0.36	0.58	0.72	0.65	0.67	0.69
		5 × 4	0.83	0.34	0.20	0.25	0.21	0.30	0.48	0.59	0.54	0.55	0.56
		6 × 4	0.71	0.29	0.17	0.22	0.18	0.25	0.41	0.51	0.46	0.47	0.48
		7 × 4	0.60	0.25	0.15	0.18	0.16	0.22	0.35	0.43	0.39	0.40	0.41
		8 × 4	0.52	0.21	0.13	0.16	0.13	0.19	0.30	0.37	0.34	0.34	0.35
	LL | SA	2 × 4	1.09	0.56	0.35	0.49	0.40	0.55	0.88	1.05	0.97	1.04	0.88
		3 × 4	0.79	0.41	0.26	0.37	0.30	0.41	0.66	0.78	0.72	0.78	0.65
		4 × 4	0.58	0.31	0.20	0.28	0.23	0.31	0.50	0.59	0.55	0.59	0.49
		5 × 4	0.47	0.25	0.16	0.23	0.19	0.25	0.40	0.49	0.44	0.49	0.40
		6 × 4	0.41	0.22	0.14	0.20	0.16	0.22	0.35	0.42	0.38	0.42	0.34
		7 × 4	0.35	0.19	0.12	0.17	0.14	0.19	0.30	0.36	0.33	0.36	0.29
		8 × 4	0.30	0.16	0.11	0.15	0.12	0.16	0.26	0.31	0.28	0.31	0.25
	LL | SM	2 × 4	0.42	0.41	0.31	0.49	0.39	0.50	0.80	0.92	0.87	1.00	0.57
		3 × 4	0.31	0.31	0.23	0.36	0.29	0.37	0.58	0.68	0.65	0.74	0.43
		4 × 4	0.24	0.23	0.18	0.28	0.22	0.28	0.44	0.52	0.49	0.56	0.33
		5 × 4	0.19	0.19	0.14	0.22	0.18	0.23	0.36	0.42	0.40	0.45	0.26
		6 × 4	0.17	0.16	0.12	0.19	0.15	0.20	0.31	0.36	0.34	0.39	0.23
		7 × 4	0.14	0.14	0.11	0.17	0.13	0.17	0.27	0.31	0.30	0.33	0.19
		8 × 4	0.12	0.12	0.09	0.14	0.11	0.14	0.23	0.26	0.25	0.29	0.17
PROJECTIVE	Win. Alg.		1.13	1.06	0.79	1.24	0.90	1.23	2.00	2.24	2.12	2.39	1.22
	LL	2 × 4	0.56	0.57	0.41	0.63	0.47	0.63	1.02	1.18	1.10	1.28	0.77
		3 × 4	0.43	0.44	0.31	0.48	0.36	0.48	0.78	0.91	0.84	0.98	0.59
		4 × 4	0.34	0.34	0.24	0.38	0.28	0.38	0.61	0.71	0.66	0.77	0.47
		5 × 4	0.28	0.28	0.20	0.31	0.23	0.31	0.50	0.59	0.55	0.63	0.38
		6 × 4	0.24	0.24	0.17	0.27	0.20	0.27	0.43	0.50	0.47	0.54	0.33
		7 × 4	0.21	0.21	0.15	0.23	0.17	0.23	0.37	0.43	0.40	0.47	0.29
		8 × 4	0.19	0.18	0.13	0.20	0.15	0.20	0.32	0.37	0.35	0.40	0.25
	LL | SA	2 × 4	0.50	0.45	0.33	0.52	0.39	0.52	0.85	0.96	0.92	1.04	0.63
		3 × 4	0.39	0.35	0.25	0.39	0.30	0.39	0.64	0.73	0.68	0.79	0.49
		4 × 4	0.31	0.27	0.19	0.30	0.23	0.30	0.49	0.56	0.53	0.60	0.38
		5 × 4	0.26	0.22	0.16	0.25	0.19	0.25	0.40	0.47	0.43	0.50	0.32
		6 × 4	0.24	0.19	0.14	0.22	0.17	0.22	0.35	0.40	0.38	0.43	0.28
		7 × 4	0.21	0.17	0.12	0.19	0.14	0.19	0.30	0.35	0.33	0.37	0.24
		8 × 4	0.19	0.15	0.11	0.16	0.13	0.16	0.26	0.30	0.28	0.32	0.21

Table 11. Timings (in msec) for Algorithms LL, LL-SA and LL-SM for computing kG with $|k| = 160$ on Alpha 21164/533MHz (timings for Algorithm LL-SM denote timings per scalar multiplication for $t = 100$)