Efficient and Secure Scalar Multiplication in the Jacobian of Genus 2 Hyperelliptic Curves

Pradeep Kumar Mishra, Palash Sarkar, Rana Barua
Cryptology Research Group,
Indian Statistical Institute,
203, B T Road,
Kolkata-700108

Abstract

A major bottleneck in the implementation of hyperelliptic curve cryptgraphy is the problem of performing efficient arithmetic in the Jacobians of hyperelliptic curves. One approach to the problem, as suggested by several authors, is to use explicit formula to compute the addition and doubling of divisors. Some versions of such formulae come with inversion operation of field elements and some without them. It is generally believed that inversion free arithmetic leads to more efficient scalar multiplication algorithm. In this paper, we develop scalar multiplication algorithms in which it is more efficient to use formulae which involves inversion of field elements rather than inversion free arithmetic. An important constituent of our scalar multiplication algorithm is an efficient algorithm to *simultaneously* obtain the inverses of a list of underlying field elements. Our scalar multiplication algorithm compares favourably with the usual scalar multiplication using inversion free arithmetic. Moreover, our algorithm lends itself well to parallelism and provides protection against side-channel attacks.

Keywords : Hyperelliptic curves, public key cryptosystems, scalar multipication, Jacobian, field element inversion, explicit formulae for divisor addition, doubling, side-channel attacks .

1 Introduction

Hyperelliptic curve cryptosystem (HECC) was first proposed by Koblitz in [5]. The main advantage of HECC is that the underlying field size is quite small in comparison to other cryptosystems. That leads to smaller key length, making it suitable for devices with limited resources, like smart card. However, computational complexity of the HECC has always 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. In particular, a European Union sponsored research project AREHCC [1] has linked seven academic and industrial organisations to carry out research on this problem.

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. The most important and computationally costly operation in HECC is the scalar multiplication. Scalar multiplication is the operation of multiplying a divisor D with a scalar (an integer) m i.e. computing mD.

The efficiency of scalar multiplication depends upon the efficiency of addition and doubling operation of divisor. Divisors can be added in the Jacobian of the hyperelliptic curve by Cantor's algorithm. However, this approach is not very efficient. One approach to improving 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 [19] and have gone through many changes since then (see [4], [11], [20]). For genus 2 curves the most recent formulae have been described by Lange in [9] and [10].

In the first paper [9], the author presents an algorithm for addition which involves the inversion of a field element along with some squarings and multiplications. A similar algorithm is described for doubling. In [10], algorithms are presented for addition and doubling which do not require the inversion of field elements. Avoiding the inversion leads to some extra squarings and multiplications. However, the total cost of such squarings and multiplications is less than the cost of one inversion. Hence for usual scalar multiplication algorithm, the use of inversion free arithmetic leads to a faster algorithm.

Our approach in this paper is to investigate the efficacy of inversion free arithmetic in the context of scalar multiplication. The question that we ask is the following: Does there exist scalar multiplication algorithms, where it is more efficient to use the formulae of [9] (involving inversion) rather than the inversion free arithmetic presented in [10].

Our first observation is the following. Suppose D_1, D_2, D_3 and D_4 are divisors and we have to compute $D = D_1 + D_2$ and $D' = D_3 + D_4$. If we directly use the formulae of [9], then this involves the inversion of two underlying field elements α_1 and α_2. However, this can actually be done using only one inversion in the following manner: Compute $\beta = (\alpha_1\alpha_2)^{-1}$ and $\alpha_1^{-1} = \beta\alpha_2$ and $\alpha_2^{-1} = \beta\alpha_1$. The total cost is one inversion and three multiplications. We use this simple observation to combine the inverse operation in the computation of D and D'. It is clear that the same technique can also be used to combine the inverse operations in two doublings or an addition and a doubling. Based on this idea and the encapsulation of doubling and addition algorithms together for elliptic curve scalar multiplication by Izu and Takagi [17], we develop a encapsulated double-and-add algorithm for scalar multiplication for hyperelliptic curve scalar multiplication. This algorithm is more efficient than usual scalar multiplication algorithm using inversion free arithmetic. Moreover, uniformizing the computation in each step ensures resistance against side-channel attacks.

The next step is to further develop this idea. We first obtain an algorithm for the *simultaneous* inversion of more than one field element. If the number of elements is 2^r $(r > 0)$, then we show that inverse of all the elements can be obtained using one inversion and $(r+1)2^r - 1$ multiplications. We use this algorithm as a subroutine to develop an algorithm for scalar multiplication. The algorithm uses a look-up table and is resistant against simple power analysis attacks. Further, the algorithm is more efficient than scalar multiplication using inversion free arithmetic and our simple encapsulated double-and-add algorithm.

The rest of the paper is organised as follows. In the next section, we briefly describe hyperelliptic curves. In section 3, we describe an encapsulated add and double algorithm, analogous to the ones proposed for elliptic curves [17]. Section 4 deals with efficient simultaneous inversion of several field elements. In section 5, we use the results of section 4 to develop a new algorithm for scalar multiplication. In section 6, we compare performance of the new algorithms with that of the usual scalar multiplication using inversion free arithmetic. Finally section 7 concludes the paper.

