Application of Montgomery's Trick to Scalar Multiplication for Elliptic and Hyperelliptic Curves Using a Fixed Base Point

Pradeep Kumar Mishra and Palash Sarkar

Cryptology Research Group,
Applied Statistics Unit,
Indian Statistical Institute, 203 B T Road,
Kolkata-700108, INDIA

Abstract. We propose a scalar multiplication algorithm for elliptic and hyperelliptic curve cryptosystems, which uses affine arithmetic and is resistant against simple power attacks. Also, using a modification of known techniques the algorithm can be made immune against differential power attacks. The algorithm uses Montgomery's trick and a precomputed table consisting of multiples of the base point. Consequently, the algorithm is useful in a scenario where the base point is fixed, like Elgamal encryption or signature generation. Under such circumstances, for hyperelliptic curves, the algorithm compares favourably with other known algorithms over all fields. For elliptic curves, under similar circumstances, the algorithm performs better than other algorithms over prime fields. The increase in speed is due to a proper application of Montgomery's trick to efficiently perform the simultaneous inversion of several field elements.
Keywords : elliptic curves, hyperelliptic curves, scalar multipication, field inversion, explicit formulae, side-channel attacks.

1 Introduction

Elliptic curve cryptosystems (ECC) in recent years are gradually being inducted into many standards like ANSI, IEEE, NIST etc. The main advantage of these cryptosystems is that the key size is quite small in comparison to other cryptosystems like RSA, making these suitable for resource constrained devices, like smart card. Hyperelliptic curve cryptosystems (HECC) are also attractive, as the underlying field size is smaller and there are many more curves to choose from. ECC has already established itself as a popular public key cryptosystem. However, computational complexity of the HECC has till now come in the way of its commercial utilisation. Several research groups around the world have now diverted their attention to HECC to reduce its complexity and make it available for popular applications.

Both ECC and HECC are based on the discrete logarithm problem. The underlying group in ECC is provided by the set of points on the curve over a finite field on which an additive group operation is defined. On the other hand, cryptography using hyperelliptic curves is carried out in the Jacobian of such curves. The Jacobian is an additively written group and the elements of the Jacobian are called divisors. *In this paper, we will use the term point to mean both a point on an elliptic curve and a divisor in the Jacobian of a hyperelliptic curve.* The most important and computationally costly operation in (H)ECC is the scalar multiplication. Scalar multiplication is the operation of multiplying a point X with a scalar (an integer) m i.e. computing mX.

The efficiency of scalar multiplication depends to a large extent on the efficiency of addition and doubling operation of points. For elliptic curves point addition and doubling are relatively simple. Various co-ordinate systems have been proposed in the literature to reduce the complexity further. Divisors can be added in the Jacobian of hyperelliptic curves by Cantor's algorithm. However, this approach is not very efficient. One approach to improve the efficiency is to fix the genus of the curve and compute addition and doubling by explicit formulae. Such addition and doubling formulae were first described by Spallek [23] and have gone through many changes since then (see [6], [16], [24], [18], [19], [20]). For genus 2 curves an efficient set of formulae have been described by Lange in [12] and [13].

In the first paper [12], the author presents algorithms for addition and doubling, which involve the inversion of a field element along with some squarings and multiplications. In [13], algorithms are presented for addition and doubling which do not require inversion. Avoiding the inversion leads to some extra squarings and multiplications. This extra cost in terms of multiplications and squarings is not always desirable. Particularly in binary fields, where the the ratio of cost of inversion to cost of multiplication is not so high (between 3 and 8), inversion-free arithmetic is unnecessary. In prime fields, where this ratio is quite higher (≥ 30), inversion-free arithmetic seems to be more appropriate.

Side-channel attacks (SCA) were first proposed by Paul Kocher in 1996. The aim of SCA is to attack a specific implementation by measuring side-channel data like timing, power consumption traces, electro-magnetic radiation etc. One important class of these attacks, called power analysis attacks, uses the power consumption traces of the execution(s) of the implementation. Several measures have been proposed to defeat SCA in ECC.

In the current work, we present a new algorithm for computing the scalar multiplication for both elliptic and hyperelliptic curve cryptosystems. Our approach uses arithmetic with inversion and performs better than algorithms using inversion-free arithmetic in prime fields, where the cost of inversion is much higher than in binary fields. Moreover, the proposed algorithm is SCA resistant.

The efficiency of the algorithm is derived from the fact that, while computing the scalar multiplication, instead of scanning one bit at a time, several bits can be scanned from different locations in the binary representation of the multiplier and the point additions and doublings can be done simultaneously. As each of the additions and doublings involve one inversion, all these inversions can be computed simultaneously by Montgomery's trick (see Section 2.2) with only one inversion and some extra multiplications. The partial results so obtained are added by another point addition algorithm, TreeADD, which computes addition of several points in a tree structure. The partial sums at the nodes of a particular level of the tree are computed together and the involved inversions are computed simultaneously by Montgomery's trick. This yields a very efficient scalar multiplication algorithm.

Our algorithm uses a precomputed table. So, it is useful in applications where the base point is fixed, like Elgamal encryption and signature generation etc. Also, the use of Montgomery's trick requires storage of several points which increases the memory requirement. In Section 5, it can be seen that for HECC over prime fields, when the base point is fixed, the algorithm is 77% faster than DPA resistant version of Coron's dummy addition method. Over binary fields the speed enhancement is about 28.1%. The performance is lower due to the fact that inversion is cheaper over such fields. For ECC over fields of characteristic > 3, under similar assumptions, the performance of our algorithm compares favourably against all SCA-resistant algorithms. The speed up is around 10% in the best scenario (window-size $= 5$).

