Novel Table Look-up Methods for Faster Frobenius Map Based Scalar Multiplication in the Jacobian of Hyperelliptic Curves

Rana Barua, Pradeep Kumar Mishra, Palash Sarkar.
Indian Statistical Institute,
203, B.T. Road, Calcutta 700 108, INDIA

Abstract

Koblitz proposed the hyperelliptic cryptosystem in 1989. Since then continuous efforts have been made by the research community to reduce the complexity of implementation of the hyperelliptic curve cryptosystem and make it availabe for cryptographic utilisation. One approach, as suggested by Koblitz himself in his original paper, is to use Frobenius map to speed up the scalar multiplication. Several researchers, working on that suggestion have come up with algorithms, which perform faster than the traditional ones. In this paper, we use Frobenius map and some novel table look-up methods to bring down the complexity. Also, we incorporate the new table look-up methods with window methods to propose new algorithms which are more efficient than the existing ones. Depending on the extension degree of the underlying field one can always choose one of the proposed algorithms which performs better than the previously proposed ones and uses a small amount of storage for the look-up table.
Keywords : Frobenius map, hyperelliptic curves, Jacobian, window methods, look-up table, normal basis.

1 Introduction

Neil Koblitz (1989) proposed Jacobian of hyperelliptic curves as a rich source of additive abelian groups to be used for cryptographic purposes. However, hyperelliptic curve cryptosystems (HECC) are not yet commercially available. One of the main reasons for this is that implementation issues for HECC are still at the research level. Fortunately, this task has recently been taken up seriously by the cryptographic community. Particularly in Europe, a EU sponsored project AREHCC, involving experts from seven academic and industrial organisations is making a concerted effort in the implementation of HECC. This paper should be seen as another contribution to the implementation aspects of HECC.

The most important and expensive operation in an HECC is scalar multiplication, i.e. the operation of finding an integer m times a divisor D, where D is a base divisor and m is an integer. The performance of HECC can be significantly improved if the scalar multiplication can be performed efficiently. The general technique to compute mD is the double-and-add approach which has been described in section 2. In his original paper proposing the HECC, Neal Koblitz had proposed use of Frobenius expansion to speed-up the scalar multiplication. In [6], [9], [12], [13] a family of elliptic curves, called *Koblitz curves or anomalous binary curves(ABC Curves)* was investigated, which allows faster scalar multiplication using the Frobenius automorphism. In [3], Gunther et al generalised the same ideas to Koblitz curves of genus 2. They have reported an performance enhancement of 19 percent. In [2], Choie and Lee carried over the idea to the Jacobian J_F of general hyperelliptic curves in certain cases reported a 41 percent improvement over generic double-and-add approach.

In the current work, we adopt the same technique, but carry out the additions and doublings using new table look-up based techniques to improve the over all performance. In a hyperelliptic security domain, the underlying field, the curve and its Jacobian, size of the Jacobian are fixed. If an ElGamal type cryptosystem or Diffie-Helman key exchange are implemented over it, then the scalar multiplication i.e. computing mD is the dominant operation. In most cryptographic applications, the base divisor D is fixed and m varies from user to user. Therefore some precomputations involving D can be done and stored as a part of the set-up of the cryptographic system. In this work, we show that under such a domain, the scalar multiplication can be efficiently done using the precomputed tables. The involved table size is also not large. In certain situations the table size is small enough to be implemented on smart cards also.

The creation of look-up tables is based on some new ideas, which helps in reducing the table size. We propose new algorithms using these tables which are more efficient than the existing ones. The window methods have been successfully utilised in elliptic curves (see for example [11]). We incorporate our new ideas into window methods to provide algorithms which are still more efficient in terms of both computations and table size. Note that, in computations we do not invlove the cost of precomputations for table creation. That is because the tables are to be created once and to be retained as part of the security domain. These tables do not change unless the base divisor changes. Also, these tables are independent of the scalar m.

In Section 2, we provide a brief description of arithmetic on hyperelliptic curves. In Section 3, we describe the Frobenius map based scalar multiplication. Sections 4, 5 and 7 describe the new algorithms. In Section 7 we compare efficiency of the proposed algorithms with the existing ones. Finally, Section 8 concludes the paper.