2 Preliminaries of Hyperelliptic Curves

In this section, we give a brief overview of hyperelliptic curves. For details, readers can refer to [5], [12], [6] or [3]. Let K be a field and let $\overline{K}$ be the algebraic closure of K. A *hyperelliptic curve* C of genus g over $K (g > 1)$ 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 solutions (u,v) in $\overline{K} \times \overline{K}$, which simultaneously satisfy the equations

$$\left.\begin{array}{lcl} v^2 + h(u)v & = & f(u); \\ 2v + h(u) & = & 0; \\ h'(u)v - f'(u) & = & 0. \end{array}\right\} \tag{1}$$

A *singular point* on C is a solution $(u,v) \in \overline{K} \times \overline{K}$ which simultaneously satisfies the equations in (1). Thus, a hyperelliptic curve, by definition, does not have any singular points. Elliptic curves are hyperelliptic curves of genus 1.

A *divisor* D is a formal sum, $D = \sum_{P \in C} m_P P$, where $m_P \in \mathcal{Z}$, $\mathcal{Z}$ is the set of integers and only finitely many of m_P's are non-zero. The *degree* of D is the integer $\sum_{P \in C} m_P$ and *order* of D at P is, $ord_P(D) = m_P$. The *support* of D is the set of all points P for which m_P is non-zero. Let $\mathcal{D}$ stand for the set of all divisors. For $D_1 = \sum_{P \in C} m_P P$ and $D_2 = \sum_{P \in C} n_P P$ in $\mathcal{D}$, we define,

$$D_1 + D_2 = \sum_{P \in C} (m_P + n_P) P$$

It is easy to see that $(\mathcal{D}, +)$ is a group. Also, the set $\mathcal{D}^0$ of all divisors of degree 0 forms a subgroup of $\mathcal{D}$.

For every rational function R, defined on the curve C, we can associate one divisor, called divisor of R. A divisor is called a *pricipal divisor* if it is the divisor of a rational function. The set $\mathcal{P}$ of all principal divisor is a subgroup of $\mathcal{D}^0$. The quotient group $\mathcal{D}^0/\mathcal{P}$ is called the *Jacobian* of the curve C. It is seen that every element of the Jacobian can be represented by a divisor of the form $\sum_{i=0}^{g} a_i P_i$ with $a_i \in \{0,1\}$. Such divisors are called *reduced divisors.*

Reduced divisors have a beautiful cannonical representation by means of two polynomials $[a(u), b(u)]$, where $a(u)$ has degree $\leq g$ and degree of $b(u)$ is less than that of $a(u)$. This is called Mumford's representation. The reduced divisors can be added using Cantor's algorithm [3].

3 A Simple HCADDDBL Algorithm

In HECC, the underlying group, on which a ElGamal-type cryptosystem is constructed, is the divisor class group $J_C(F)$ of a hyperelliptic curve C over the field F. The DLP on this group can be stated as : Given any two divisors D_1 and $D_2 \in J_C(F)$ find a λ such that $D_1 = \lambda D_2$, if such a λ exists. Given a divisor D and an integer m, computing $mD = \sum_{1 \leq i \leq m} D$ is called the operation of scalar multiplication. It is generally computed by left-to-right or right-to-left binary algorithm. We state the algorithms below. In the description, by HCADD we mean an algorithm which adds two input divisors and HCDBL is an algorithm to compute double of a givensimple power divisor.

Let $m = m_{k-1}2^{k-1} + \cdots + m_1 2 + m_0 2^0$ be the binary representation of m. We can compute mD using one of the following algorithms:

Algorithm DBL-AND-ADD-L (Left-to-right binary method)	**Algorithm DBL-AND-ADD-R (Right-to-left binary method)**
Input : $D, m(m_{k-1}, \cdots m_1, m_0)$	*Input :* $D, m(m_{k-1}, \cdots m_1, m_0)$
Output : mD.	*Output :* mD.
1. $E = m_{k-1}D$	1. $E_0 = D, E_1 = 0$
2. for $i = k-2$ down to 0	2. for $i = 0$ to $k-1$
3. $\quad$ E = HCDBL(E)	3. $\quad$ if $m_i = 0$
4. $\quad$ if $m_i = 1$	4. $\quad\quad$ E_1 = HCADD(E_0, E_1)
5. $\quad\quad$ E = HCADD(E, D)	5. $\quad$ E_0 = HCDBL(E_0)
6. return E	6. return(E_1)

Addition and doubling can be carried out by several alternative algorithms. Let [m] stand for the time required to compute one multiplication in the underlying field. Let [i] and [s] denote those for a field inversion and squaring repectively. For curves of genus 2, the complexity of add and double operations when carried out by explicit formula as reported by Lange [9] are $1[i]+23[m]+2[s]$ (for HCADD) and $1[i] + 22[m] + 5[s]$ (for HCDBL) respectively in the most general and frequent case. Recently Lange in [10] has proposed inversion-free arithmetic in the Jacobian of hyperelliptic curve of genus 2. The author reports that by introducing an extra variable in the structure of a divisor the inversions can be avoided and cost of one addtion will be $47[m] + 4[s]$ and that of a doubling will be $40[m] + 6[s]$.

In this section, we propose an algorithm HCADDDBL(D_1, D_2), which simultaneously computes $D_1 + D_2$ and $2D_1$. This algorithm is used to obtain an algorithm for scalar multiplication. It turns out that the scalar multiplication algorithm is resistant to side-channel attacks and is more efficient than usual scalar multiplication using inversion free arithmetic. We note that simultaneous addition and doubling algorithm was first proposed by Izu and Takagi [17] , [18] for elliptic curve scalar multiplication. We first present the scalar multiplication algorithm.

Algorithm ENC-DBL-AND-ADD

Input: $D, m(m_{k-1} = 1, m_{k-2}, \cdots m_1, m_0)$

Output: $m * D$.