2 Preliminaries

We first present a brief overview of hyperelliptic curves. For details, readers can refer to [10], [15], [11] or [3]. Let K be a field and let $\overline{K}$ be the algebraic closure of K. A *hyperelliptic curve* C of genus g (≥ 1) over K is an equation of the form $C : v^2 + h(u)v = f(u)$ where $h(u)$ in $K[u]$ is a polynomial of degree at most $g, f(u)$ in $K[u]$ is a monic polynomial of degree $2g + 1$, and there are no "singular points". *Elliptic curves are hyperelliptic curves of genus 1.*

A *divisor* D is an element of the free abelian group generated by all the points of the curve C over K. Let $\mathcal{D}$ stand for the set of all divisors. The *degree* of a divisor is defined to be the sum of all integer coefficients of the points occuring in the divisor. The set $\mathcal{D}^0$ of all divisors of degree 0 forms a subgroup of $\mathcal{D}$.

The set $\mathcal{D}^0$ can be partitioned into equivalence classes of divisors, each of which contains and hence is represented by an unique special type of divisor, called *reduced* divisors. Reduced divisors have a beautiful cannonical representation by means of two polynomials $[a(u), b(u)]$ of small degree over K. This is called Mumford's representation. The reduced divisors can be effectively added using Cantor's algorithm [3]. This group of reduced divisors is called the *Jacobian* of the curve C. It is generally denoted by $J_C(K)$. The discrete logarithm problem on the Jacobian of hyperelliptic curves of lower genus ($g \leq 4$) over suitable finite fileds K, is believed to be hard. This opens the possibility of realising different cryptographic primitives over it.

2.1 Point Arithmetic in (H)ECC

Let $[i], [m]$ and $[s]$ denote the amount of time required to compute an inversion, a multiplication and a squaring respectively in the underying field. We will use the notation $\mathbf{i}$ to denote the ratio $[i]/[m]$, which represents the relative cost of an inversion compared to a multiplication. The value of **i** depends on the choice of the underlying field. For binary fields this value has been reported to be between 3 and 10 and for prime fields it is somewhere between 30 and 40 (see [5]). In prime fields the cost of a squaring is known to be somewhat less than the cost of a multiplication. *For simplicity, in the current work we assume [m] = [s].*

Elliptic Curve Arithmetic We only consider elliptic curves over prime fields. The equation of an elliptic curve over such a field is $y^2 = x^3 + ax + b$ where $a, b \in K$ and $4a^3 + 27b^2 \neq 0$. The cost of addition (ECADD) and doubling (ECDBL) algorithms for ECC in affine co-ordinates are $1[i] + 1[m] + 1[s]$ and $1[i] + 2[m] + 1[s]$ respectively.

Hyperelliptic Curve Arithmetic For addition of divisors in the Jacobian of hyperlliptic curves, use of explicit formulae has been proved to be the most efficient method. Many such formulae have been proposed in the literature by various authors. In this work, we will mostly concentrate on hyperlliptic curves of genus 2. For these curves Lange has provided a set of efficient formulae for addition and doubling in [12], [13]. The formulae proposed in [12] involve inversions in the underlying field. We will refer to these formulae as affine arithmetic. The formulae proposed in [13] do not involve inversions. We will refer to these as inversion-free arithmetic. For genus 3 and genus 4 curves affine formulae are proposed by Pelzl et al [19], [20]. Our proposed algorithm can also be used for curves of genera 2 and 3, which require 1 inversion each for divisor addition and doubling. In the current paper we will concentrate on curves of genus 2. Table 1 describes the complexity of addition and doubling formulae, proposed in [12], [13].

Table 1. Complexity of Explicit Formulae described in [12, 13]

Name/Proposed in	Cost(Add)	Cost(Double)
Lange [12]	$1[i] + 22[m] + 3[s]$	$1[i] + 22[m] + 5[s]$
Lange [13]	$40[m] + 6[s]$	$47[m] + 4[s]$

2.2 Computing Inverses Simultaneously

Our scalar multiplication algorithm derives its efficiency from the fact that inversions of several field elements can be computed simultaneously by one inversion and some extra multiplications. One well known technique for doing this is Montgomery's trick [22], [17] which works as follows. Let $x_1, \cdots, x_n$ be the elements to be inverted. The algorithm first computes $a_1 = x_1, a_2 = x_1x_2, \cdots, a_n = x_1 \cdots x_n$, by $(n - 1)$ multiplications. Then it inverts a_n. Now, x_n^{-1} is computed by multiplying a_n^{-1} by a_{n-1}. Also, $a_{n-1}^{-1} = a_n^{-1}x_n$ and x_{n-1}^{-1} is $a_{n-1}^{-1}a_{n-2}$. Similarly, it computes inverse of other elements. It is not difficult to see that the algorithm uses only $3(n-1)$ multiplications and one inversion. We will denote the cost of computing the inverses of n field elements by $\mathcal{I}(n)$. Montgomery's trick shows that we can take $\mathcal{I}(n) = 1[i] + 3(n-1)[m]$.