2 Arithmetic in the Jacobian of a Hyperelliptic Curve:

In this section we give a brief overview of hyperelliptic curve cryptosystems. For details, readers can refer to [4], [10] or [1]. 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.

If L is any extension field of K, then the set of all L-rational points of C is the set $\{(x, y) \in L \times L : y^2 + h(x)y = f(x)\} \bigcup \{\infty\}$ where ∞ is a special point called the point at infinity. The opposite point of a point $P(x, y)$ is $\tilde{P}(x, -h(x) - y)$. A point for which $P = \tilde{P}$ is called a special point otherwise, it is called ordinary.

A *divisor* D is a formal sum, $D = \sum_{P \in C} m_P P$, where $m_P \in \mathcal{Z}$ and $\mathcal{Z}$ 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 $\mathcal{D}$.

Let R be a rational function defined over C. A unique divisor can be associated to R, called the divisor of R, denoted by $\text{div}(R)$. A divisor D is called a *principal divisor* if $D = \text{div}(R)$ for some rational function R. The set $\mathcal{P}$ of all principal divisors is a subgroup of D^0. The *Jacobian* $\mathcal{J}$ of C is the quotient group $\mathcal{D}^0/\mathcal{P}$.

It is possible to do cryptography in the Jacobian. For example one can define an ElGamal type cryptosystem in a suitable cyclic subgroup of the Jacobian. The elements of such a group, which are called *reduced divisors*, have a nice cannonical representation by means of two polynomials $a(u)$ and $b(u)$ defined over K. We will use the notation $div(a, b)$ for the divisor represented by $a(u)$ and $b(u)$.

To compute mD we may resort to generic double-and-add approach. In this case we need $\log m$ doublings and $\text{wt}(m)$ additions, where $\text{wt}(m)$ is the hamming weight of binary representation of m. The classical double-and-add algorithm is described below.

Let $m = m_0 + m_1 2 + m_2 2^2 + \cdots + m_r 2^r$, be the binary representation of m.

Algorithm 1

Input : $D, m(m_0, m_1, \cdots, m_r)$.

Output : mD.

1. let $X = D$.
2. for $i = r - 1$ down to 0
3. $\quad X = 2 * D$,
4. $\quad$ If $m_i = 1$ then $X = X + D$
5. return X

Hyperelliptic curve addition and doubling do not take same amount of computations. Lange [8] has reported that in hyperelliptic curves of genus 2, where these operations can be carried out efficiently through explicit formulae, these operations take 4 squaring, 47 multiplications (4[S] + 47[M]) and 6 squaring, 40 multiplications (6[S] + 40[M]) of the underlying field elements, respectively in most general case.

3 Scalar Multiplication using the Frobenius map.

In [4], Koblitz suggested to make use of a Frobenius expansion to speed-up the scalar multiplication in the Jacobian of hyperelliptic curves. Gunther, Lange and Stein [3] modified Koblitz's Frobenius expansion method and applied it to Koblitz curves of genus 2 over F_2. Choie and Lee [2] generalised this result to the case of curves of arbitrary genus defined over finite fields of any characteristic. In this section we provide a brief description of the method.

Let F_q be a finite field. The Frobenius map $\phi : F_{q^n} \to F_{q^n}$ is defined as $\phi(x) = x^q$. If $C : y^2 + h(x)y = f(x)$ is a hyperelliptic curve of genus g defined over F_q and $J(F_{q^n})$ is its Jacobian over the field F_{q^n}, then the Frobenius automorphism ϕ on F_{q^n} induces an endomorphism on $J(F_{q^n})$ defined as follows : For aWe point P(x, y) on C, we define P^ϕ to be the point (x^ϕ, y^ϕ). If $D = \sum m_i P_i - (\sum m_i)(\infty)$, then $D^\phi = \sum m_i P_i^\phi - (\sum m_i)\infty$. We define u^{ϕ^i} as follows. If $u = x^g + u_{g-1}x^{g-1} + \cdots + u_1 x + u_0$, then $u^\phi = x^g + \phi(u_{g-1})x^{g-1} + \cdots + \phi(u_1)x + \phi(u_0)$. We can define u^{ϕ^i} similarly. Also, it can be seen that if $D = \text{div}(u(x), v(x))$, then $D^\phi = \text{div}(u(x)^\phi, v(x)^\phi)$. We note that ϕ^n is the identity map on F_{q^n}. So the induced map ϕ^n on $J(F_{q^n})$ is also identity. The characteristic polynomial $\chi(T)$ of the induced Frobenius map ϕ on $J(F_{q^n})$ is given by