1. $E[0] = D, E[1] = \text{HCDBL}(D)$
2. for $i = k - 2$ down to 0
3. $\quad (E[m_i \oplus 1], E[m_i]) = \text{HCADDDBL}(E[m_i], E[m_i \oplus 1])$
4. return E[0]

In each iteration of the above algorithm, irrespective of the value of the corresponding m_i the same algorithm HCADDDBL is called. Thus the power consumption trace in each iteration is independent of the bit m_i and hence the algorithm is resistant to side-channel attacks.

3.1 The Algorithm HCADDDBL

The explicit formulae to add and double divisors in HECC has gone through a long process of evolution. Spallek [19] was first to conceive the idea and devised formulae for genus 2 curves over fields of odd characteristic. Harley [4], Miyamto, Doi, Matsuo, Chao and Tsuji [11] and Takahasi [11] have gradually carried over the idea to different underlying fields and have added more speed culminating in Lange ([9], [10]), which, present the fastest versions HCADD and HCDBL algorithms for genus 2 curves. Several authors are trying to generalise the idea to higher genus (e.g. Nagao [14], Pelzl, Paar et al [15] for genus 3). In the present work we will only concentrate on curves of genus 2.

Let us consider the addition and doubling algorithms in [9] 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 Fig 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). Note that in $\mathbf{A_1}$ and $\mathbf{A_3}$ we need to compute a total of 23 multiplications and 2 squarings. In $\mathbf{D_1}$ and $\mathbf{D_3}$ we compute a total of 22 multiplications and 5 squarings. In each of $\mathbf{A_2}$ and $\mathbf{D_2}$, we compute only one inverse. We observe that,

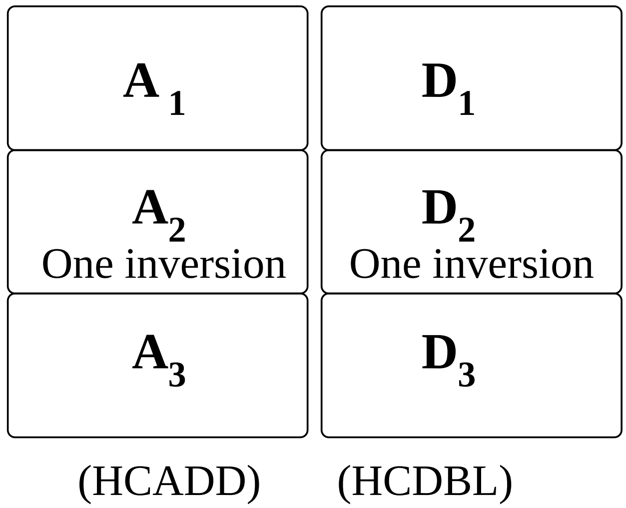

Figure 1: HCADD and HCDBL algorithms proposed in [9]

Lemma 3.1 *Inversion of 2 field elements can be done by one inversion and 3 multiplications only.*

Proof : Let α and β be the two field elements to be inverted. Compute $\gamma = \alpha \times \beta$ and invert γ. Then compute $\alpha^{-1} = \gamma^{-1} \times \beta$ and $\beta^{-1} = \gamma^{-1} \times \alpha$. ■

Based on this simple observation we describe the following simultaneous add and double algorithm. We will use the following convention. Suppose D_1, D_2 and D are divisors and we are required to compute $D_1 + D_2$ and $2D$. By $\alpha = \mathbf{A_1}(D_1, D_2)$ we denote the field element α that is to be inverted in part $\mathbf{A_2}$ and by $E = \mathbf{A_3}(D_1, D_2)$, we denote the result $E = D_1 + D_2$. Similarly, by $\beta = \mathbf{D_1}(D)$ we denote the field element β which is to be inverted in part $\mathbf{D_2}$ and $F = \mathbf{D_3}(D)$ denotes the result $F = 2D$.

Algorithm HCADDDBL

Input: D_1 and D_2.

Output: $D_1 + D_2$ and $2D_1$.

[$\mathbf{AD_1}$]: Let $\alpha = \mathbf{A_1}(D_1, D_2)$.

[$\mathbf{AD_2}$]: Let $\beta = \mathbf{D_1}(D_1)$.

[$\mathbf{AD_3}$]: Compute α^{-1} and β^{-1} using the method of lemma 3.1.

[$\mathbf{AD_4}$]: Let $E = \mathbf{A_3}(D_1, D_2)$.

[$\mathbf{AD_5}$]: Let $F = \mathbf{D_3}(D_1)$.

[$\mathbf{AD_6}$]: Return (E, F).

Proposition 3.1 *Algorithm* HCADDDBL(D_1, D_2) *computes* $D_1 + D_2$ *and* $2D_2$ *with a cost of 1[i]+48[m]+7[s].*

Proof : Computational cost of steps $[\mathbf{AD_1}], [\mathbf{AD_2}], [\mathbf{AD_4}]$ and $[\mathbf{AD_5}]$ is the sum of those of $\mathbf{A_1}, \mathbf{A_3}, \mathbf{D_1}, \mathbf{D_3}$. That is cost of $[\mathbf{AD_1}], [\mathbf{AD_2}], [\mathbf{AD_4}]$ and $[\mathbf{AD_5}]$ is 45 multiplications and 7 squarings. The cost of step $[\mathbf{AD_3}]$, by lemma 3.1, is 1[i]+3[m]. So computational cost of the algorithm is 1[i] + 48[m] + 7[s]. ■

In the literature, generally $1[i]$ is considered compuationally equivalent to $30[m]$ and one $1[s]$ is considered equivalent to $0.8[m]$ (see [16], [7]). Using these estimates, the cost of HCADDDBL is equivalent to 83.6[m]. The corresponding costs for the algorithms presented in [9] and [10] are 110.6[m] and 95[m] respectively. We note that a similar approach can be used to develop a simultaneous add-and-double algorithm for the genus 3 curves [15]. The resulting algorithm will be more efficient than the algorithms of [15] and also SCA resistant.