2.3 Side Channel Attacks

In this subsection we discuss variuos countermeasures proposed in literature to resist side-channel attacks on (H)ECC.

Countermeasures Against SPA The usual binary algorithm for scalar multiplication is not secure against side channel attacks. To resist SPA, two approaches are generally resorted to in ECC. The first one is to make the computation independent of the bit pattern representing the scalar multiplier. Several countermeasure against SPA fall in this category. The simplest one is Coron's dummy addition method, i.e. to carry out one dummy addition if the corresponding bit is 0. Other approaches are based on various addition chains and window based methods. Particularly, for ECC, the Montgomery's ladder along with x-coordinate only encapsulated add-and-double algorithm proposed by Izu and Takagi and window-based methods proposed by Moller are very efficient and secure against SPA. The second approach uses indistinguishable algorithms for point addition and doubling. Certain elliptic curves like Hess and Jacobi form elliptic curves admit such algorithms. For a detailed treatment of these methods reader can refer to [8].

There is no result specific to HECC proposed in the literature to immunize the scalar multiplication algorithm against SPA. However, Coron's dummy addition method can easily be carried over to HECC.

Countermeasures Against DPA Several remedies have been proposed for immunising ECC from DPA. We briefly mention one such method – the Joye-Tymen countermeasure [9]. Let z be a random nonzero field element. The steps are as follows.

1. Compute z^2, z^3, z^4, z^6.
2. Transform the base point $P(x, y)$ to (z^2x, z^3y).
3. Transform the curve coefficients (a, b) to $a' = z^4a, b' = z^6b$.
4. Compute scalar multiplication with the new point on the new curve.
5. Transform the result (x, y) back to the original curve using $(x, y) \to (x/z^2, y/z^3)$.

The additional cost of obtaining DPA resistance is $4[m]$ for Step 1; $2[m]$ for Step 2; $2[m]$ for Step 3 and finally $1[i] + 2[m]$ for Step 5.

Recently, Avanzi [1] has generalised these techniques for HECC. Briefly, we describe the curve randomisation countermeasure which we will use in our algorithm. Let the underlying curve C of the cryptosystem be $y^2 + h(x)y = f(x)$ where $h(x) = h_2x^2 + h_1x + h_0$ and $f(x) = x^5 + f_4x^4 + f_3x^3 + f_2x^2 + f_1x + f_0$. Let $D = [u(x), v(x)]$ be the base divisor in C and let z be a random field element.

1. Compute $z^{-1}, z^{-2}, z^{-3}, z^{-4}, z^{-5}, z^{-6}, z^{-8}$ and z^{-10}.
2. Transform $h(x)$ and $f(x)$ into $\widetilde{h}(x)$ and $\widetilde{f}(x)$ as follows:
   $\widetilde{h}(x) = z^{-1}h_2x^2 + z^{-3}h_1x + z^{-5}h_0$ and
   $\widetilde{f}(x) = x^5 + z^{-2}f_4x^4 + z^{-4}f_3x^3 + z^{-6}f_2x^2 + z^{-8}f_1x + z^{-10}f_0$.
3. Transform D to $\widetilde{D} = [\widetilde{u}(x), \widetilde{v}(x)]$, where $\widetilde{D}$ is defined as follows :
   If $\deg(u) = 2$ and $u(x) = x^2 + u_1x + u_0$ and $v(x) = v_1x + v_0$ then
   $\widetilde{u}(x) = x^2 + z^{-2}u_1x + z^{-4}u_0$ and $\widetilde{v}(x) = z^{-3}v_1x + z^{-5}v_0$.
   If $\deg(u) = 1$ and $u(x) = x + u_0$ and $v(x) = v_0$ then $\widetilde{u}(x) = x + z^{-2}u_0$ and $\widetilde{v}(x) = z^{-5}v_0$.
4. Compute scalar multiplication using the new curve and the new divisor.
5. The result is transformed back to the original curve using the inverse of Step 3 and the relevant powers of z (rather than z^{-1}).

The additional cost of attaining DPA resistance is as follows: $1[i] + 7[m]$ for Step 1; $8[m]$ for Step 2; maximum $4[m]$ for Step 3 and $8[m]$ for Step 5. If the characteristic of the field is odd, then the polynomial $h(x)$ can be taken to be zero and hence the cost of Step 2 would come down to $5[m]$.

2.4 Scalar Multiplication Methods for HECC

Many scalar multiplication algorithms immunized against SCA have been proposed for ECC. We discuss some specific methods for genus 2 HECC using Lange's formula (see Table 1) along with their costs below.

(a) The usual add and double algorithm: For an n-bit multiplier such an algorithm requires n doublings and $n/2$ additions on the average (though this computation is not SCA resistant). So cost of computing the scalar multiplication using [13] is $n \times 46[m] + (n/2) \times 51[m] = 71.5n[m]$. Using affine arithmetic [12] the cost of n doublings and $n/2$ additions is $n \times (1[i] + 22[m] + 5[s]) + (n/2) \times (1[i] + 22[m] + 3[s]) \approx ((3/2)\mathbf{i} + 39.5)n[m]$. However, this computation is not SCA resistant.

(b) To make the computation SPA resistant we can resort to Coron's countermeasure for ECC. That is we can make some dummy additions if the corresponding bit in the binary representation is 0. In this case, we have to compute n additions and n doublings. Cost of the computation in inversion-free arithmetic will be $51n[m] + 46n[m] = 97n[m]$. This computation is costlier than that of (a), but is SPA resistant. But, again in binary fields the affine arithmetic will be preferable. In binary fields, the cost will be $n(1[i] + 22[m] + 5[s]) + n(1[i] + 22[m] + 3[s]) \approx (2\mathbf{i} + 52)n[m]$. It can be made resistant against DPA using the methods described above.

(c) We can encapsulate the addition and doubling formula of affine co-ordinates to obtain a more efficient formula. Suppose, we wish to compute an addition an doubling simultaneously. In affine co-ordinates, both will involve one inversion. Instead of computing two inversions, we can compute them by Montgomery's trick with 1 inversion and 3 multiplications. So cost of one addition and doubling is $1[i] + 3[m] + 52[m] \approx (\mathbf{i} + 55)[m]$. We can now use this algorithm in Montgomery's ladder type scalar multiplication algorithm to compute the scalar multiplication. The method involves one doubling at the outset.

Amount of computations involved in computing the scalar multiplication would be $((\mathbf{i}+55)n+\mathbf{i}+27)[m]$. Again the computation will be SPA resistant. It can also be made resistant against DPA.

We produce the summary of this discussion in Table 2. In the table we have considered two specific values of **i**, 8 and 30. It is clear from the Table that, (c) is better than (b). Although, the average case complexity of (a) is better than (c), the former is not SCA resistant. Note that both (b) and (c) can be made DPA resistant (at an additional cost) by using Avanzi's countermeasure as discussed in Section 2.3.

Table 2. Complexity of different algorithms for HECC.

ALGORITHM	**i**	COMPLEXITY
(a)	30	$71.5n[m]$ (avg case)
	8	$51.5n[m]$
(b)	30	$97n[m]$
	8	$68n[m]$
(c)	30	$(84n+57)[m]$
	8	$(63n+35)[m]$

3 New Algorithm for Scalar Multiplication

Before describing the proposed algorithm, let us have a close look at the addition and doubling algorithms in affine co-ordinates.

3.1 Addition and Doubling in Affine Co-ordinates

Let us consider the addition (HCADD) and doubling (HCDBL) algorithms for HECC in [12] in the most general and frequent case. These can be divided into three parts. In part one, some multiplications and squarings of the underlying field elements are carried out. In part two, a field element, generated in part one is inverted. The inverse so obtained in part two is used in part three along with some more multiplications and squarings of field elements. The output of part three provides the required divisor. See Figure 1.

Let us name the modules of these algorithms as $\mathbf{A_1}, \mathbf{A_2}, \mathbf{A_3}$ (parts of addition algorithms) and $\mathbf{D_1}, \mathbf{D_2}, \mathbf{D_3}$ (parts of doubling algorithms). In each of $\mathbf{A_2}$ and $\mathbf{D_2}$, we compute only one inverse. Let the number of multiplications and squarings required in $\mathbf{A_1}$, $\mathbf{A_3}$, $\mathbf{D_1}$ and $\mathbf{D_3}$ be $\mathbf{a_1}$, $\mathbf{a_3}$, $\mathbf{d_1}$ and $\mathbf{d_3}$ respectively. We will use the notation $\mathbf{a}$ for $\mathbf{a_1}+\mathbf{a_3}$ and $\mathbf{d}$ for $\mathbf{d_1}+\mathbf{d_3}$. By $\mathbf{A_1}(D_1, D_2)$, we will mean the field element α created in module $\mathbf{A_1}$ of the addition algorithm and which is

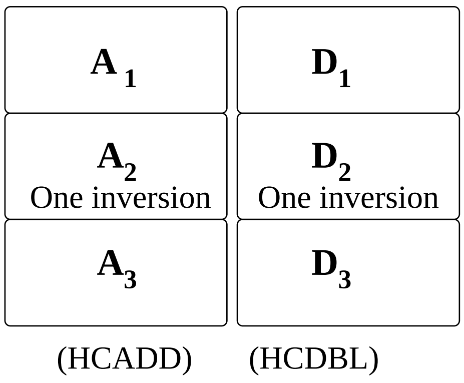

Fig. 1. HCADD and HCDBL algorithms proposed in [12]

inverted in $\mathbf{A_2}$. Similarly, by $\mathbf{D_1}(D_1)$, we will mean the field element β created in module $\mathbf{D_1}$ of the doubling algorithm and which is inverted in module $\mathbf{D_2}$. By $\mathbf{A_3}(D_1, D_2, \alpha^{-1})$ (resp. $\mathbf{D_3}(D_1, \beta^{-1})$) we mean the divisor produced by the module $\mathbf{A_3}$ (resp. $\mathbf{D_3}$) as sum of D_1 and D_2 (resp. D_1).

The same can be said about addition and doubling algorithms for ECC in affine co-ordinates. In ECC, the values of $\mathbf{a_1}, \mathbf{a_3}, \mathbf{a}$ etc will be much smaller.