This motivates us to investigate further in the direction suggested by lemma 3.1. Lemma 3.1 presents us a novel way of computing inverses of two field elements. How far can we go ? To compute inverse of n elements how many multiplications and one inversion will suffice ? We propose an algorithm for doing it efficiently in the next section.

4 Computing Inverses Efficiently

Suppose we wish to compute the inverses of the field elements $\alpha_0, \cdots, \alpha_{n-1}$. For simplicity, let us assume, $n = 2^r$ for some r. Consider the full binary tree with levels numbered 0 to r. In the illustration (see Fig 2) we have taken $r = 2$. At jth node of the i-th level of the tree, we associate a set S_j^i and an expression β_j^i, $0 \le i \le r; 0 \le j \le 2^r - 1$. At the leaf (i.e. rth) level, we take $\beta_j^r = \alpha_j$ and $S_i^r = \{1\}$ for all i. From the S_j^i and β_j^i, we compute S_j^{i-1} and β_j^{i-1} as follows. We use the following notation: for any set of field elements A and any field element x, we define, $xA = \{xy : y \in A\}$.

We define, $S_j^{i-1} = \{x \times y : x \in S_{2j}^i, y = \beta_{2j+1}^i \text{ or } x \in S_{2j+1}^i, y = \beta_{2j}^i\} = \beta_{2j+1}^i S_{2j}^i \bigcup \beta_{2j}^i S_{2j+1}^i$ and $\beta_j^{i-1} = \beta_{2j}^i \times \beta_{2j+1}^i$. We claim that:

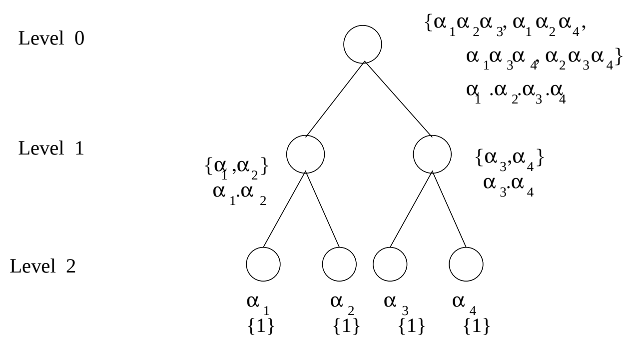

Figure 2: Computing S_j^i and β_j^i for $n = 4$.

Lemma 4.1 $S_0^0 = \{\overline{\alpha_0}, \overline{\alpha_2}, \cdots, \overline{\alpha_{n-1}}\}$ *and* $\beta_0^0 = \beta = \alpha_0 \times \alpha_2 \times \cdots \times \alpha_{n-1}$, *where* $\overline{\alpha_i} = \alpha_0 \ldots \alpha_{i-1}\alpha_{i+1} \ldots \alpha_{n-1}$.

Proof : Observe that, at rth level, $S_j^r = \{1\}$ and $\beta_j^r = \alpha_j$. At level $r-1$, S_j^{r-1} contains 2 elements of $\alpha_1, \alpha_2, \cdots, \alpha_n$ and β_j^{r-1} are product of two elements each of $\alpha_1, \alpha_2, \cdots, \alpha_n$. We claim that, at level $(r-i)$, each of the sets S_j^{r-i} contains 2^i elements, each of which is a product of $2^i - 1$ elements of $\alpha_1, \alpha_2, \cdots, \alpha_n$ and β_j^{r-i} are products of 2^i elements of $\alpha_1, \alpha_2, \cdots, \alpha_n$. Clearly, our claim is true for the $i = 0$. Suppose, it is true for some $(r-k)$th level, i.e. $i = k$. Then for all j, we have $|S_j^{r-k}| = 2^k$ and each element of S_j^{r-k} is a product of $2^k - 1$ elements and each of β_j^{r-k} is a product of 2^k elements. Let us consider the level just above it i.e.$(r-k-1)$th level. By definition, $S_j^{r-k-1} = \beta_{2j+1}^{r-k} S_{2j}^{r-k} \bigcup \beta_{2j}^{r-k} S_{2j+1}^{r-k}$. Hence for all j, $|S_j^{r-k-1}| = |S_{2j}^{r-k}| + |S_{2j+1}^{r-k}| = 2^k + 2^k = 2^{k+1}$. Also, As elements of the sets S_{2j}^{r-k} and S_{2j+1}^{r-k} are products of $2^k - 1$ elements and each of β_j^{r-k} is a product of 2^k elements, the elements of S_j^{r-k-1} are products of $2^{k+1}-1$ elements of $\alpha_1, \alpha_2, \cdots, \alpha_n$. Also, by definition, it is clear that β_j^{r-k-1} is product of 2^{k+1} elements. This establishes our claim.