3.2 The Proposed Algorithm

Let $w \geq 2$ be a positive integer. We express m in the base 2^w. Let $m = c_0 + c_1 2^w + \cdots + c_{t-1} 2^{w(t-1)}$, where each $c_j \in \{0, \ldots, 2^w - 1\}$. Then $mD = c_0 D + c_1 2^w D + \cdots + c_{t-1} 2^{w(t-1)} D$. For all $j, 0 \leq j \leq t-1$ we precompute $2^{jw} D$ and store it in a table $T[\,]$. Thus $T[j] = 2^{jw} D$ for $0 \leq j \leq t-1$. This table is used to simultaneously compute $c_0 D, c_1 2^w D, \cdots, c_{t-1} 2^{w(t-1)} D$ using the right-to-left binary method. (A similar algorithm can also be developed using the left-to-right binary method.) Finally we add them to obtain mD.

Let the n-bit representation of m be $m_{n-1} \ldots m_0$. Note that $t = \lceil (n/w) \rceil$. We express c_j in binary, i.e., we write $c_j = c_j^0 + c_j^1 2 + \cdots + c_j^{w-1} 2^{w-1}$, where $c_j^i = m_{wj+i}$. We require $2t+1$ point type variables $R_0, \ldots, R_{t-1}$ and $Q_0, Q_1, \ldots, Q_t$. The variable R_j is initialized to $2^{jw} D$. Starting from the least significant bit of c_i, we scan a bit in each iteration. If the scanned bit is 1, then we add R_j to Q_{j+1}; if the scanned bit is 0, then we add R_j to Q_0; in either case we double R_j. After w iterations, Q_{j+1} is $c_j 2^{wj} D$. For $0 \leq j \leq t-1$, we compute all the expressions $c_j 2^{wj} D$ simultaneously. Each of the additions and doublings in each iteration will involve one inversion. While doing these additions and doublings, we carry out all the inversions simultaneously by Montgomery's trick. This yields an efficient algorithm for scalar multiplication. Our algorithm calls a routine INVERT, which simultaneously inverts a number of field elements using Montgomery's trick. Recall that by $\mathcal{I}(n)$ we denote the cost of inversion of n elements using INVERT and $\mathcal{I}(n) = 1[i] + 3(n-1)[m]$.

Algorithm EFF-SCLR-MULT
Input: $m, t, c_0, c_1, \cdots, c_{t-1}, D$.
Output: mD.

```
1.  For $j = 0$ to $t-1$ $\{R_j = T[j]$;
2.      If $c_j^0 = 0$ then $b = 0, t_b = j+1$
        else $b = j+1, t_b = 0$, $Q_b = R_j$; $\}$
2.  For $j = 0$ to $t-1$ let $\beta_j = \mathbf{D_1}(R_j)$;
3.  Let $(\beta_0^{-1}, \cdots, \beta_{t-1}^{-1}) = \mathrm{INVERT}(\beta_0, \cdots, \beta_{t-1})$
4.  For $j = 0$ to $t-1$ let $R_j = \mathbf{D_3}(R_j, \beta_j^{-1})$;
5.  For $i = 1$ to $w-2$
6.      For $j = 0$ to $t-1$ $\{$ $\alpha_j = \mathbf{A_1}(R_j, Q_{j+1})$; $\beta_j = \mathbf{D_1}(R_j)$;$\}$
9.      Let $(\alpha_0^{-1}, \cdots, \alpha_{t-1}^{-1}, \beta_0^{-1}, \cdots, \beta_{t-1}^{-1},) = \mathrm{INVERT}(\alpha_0 \cdots \alpha_{t-1}, \beta_0, \cdots, \beta_{t-1})$
10.     For $j = 0$ to $t-1$ $\{Q_{c_j^i(j+1)} = \mathbf{A_3}(R_j, Q_{j+1}, \alpha_j^{-1})$; $R_j = \mathbf{D_3}(R_j, \beta_j^{-1})$;$\}$
13. end do.
14. For $j = 0$ to $t-1$ let $\alpha_j = \mathbf{A_1}(R_j, Q_{j+1})$;
15. Let $(\alpha_0^{-1}, \cdots, \alpha_{t-1}^{-1}) = \mathrm{INVERT}(\alpha_0, \cdots, \alpha_{t-1})$
16. For $j = 0$ to $t-1$ let $Q_{c_j^{w-1}(j+1)} = \mathbf{A_3}(R_j, Q_{j+1}, \alpha_j^{-1})$;
17. Let RES = TreeADD$(Q_1, \cdots, Q_t)$
18. Return (RES)
```

Proposition 1. *The cost of the above algorithm is* $[t(w-1)(\mathbf{a}+\mathbf{d}) + t\mathbf{a}][m] + (w-1)\mathcal{I}(2t) + \mathcal{I}(t) +$ *cost(TreeADD), where cost(TreeADD) is the cost of the TreeADD algorithm invoked by the algorithm.*

The algorithm TreeADD adds a number of points efficiently. It uses a tree structure for computation. Suppose $D_0, D_1, \cdots, D_{k-1}$ are the input points. For simplicity, assume $k = 2^r$. Imagine a tree of depth r with the input points at the leaf nodes. We pairwise add the points at the nodes with a common parent and put the sum at the parent node of each pair. There are 2^{r-1} nodes at level $r-1$ and to get the points at these nodes we have to perform 2^{r-1} additions. Note that, each of these additions needs one inversion. Instead of computing 2^{r-1} inversions separately, we can compute them with 1 inversion and $(3 \times 2^{r-1} - 1)$ multiplications using Montgomery's algorithm. This process is carried out at each level to the root. The root then contains the sum of all the points.

Algorithm TreeADD
Input: $D_0, \cdots, D_{2^k-1}$
Output: $D_0 + D_1 + \cdots + D_{2^k-1}$

```
1. For $i = 0$ to $2^k - 1$ let $D_i^{(0)} = D_i$.
2. For $j = 1$ to $k$
3.     let $(D_0^{(j)}, D_1^{(j)}, \cdots, D_{2^{k-j}-1}^{(j)}) = \mathrm{ADD}(D_0^{(j-1)}, D_1^{(j-1)}, \cdots, D_{2^{k-j+1}-1}^{(j-1)})$
4. Return $(D_0^{(k)})$
```

TreeADD invokes the algorithm ADD, which takes as input $2k$ points, $D_0, D_1, \cdots, D_{2k-1}$ and returns $D_0 + D_1, D_2 + D_3, \cdots, D_{2k-2} + D_{2k-1}$. ADD computes k additions at one invocation. Hence, the inversions at $\mathbf{A_2}$ step of all these additions can be done simultaneously using the Montgomery's algorithm.

```
Algorithm ADD
Input: $D_0, \cdots, D_{2k-1}$.
Output: $D_0 + D_1, \cdots, D_{2k-2} + D_{2k-1}$.
1. For $i = 0$ to $k-1$, let $\alpha_i = \mathbf{A_1}(D_{2i}, D_{2i+1})$.
2. Let $(\alpha_0^{-1}, \alpha_1^{-1}, \cdots, \alpha_{k-1}^{-1}) = \text{INVERT}(\alpha_0, \alpha_1, \cdots, \alpha_{k-1})$.
3. For $i = 0$ to $k-1$ let $E_i = \mathbf{A_3}(D_{2i}, D_{2i+1}, \alpha_i^{-1})$.
4. Return $(E_0, E_1, \cdots, E_{k-1})$.
```

Proposition 2. *ADD takes $2^r\mathbf{a}[m] + \mathcal{I}(2^r)$ computations to compute the $k = 2^r$ sums of $2k = 2^{r+1}$ input points.*

With $\mathcal{I}(n) = 1[i] + 3(n-1)[m]$ (see Subsection 2.2), the cost of ADD becomes, $((2^r\mathbf{a} + 3(2^r - 1)[m] + 1[i]$.

Now we can compute the complexity of the algorithm TreeADD. TreeADD repeatedly calls the algorithm ADD, first with 2^r points, then with 2^{r-1} points and so on. Let the cost of ADD with k input points be $[kA]$. Then, cost of TreeADD with 2^r points is $[2^rA] + [2^{r-1}A] + \cdots + [1A]$. By Proposition 2, $[2^iA] = 2^{i-1}\mathbf{a}[m] + \mathcal{I}(2^{i-1})$. Hence computational cost of TreeADD is $\sum_{i=1}^{r-1}[2^iA]$ $= \sum_{i=1}^{r-1} 2^{i-1}\mathbf{a}[m] + \mathcal{I}(2^{i-1}) = ((2^r - 1)\mathbf{a}[m] + \sum_{i=1}^{r-1}\mathcal{I}(2^{i-1})$. With $\mathcal{I}(n) = 1[i] + 3(n-1)[m]$, we have:

Proposition 3. *TreeADD takes $(2^r - 1)\mathbf{a}[m] + \sum_{i=1}^{r-1}\mathcal{I}(2^i) = [(2^r - 1)\mathbf{a} + 3 \times 2^r - 3r - 3][m] + r[i]$ computations to compute the sum of 2^r input points.*

We now compute the complexity of the algorithm EFF-SCLR-MULT. In the Steps 2–4 we double t points, inverting t elements by INVERT. In each iteration of the loop in Steps 5–13, we add t points and double t points. So in each iteration we invoke INVERT with $2t$ field elements. In the Steps 14–16 we add t points, inverting t elements by INVERT. Finally in Step 17 the TreeADD algorithm is invoked.

Proposition 4. *EFF-SCLR-MULT takes $(w+r)[i] + [2^r(w-1)(\mathbf{a}+\mathbf{d}+6) - 3w + (2^r - 1)\mathbf{a} + 3 \times 2^r - 3r - 3][m]$ computations to compute the scalar multiplication mD where, m is an n-bit integer, w is the window size and $t = \lceil n/w \rceil = 2^r$.*

The algorithm uses a table which must be precomputed. Online computation will be very costly. The table will store t points. An elliptic curve point is an ordered pair of field elements. Each field element is of 160 bits. So a point occupies 320 bits of memory. Similarly, a divisor of a hyperelliptic curve of genus 2 is a 4-tuple of field elements, where each field element is of around 80 bits. So, a divisor also occupies almost the same amount of memory. The algorithm needs to store t points means, it requires about $320t$ bits of storage.

4 Resistance Against SCA

In this section we discuss the resistance of our algorithm to side-channel attacks and the cost involved in achieving such resistance.

4.1 Resistance Against SPA

Algorithm EFF-SCLR-MULT is resistant against simple power attacks, the reason being the following. At each iteration in steps 2 to 10, we are scanning t bits from the binary representation of m and computing some additions and doublings. The numbers of additions and doublings are fixed and independent of the actual bit pattern scanned. Similarly in steps 12 to 17, the same number of additions are being computed irrespective of the actual bits scanned. Hence we conclude that the computations are resistant against SPA.