Hence, at 0th level i.e. the root, S_0^0 will be the set of products of $2^r - 1 = n - 1$ elements of $\alpha_1, \alpha_2, \cdots, \alpha_n$ and β_0^0 will be the product of $2^r = n$ elements of $\alpha_1, \alpha_2, \cdots, \alpha_n$, i.e. all of them. This establishes that $S_0^0 = \{\overline{\alpha_1}, \overline{\alpha_2}, \cdots, \overline{\alpha_n}\}$ and $\beta_0^0 = \beta = \alpha_1 \times \alpha_2 \times \cdots \times \alpha_n$. ■

Observe that the data associated with the nodes of the tree is a list L. The list L consists of two elements: a field element β and a set of field elements S, i.e. $L = (\beta, S)$. We propose the following recursive algorithm to merge lists at any number of leaf level nodes and compute the list at the root.

Algorithm MERGELST
Input: Field elements $\alpha_0, \alpha_1, \cdots, \alpha_{n-1}$.
Output: The list L at the root.
1. If $n = 2$ return $(\alpha_0\alpha_1, \{\alpha_0, \alpha_1\})$
2. Let $(\beta^1, S_1) =$ MERGELST$(\alpha_0, \cdots, \alpha_{\lfloor n/2 \rfloor})$
3. Let $(\beta^2, S_2) =$ MERGELST$(\alpha_{\lfloor n/2 \rfloor + 1}, \cdots, \alpha_{n-1})$
4. Return$(\beta^1\beta^2, \beta^1 S_2 \bigcup \beta^2 S_1)$

Proposition 4.1 *The cost of merging n lists by MERGELST is $O(n \log n)$ multiplications. When $n = 2^r$, the exact number of multiplication required is $r2^r - 1$.*

Proof : We observe that complexity of the MERGELIST is the solution of the recurrence relation

$$T[n] = T[\lfloor n/2 \rfloor] + T[\lceil n/2 \rceil] + n + 1$$

Solving this relation we get, $T[n] = O(n \log n)$. ■

Now we can state the general INVERT algorithm for inverting any number of field elements.

Algorithm INVERT
Input: Field elements $\alpha_0, \cdots, \alpha_{n-1}$.
Output: $\alpha_0^{-1}, \cdots, \alpha_{n-1}^{-1}$.
1. Let $(\beta, S) =$ MERGELST$(\alpha_0, \cdots, \alpha_{n-1})$
2. Return$(\beta^{-1} S)$

The folowing proposition states the complexity of the algorithm INVERT.

Proposition 4.2 *The cost of algorithm INVERT is $1[i] + O(p \log p)[m]$. When $n = 2^r$ the exact cost is $1[i] + (r+1)(2^r - 1)[m]$. For other values of n, the exact number of multiplications required can be found by a trace of the algorithm MERGELST.*

As an example, we can compute inverses of 16 elements by 1 inversion and 79 multiplications. Similarly, to compute inverses of 64 elements we require 1 inversion and 447 multiplications. Assuming that the cost of one inversion is approximately equal to that of 30 multiplications [16, 7], the cost of directly computing the inverses for 16 and 64 elements would be 480 and 1920 multiplications respectively.

5 A New Method for Scalar Multiplication in HECC

Let $w \geq 2$ be a positive integer. This is our window size. 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 compute $c_0 D, c_1 2^w D, \cdots c_{t-1} 2^{w(t-1)}$

simultaneously 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 $m_{n-1}\dots m_0$. 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 several divisor type variables R_j and $Q_j^{(0)}$ which are initialized to $2^{jw}D$ and 0 respectively. Then 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 $Q_j^{(0)}$ and double R_j. If the scanned bit is 0, then we do not add, only double R_j. After w iterations, $Q_j^{(0)}$ is $c_j 2^{wj} D$. For $0 \le j \le t-1$, 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 the method described in section 4. This yields a very efficient algorithm for scalar multiplication.

One problem with the above approach is that the addition of the divisors depends on the bit c_j^i. Thus the power consumption trace when $c_j^i = 0$ is different from the power consumption trace when $c_j^i = 1$. This makes the algorithm vulnerable to simple power analysis attacks. To avoid this problem, we use the following idea. We take one more variable $Q_j^{(1)}$ for each j. In each iteration, we add R_j to $Q_j^{(c_j^i)}$ and double R_j.

Algorithm EFF-SCLR-MULT

Input: $m, t, c_0, c_1, \cdots, c_{t-1}, r, D$.

Output: mD.

1. For $j = 0$ to $t-1$ set $R_j = T[j], Q_j^{(0)} = Q_j^{(1)} = 0$
2. For $i = 0$ to $w-1$
3.   For $j = 0$ to $t-1$
4.   let $\alpha_j = \mathbf{A_1}(R_j, Q_j^{c_j^i})$;
5.   let $\beta_j = \mathbf{D_1}(R_j)$.
6.   end do.
7.   Let $(\beta_0^{-1}, \cdots, \beta_{t-1}^{-1}, \alpha_0^{-1} \cdots \alpha_{t-1}^{-1}) = \mathrm{INVERT}(\beta_0, \cdots, \beta_{t-1}, \alpha_0 \cdots \alpha_{t-1})$
8.   For $j = 0$ to $t-1$
9.   let $Q_j^{(1)} = \mathbf{A_3}(R_j, Q_j^{c_j^i})$; let $Q_j^{(0)} = \mathbf{D_3}(R_j)$;
10.   end do.
11. end do.
12. Let RES = TreeADD$(Q_0^{(1)}, \cdots, Q_{t-1}^{(1)})$
13. Return (RES)

Note that the doubling of R_j, $0 \le j \le t-1$, in the $(w-1)$th iteration is not required. Avoiding this doubling will make the algorithm faster. However, the description will become more complicated. Hence, we have chosen not to do that for the present.