4.2 Resistance Against DPA

We discuss a method for making the algorithm resistant against DPA. Recall that we use a look-up table $T[\,]$ of t points. The steps for the counter-measure for both ECC and HECC are as follows.

1. Choose a random nonzero field element z.
2. Compute the relevant powers of z. (see Subsection 2.3
3. Transform the curve parameters.
4. Transform each of the t points of $T[\,]$.
5. Perform scalar multiplication using Algorithm EFF-SCLR-MULT.
6. Transform the result back to the original curve.

The specific transformations are different for ECC and HECC. For ECC we use Joye-Tymen transformation while for HECC we use Avanzi's transformation. Accordingly the costs are also different. From the discussion in Section 2.3 we get the following costs.

ECC	: $1[i] + 4[s] + (4 + 2t)[m] \approx 1[i] + (8 + 2t)[m]$ assuming $[m] = [s]$.
HECC	: $1[i] + (23 + 4t)[m]$ $(1[i] + (20 + 4t)[m]$ for odd characteristic).

5 Results and Comparison

In this section, we present some results of our algorithm and compare it with other algorithms. We do it separately for HECC of genus 2 and ECC.

5.1 HECC

We compare the performance of Algorithm EFF-SCLR-MULT to the algorithms (a), (b) and (c) described in Section 2.4. Table 3 displays these calculations. In Table 3, columns (a)-(c) refer to the algorithms listed in rows (a)-(c) of Table 2. Cost of DPA resistance has been added to the cost of (b) and (c). Column (d) stands for our algorithm. The column n stands for the bit size of the multiplier m. The parameter w stands for the window size. The complexity of the algorithms (a)-(c) do not vary with w as they are not window-based. The parameter t stands for the size of the look up table and is equal to the number of points required

Table 3. HECC: Comparison of the number of multiplications for different values of the number of bits n required to represent the scalar multiplier m.

Parameters			$\mathbf{i} = 8$				$\mathbf{i} = 30$			
n	w	t	(a)	(b)	(c)	(d)	(a)	(b)	(c)	(d)
160	5	32	8240	10896	10129	8501	11440	15539	13665	8743
160	10	16	8240	10896	10129	8937	11440	15539	13665	9267
160	20	8	8240	10896	10129	9190	11440	15539	13665	9718
160	40	4	8240	10896	10129	9389	11440	15539	13665	10335
160	80	2	8240	10896	10129	9690	11440	15539	13665	11440

to be stored. One can easily observe that, for certain window sizes over prime fields, the proposed algorithm (d) is better than even average case complexity of double-and-add (column (a)), which is not SPA resistant. In the best scenario, the new algorithm achieves a speed-up of around 77 percent if $\mathbf{i} = 30$ over usual SPA resistant double and always add approach (b). Over binary fields, the performance enhancement is lower, only 28.1%, due to the fact that $\mathbf{i}$ is lower. For other values of $\mathbf{i}$ similar comparisons can be made. We have observed that as $\mathbf{i}$ increases from 30 to 40, the cost of (b) goes up by around 10%, whereas the cost of our algorithm goes up by about 1% only.

5.2 Efficiency for elliptic curves

The algorithm EFF-SCLR-MULT can also be used for ECC over prime fields. This is because ECADD and ECDBL algorithms in affine co-ordinates have a structure similar to that of Figure 1. For 160 bits scalar multiplier, the amount of computation required by the algorithm to compute the scalar multiplication is shown in Table 4. To compare the performance of the algorithm with other

Table 4. ECC: Number of multiplications required by EFF-SCLR-MULT assuming $\mathbf{i} = 30$.

n	w	t	Complexity
160	5	32	2222[m]
160	10	16	2410[m]
160	20	8	2693[m]
160	40	4	3226[m]

algorithms proposed in the literature we show Table 5 which is taken from [8]. It shows efficiency of some other SCA resistant methods. Note that the table does not exactly matches with the table presented in [8], as we have not taken additions into account and have taken $[s] = [m]$. Table 5 shows that for efficient and secure computation of scalar multiplication, Improved Moller's method with

Table 5. ECC: Number of multiplications required by previous algorithms under the assumptions $[\mathbf{i}] = 30$ and $[m] = [s]$.

Method	(160-bit ECC)
Coron's dummy addition	3375[m]
Coron's dummy addition with $a = -3$	3057[m]
Improved Moller with $w = 2$	3220[m]
Improved Moller with $w = 2$ and $a = -3$	3064[m]
Improved Moller with $w = 3$	2543[m]
Improved Moller with $w = 3$ and $a = -3$	2429[m]
Improved Izu-Takagi	2758[m]
Improved Izu-Takagi with $a = -3$	2439[m]

window size 3 and $a = -3$ is the best. It takes $2429[m]$ computations. Our algorithm takes $2222[m]$ in the best situation, when the window size is 5, nearly 10% performance enhancement.

5.3 Memory Requirement

The parameter t in Table 3 determines the size of the look-up table. It is equal to the number of points to be stored in the look-up table. It is natural that the efficiency of the algorithm goes up as we invert more and more elements together. If a window size of 5 is chosen then the table size will be 32. In hyperelliptic curve cryptosystem with reasonable security, a point size is around 320 bits. So, the table will occupy around 1.2 kilobyte of memory.

Additionally Algorithm EFF-SCLR-MULT requires some more intermediate points and field elements. The calculation for these is as follows.

- $2t+1$ intermediate points including one dummy point (Q_0) for Coron's trick.
- $2t$ field elements $(\alpha_0, \ldots, \alpha_{t-1})$ and $(\beta_0, \ldots, \beta_{t-1})$ for applying Montgomery's trick.

This memory requirement might be costly for memory constrained applications (as in smart card applications). In such situations our algorithm cannot be used. However, we note that in situations where the amount of memory is not a constraint (as in desktops), our algorithm provides a speed-up over the known algorithms (for fixed base point).

References

1. Roberto M. Avanzi. Countermeasures Against Differential Power Analysis for Hyperelliptic Curve Cryptosystems. In *CHES 2003*, to appear.
2. E. Brior and M. Joye. Weierstrass Elliptic Curves and Side-Channel Attacks. In *PKC 2002*, LNCS 2274, pages 335-345, Springer-Verlag,2002.
3. D. G. Cantor. Computing in the Jacobian of a Hyperelliptic curve. In *Mathematics of Computation*, volume 48, pages 95-101, 1987.

4. J.-S. Coron. Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems. In *CHES 1999*, pages 292-302.
5. K. Fong and D. Hankerson and J. López and A. Menezes. Field inversion and point halving revisited. Technical Report, CORR 2003-18, Department of Combinatorics and Optimization, University of Waterloo, Canada, 2003.
6. R. Harley Fast Arithmetic on Genus 2 Curves. *Avaiable at http://cristal.inria.fr/ harley/hyper,2000.*
7. T. Izu and T. Takagi. A Fast Parallel Elliptic Curve Multiplication Resistant against Side-Channel Attacks Technical Report CORR 2002-03, University of Waterloo,2002. Available at http://www.cacr.math.uwaterloo.ca
8. T. Izu, B. Moller and T. Takagi. Improved Elliptic Curve Multiplication Methods Resistant Against Side Channel Attacks. Proceedings of Indocrypt 2002, LNCS 2551, pp 296-313, Springer-Verlag.
9. M. Joye and C. Tymen Protection against differential attacks for elliptic curve cryptography. CHES 2001, LNCS 2162, pp 402-410, Springer-Verlag.
10. N. Koblitz. Hyperelliptic Cryptosystems. In *Journal of Cryptology*, 1: pages 139–150, 1989.
11. N. Koblitz. *Algebraic Aspects of Cryptology*, Algorithms and Computation in Mathematics. Springer Verlag, 1998.
12. T. Lange. Efficient Arithmetic on Genus 2 Curves over Finite Fields via Explicit Formulae. Cryptology ePrint Archive, Report 2002/121, 2002. *http://eprint.iacr.org/* .
13. T. Lange. Inversion-free Arithmetic on Genus 2 Hyperelliptic Curves. Cryptology ePrint Archive, Report 2002/147, 2002. *http://eprint.iacr.org/* .
14. A. J. Menezes, P. C. van Oorschot and S. A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1997.
15. A. Menezes, Y. Wu, R. Zuccherato. An Elementary Introduction to Hyperelliptic Curve. Technical Report CORR 96-19, University of Waterloo(1996),Canada. Available at http://www.cacr.math.uwaterloo.ca
16. Y. Miyamoto, H. Doi, K. Matsuo, J. Chao and S. Tsujii. A fast addition algorithm for genus 2 hyperelliptic curves. In *Proc of SCIS2002, IEICE, Japan*,pp 497-502,2002, in Japanese.
17. P. Montgomery. Speeding the Pollard and Elliptic Curve Methods for Factorisation. In *Math. Comp.*, vol 48, pp 243-264, 1987.
18. K. Nagao. Improving Group Law Algorithms for Jacobians of Hyperelliptic Curves. In W. Bosma, editor, *ANT IV*, LNCS 1838, Berlin 2000, Springer-Verlag.
19. J. Pelzl, T. Wollinger, J. Guajardo and C. Paar. Hyperelliptic Curve Cryptosystems: Closing the Performance Gap to Elliptic Curves. Cryptology ePrint Archive, Report 2003/26, 2003. *http://eprint.iacr.org/* .
20. J. Pelzl, T. Wollinger, J. Guajardo and C. Paar. Low Cost Security: Explicit Formulae for Genus 4 Hyperelliptic Curves . Cryptology ePrint Archive, Report 2003/97, 2003. *http://eprint.iacr.org/* .
21. K. Okeya and K. Sakurai. Efficient Elliptic Curve Cryptosystems from a Scalar Multiplication Algorithm with Recovery of the y-coordinate on a Montgomery form Elliptic Curve. In *CHES 2001*, LNCS 2162, pp 126-141, Springer-Verlag 2001.
22. H. Shacham, D. Boneh. Improving SSL Handshake Performance via Batching. In *CT-RSA*, LNCS 2020, pp 28-43 Springer-Varlag, 2001.
23. A. M. Spallek. Kurven vom Geschletch 2 und irhe Anwendung in Public-Key-Kryptosystemen. Ph D Thesis, Universitat Gesamthochschule, Essen, 1994.
24. M. Takahashi. Improving Harley Algorithms for Jacobians of Genus 2 Hyperelliptic Curves. In *Proc of SCIS 2002*, ICICE, Japan, 2002, in Japanese.