In the algorithm EFF-SCLR-MULT we invoke an algorithm TreeADD. It computes the sum of a given number of divisors efficiently. After describing the algorithm TreeADD and proving its complexity we will return to the complexity of the algorithm EFF-SCLR-MULT.

The algorithm TreeADD adds a number of divisors efficiently. It uses a tree structure for computation. Suppose $D_0, D_1, \cdots, D_{k-1}$ are the input divisors. For simplicity, assume $k = 2^r$. Imagine a tree of depth r with the input divisors at the leaf nodes. We pairwise add the divisors 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 the level $r-1$ and to get the divisors at these nodes we have to compute 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 $r2^{r-1}-1$ multiplications using algorithm INVERT. This process is carried out at each level upto the root. The root then contains the sum of all the divisors.

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 $(D_0^{(j)}, D_1^{(j)}, \cdots, D_{2^{k-j}-1}^{(j)}) = \text{ADD}(D_0^{(j-1)}, D_1^{(j-1)}, \cdots, D_{2^{k-j+1}-1}^{(j-1)})$
4. Return $(D_0^{(r)})$

TreeADD invokes the algorithm ADD, which takes as input $2k$ divisors, $D_0, D_2, \cdots, D_{2k-1}$ 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 algorithm INVERT of section 4.

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 $E_i = \mathbf{A_3}(D_{2i}, D_{2i+1})$.
4. Return $(E_0, E_1, \cdots, E_{k-1})$.

In the following we will use the following notation: Let A^A (resp. D^A) be the number of multiplications and squarings made by HCADD (resp. HCDBL) algorithm in [9]. We further assume that A^A and D^A are expressed in terms of equivalent multiplication operations only, i.e., we assume $[s] = 0.8[m]$. Hence $A^A = 24.6[m]$ and $D^A = 26[m]$.

Proposition 5.1 *ADD takes* $(2^r A^A + ((r+1)2^r - 1)[m] + 1[i]$ *computations to compute the* $k = 2^r$ *sums of* $2k = 2^{r+1}$ *input divisors.*

Proof : From proposition 4.2 the invocation of INVERT in step 2 of the algorithm takes $((r+1)2^r - 1)[m] + 1[i]$ computations. One iteration of steps 1 and 3 combined takes $A^A[m]$ computations. There are 2^r such iterations. Hence the proposition follows. ∎

Now we can compute the complexity of the algorithm TreeADD. TreeADD repeatedly calls the algorithm ADD, first with 2^r divisors, then with 2^{r-1} divisors and so on. Let the cost of ADD with k input divisors be $[kA]$. Then, cost of TreeADD with 2^r divisors is $[2^r A] + [2^{r-1} A] + \cdots + [1A]$. By proposition 5.1, $[2^i A] = (2^{i-1} A^A + (i2^{i-1} - 1))[m] + 1[i]$. Hence computational cost of TreeADD is $\sum_{i=1}^{r-1}[2^i A] = \sum_{i=1}^{r-1}((2^{i-1} A^A + (i2^{i-1} - 1))[m] + 1[i]) = (2^r - 1)(A^A + r - 1) + r[i]$. Thus we have proved:

Proposition 5.2 *TreeADD takes* $(2^r - 1)(A^A + r - 1) + r[i]$ *computations to compute the sum of* 2^r *input divisors.*

We now compute the complexity of the algorithm EFF-SCLR-MULT. In each iteration of the i-loop in the algorithm EFF-SCLR-MULT, we add t divisors and double t divisors. So in each iteration we invoke INVERT with $2t$ field elements.

Proposition 5.3 *EFF-SCLR-MULT takes* $(w+r)[i] + [wtD^A + wtA^A + w((r+2)2^{r+1} - 1) + (t-1)(A^A + r - 1)][m]$ *computations to compute the scalar multiplication* mD *where,* m *is an* n*-bit integer,* w *is the window size and* $t = \lceil n/w \rceil = 2^r$.

Proof : EFF-SCLR-MULT computes the scalar multiplication mD by computing $c_i 2^{iw} D$ for $0 \le i \le t-1$ and then computing their sum. Let $t = 2^r$. Then $c_i 2^{iw} D$ are computed in steps 4 to 18. In one iteration of the i-loop in step 3, we read t bits c_i^j, compute t doublings and t additions. This will involve tD^A multiplications for doublings and tA^A multiplications for additions. At step 13, algorithm EFF-SCLR-MULT calls algorithm INVERT to invert the field elements for these additions and doublings. Number of arguments passed to the algorithm INVERT is $2t$. Hence by proposition 4.2, INVERT requires $((r+2)2^{r+1}-1)[m]+1[i]$ operations to compute these inverses. Since the i-loop iterates w times, to compute $c_i 2^{iw} D$ for all i, the algorithm EFF-SCLR-MULT takes $w(tD^A + tA^A + ((r+2)2^{r+1}-1)[m]+1[i])$ computations. To compute the sum of the divisors $c_i 2^{iw} D$, EFF-SCLR-MULT invokes the algorithm TreeADD with t variables as input. By proposition 5.2 this invocation involves $(2^r-1)(A^A + r - 1)[m] + r[i]$.

To sum up, EFF-SCLR-MULT takes $w(tD^A + tA^A + ((r+2)2^{r+1}-1)[m]+1[i]) + (2^r-1)(A^A + r-1)[m]+r[i] = (w+r)[i] + [wtD^A + wtA^A + w((r+2)2^{r+1}-1) + (t-1)(A^A+r-1)][m]$ computations to compute mD. ■

6 Parallelisation

In this section, we discuss how the algorithm EFF-SCLR-MULT can be implemented in parallel. Due to lack of space, we do not discuss parallelism in details, we will just highlight the parts of the algorithm which allow parallel processing.

First we observe that the algorithm INVERT for computing inverses of field elements simultaneously, which led to development of the algorithm EFF-SCLR-MULT can be implemented in a parallel architecture. Suppose 2^r elements are to be inverted and we have a system with 2^{r-1} processors. We start computing at $(r-1)$st level of the tree. The ith $(0 \le i \le 2^{r-1})$ processor receives two field elements $\alpha_{2i}, \alpha_{2i+1}$ and computes the product $\alpha_{2i}\alpha_{2i+1}$ and sends the list $L = (\alpha_{2i}\alpha_{2i+1}, \{\alpha_{2i}, \alpha_{2i+1}\})$ to the $(\lfloor i/2 \rfloor)$th processor. This completes 1st round. In the jth round, there will be 2^{r-j} processors getting 2 lists each, one from each at its child node. Again the ith processor will merge the lists by the formula described in section 4 and send it to $\lfloor i/2 \rfloor$th processor. Each processor is a field element multiplier. Clearly, the computation will end in r parallel rounds.

In the algorithm EFF-SCLR-MULT, we fix a window size w and then write m in the base 2^w. If the representation of m in base 2^w is $c_0 + c_1 2^w + \cdots + c_{t-1} 2^{(t-1)w}$ then we compute $c_i 2^{wi} D$, for $0 \le i \le t-1$. These computations can be carried out in parallel. We can use t processors to compute $\alpha_j = \mathbf{A_1}(R_j, Q_j^{c_j^i})$ in one parallel round. The processors involved are hyperelliptic curve adders. At the same time we can use t processors (each capable of doubling a divisor) to perform the task $\beta_j = \mathbf{D_1}(R_j)$ in parallel. Then all the $2t$ number of elements α_j and β_j can be inverted in parallel in $\log t$ rounds of the algorithm described in the preceeding paragraph. The steps of EFF-SCLR-MULT involving $\mathbf{A_3}, \mathbf{D_3}$ can similarly be carried out in parallel. Thus one iteration of EFF-SCLR-MULT involves the parallel execution of $\mathbf{A_1}$, $\mathbf{D_1}$; then one parallel execution of INVERT with $2t$ elements; followed by parallel execution of $\mathbf{A_3}$, $\mathbf{D_3}$. As there are w iterations, all $c_i 2^{wi} D$ can be computed in w rounds.

Finally to add the $c_i 2^{wi} D$'s, we invoke the TreeADD algorithm. The algorithm computes the sum in a tree structure, with the input divisors at the leaves. To add these t divisors, we can use $t/2$ adder-type processors in the first level. At jth level we will be using $t/2^{r-j}$ processors. TreeADD algorithm can thus be easily parallelised with $\log t$ parallel rounds of computations and the time taken at each parallel round is equal to that of a divisor addition.

Further details of parallel implementation will be provided in the full version of the paper.

Table 1: Complexity of different algorithms. In (a) and (b) inversion-free arithmetic [10] is used and in (c) and (d) arithmetic of [9] is used.

	ALGORITHM	COMPLEXITY
(a)	DBL-AND-ADD-L	$69.9n[m]$ (average case)
(b)	DBL-AND-ALWS-ADD	$95n[m]$
(c)	ENC-DBL-AND-ADD	$83.6n[m]$
(d)	EFF-SCLR-MULT	$((50.6+2r)n+(t-1)(23.6+r)+29w+30r)[m]$

7 Comparison

In this section, we compare the efficiency of this algorithm with that of some of the existing algorithms. We note that in HECC with a curve of genus 2, the elements of underlying field should be at least 80 bits in length to provide adequate security. In that case the size of the Jacobian is approximately 160 bits. Hence the scalar multiplier can be assumed to be around 160 bit integer.

Of the proposed algorithms ENC-DBL-AND-ADD (section 3) is SCA resistant and algorithm EFF-SCLR-MULT is SPA resistant. A scalar multiplication algorithm using inversion-free arithmetic for addition and doubling proposed in [10] requires $44.8[m]$ for doubling and $50.2[m]$ for addition. So, for an n-bit integer, the usual scalar multiplication algorithm (DBL-AND-ADD-L of section 3) using inversion-free arithmetic of [10] will take, on the average, $44.8n+50.2(n/2) \approx 69.9n[m]$. However, it is vulnerable to side-channel attacks. It will be SCA resistant if it carries out one addition for each bit of m, irrespective of whether it is 1 or 0. In this case, n additions and n doublings are to be carried out to compute the scalar multiplication. Let us name this algorithm as ENC-DBL-AND-ADD-L-SCAR. ENC-DBL-AND-ADD-L-SCAR will take $(44.8n+50.2n) \approx 95n[m]$. If ADD-AND-ALWS-DBL is used for computing the scalar multiplication, for each bit it will require $1[i]+48[m]+7[s] \approx 83.6[m]$. Besides there is a doubling at the begining, which will cost $1[i]+22[m]+5[s] \approx 56[m]$. So the total cost of the computation will be $(83.6n+56)[m]$. Algorithm EFF-SCLR-MULT takes $((50.6+2r)n+(t-1)(23.6+r)+29w+30r)[m]$ to compute the scalar multiplication, where r, t, w have their usual meaning. (Using algorithms from [9], we take $D^A = 22[m]+5[s] \approx 26[m]$ and $A^A = 23[m]+2[s] \approx 24.6[m]$.) Table 1 summarises these calculations.

It is clear from the above table that, ENC-DBL-AND-ADD is better than DBL-AND-ALWS-ADD. Although the average case complexity of DBL-AND-ADD-L is better than ENC-DBL-AND-ADD, the former is not SCA resistant.

To compare efficiencies of EFF-SCLR-MULT with others we take some representative values of n and w and calculate the exact amount of computation required for these algorithms in terms of multiplication only. The following tables displays those calculations. In table 2 columns (a)-(d) refer to the algorithms listed in rows (a)-(d) of table 1. 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 divisors required to be stored.

One can easily observe that, for certain window sizes, algorithm (d) is better than even average case complexity of DBL-AND-ADD (coulumn (a)), which is not SPA resistant. In the best scenario, the new algorithm achieves a speed-up of around 44 percent in all three case $n = 160, 192, 256$ over (b). In efficiency, our simple ENC-DBL-AND-ADD algorithm (column (c)) is better than DBL-AND-ALWS-ADD(column (b)). It can be checked that in all case the former achieves around 13

Table 2: Comparison for specific values of n.

n	w	t	(a)	(b)	(c)	(d)
160	5	32	11484	15200	13432	11517.6
160	10	16	11484	15200	13432	10840
160	20	8	11484	15200	13432	10552.2
160	40	4	11484	15200	13432	10672.8
160	80	2	11484	15200	13432	11430.6
192	3	64	13420.8	18240	16107.2	14919
192	6	32	13420.8	18240	16107.2	13613.8
192	12	16	13420.8	18240	16107.2	12901.2
192	24	8	13420.8	18240	16107.2	12607.4
192	48	4	13420.8	18240	16107.2	12780
192	96	2	13420.8	18240	16107.2	13705.8
256	2	128	17849.4	24320	21457.6	21715.8
256	4	64	17849.4	24320	21457.6	19210.4
256	8	32	17849.4	24320	21457.6	17806.2
256	16	16	17849.4	24320	21457.6	17023.6
256	32	8	17849.4	24320	21457.6	16717.8
256	64	4	17849.4	24320	21457.6	16994.4

percent speed-up over the later.

The parameter t in table 2 determines the size of the look-up table. It is equal to the number of divisors to be stored in the look-up table. It can be observed that for the cases, when the algorithm gives the best result, the table size is also considerably low. For example, for all three values of n, the algorithm has the best performance for $t = 8$. In hyperelliptic curve cryptosystem with reasonable security, a divisor size is around 320 bits. So, the table will occupy only 320 bytes of memory.

8 Conclusion

In this paper, we present two new algorithms for scalar multiplication in the Jacobian of hyperelliptic curves. The first one is SCA resistant and is a simple consequence of the fact that inverses of two field elements can be computed with one inversion and 3 multiplications. We investigate the idea further to propose an algorithm to efficiently compute the inverses of a set of field elements. We use this newly proposed algorithm to develop a scalar multiplication algorithm, which is SPA resistant and achieves a speed-up of 44 percent over any such existing algorithms. A key feature of our algorithm is that it uses addition and doubling algorithms involving inversion, which are considered less efficient than the inversion-free ones. We believe that further research is required to settle the following question: In the context of scalar multiplication, when is inversion-free arithmetic is better than arithmetic involving inversion?

References

[1] *http://www.arehcc.com*

[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] R. Harley Fast Arithmetic on Genus 2 Curves. *Avaiable at http://cristal.inria.fr/ harley/hyper,2000.*

[5] N. Koblitz. Hyperelliptic Cryptosystems. In *Journal of Cryptology*, 1: pages 139–150, 1989.

[6] N. Koblitz. *Algebraic Aspects of Cryptology*, Algorithms and Computation in Mathematics. Springer Verlag, 1998.

[7] A. J. Menezes, P. C. van Oorschot and S. A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1997.

[8] C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *CRYPTO 96*, LNCS 1109, pp 104-113, Springer-Verlag, 1996.

[9] 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/* .

[10] T. Lange. Inversion-free Arithmetic on Genus 2 Hyperelliptic Curves. Cryptology ePrint Archive, Report 2002/147, 2002. *http://eprint.iacr.org/* .

[11] Y. Miyamoto, H. Doi, K. Matsuo, J. Chao and S. Tsuji. A fast addition algorithm for genus 2 hyperelliptic curves. In *Proc of SCIS2002, IEICE, Japan*,pp 497-502,2002, in Japanese.

[12] 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

[13] P. Montgomery. Speeding the Pollard and Elliptic Curve Methods for Factorisation. In *Math. Comp.*, vol 48, pp 243-264, 1987.

[14] K. Nagao. Improving Group Law Algorithms for Jacobians of Hyperelliptic Curves. In W. Bosma, editor, *ANT IV*, LNCS 1838, Berlin 2000, Springer-Verlag.

[15] 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/* .

[16] 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.

[17] 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

[18] 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.

[19] A. M. Spallek Kurven vom Geschletch 2 und irhe Anwendung in Public-Key-Kryptosystemen. Ph D Thesis, Universitat Gesamthochschule, Essen, 1994.

[20] M. Takahashi. Improving Harley Algorithms for Jacobians of Genus 2 Hyperelliptic Curves. In *Proc of SCIS 2002*, ICICE, Japan, 2002, in Japanese.