$$\chi(T)=\sum_{i=0}^{2g}a_iT^i=a_{2g}T^{2g}+a_{2g-1}T^{2g-1}+\cdots+q^{g-2}a_{2g-2}T^2+q^{g-1}a_{2g-1}T+q^g$$

where $a_{2g}=1$, $a_i=q^{g-i}a_{2g-i}$ for $0\leq i\leq g$.

Let I be the set of integers in $(-q^g,q^g]$. The reduced base-ϕ expansion of m is $\sum_{i=0}^{n-1}u_i\phi^i$, where each $u_i\in I$ under suitable assumtions. It is easy to obtain the reduced base-ϕ expansion of m. For details readers can refer to [2], [3].

After obtaining the reduced base-ϕ expansion of m the scalar multiplication mD can be carried out by the following algorithm. This algorithm was proposed by Choie and Lee in [2].

For $i=0,1,...,n-1$, let $D_i=D^{\phi^i}$ i.e. if $D=\text{div}(u,v)$ then $D_i=\text{div}(u^{\phi^i},v^{\phi^i})$. Given any integer m, first we find the base-ϕ expansion $\sum_{i=0}^{n-1}u_i\phi^i$ of m. Then, for each $j=0,1,...,n-1$, we write $|u_j|=\sum_{i=0}^{A}u_i^{(j)}2^i$, where $A=max\lfloor\log_2 u_j\rfloor$

Algorithm 2.
Input : integer m and divisor D .
Output : mD.

1. Compute $D_i=D^{\phi^i}$ for $i=0,1,2,...n-1$.
2. Find the reduced base-ϕ expansion $\sum_{i=0}^{n-1}u_i\phi^i$ of m.
3. For $i=0$ to $n-1$,
   write $|u_i|=\sum_{j=0}^{A}u_j^{(i)}2^j$
4. Set $D_i=(\text{sign } u_i)D_i$ for all i.
5. Set $X=\sum_{i=0}^{n-1}u_{(i)}^{A}D_i$
6. For $i=A-1$ down to 0
7. $\quad X=2X$
8. $\quad X=X+\sum_{i=0}^{n-1}u_j^{(i)}D_i$
9. return X.

In the above algorithm, the average number of additions and doublings needed to compute mD are $n(A+1)/2$ and A respectively.

4 Table Look-up Methods

The efficiency of the above algorithm can be improved by computing the doubling and additions in steps 5, 7, and 8 more efficiently. In the present work we explore several methods for computing these sums and doubles and compare their efficiencies.

Let $\sum_{j=0}^{A}u_j^{(i)}2^j$ be the binary expansion of $|u_i|$, where $u_j^{(i)}\in\{0,1\}$. In steps 5 and 8 of the algorithm 2 we are computing the sums like $\sum_{i=0}^{n-1}u_j^{(i)}D_i$, where $D_i=(\text{sign } u_i)D_i$ for all i. In fact $\sum_{i=0}^{n-1}u_j^{(i)}D_i=u_iD_i$. If we allow $u_j^{(i)}$ to be negative also i.e. $u_j^{(i)}\in\{0,\pm1\}$, then we can write,

$$\left.\begin{array}{lcl}u_0D_0 & = & (u_0^0+u_1^02+\cdots+u_A^02^A)D_0\\ u_1D_1 & = & (u_0^1+u_1^12+\cdots+u_A^12^A)D_1\\ \vdots & \vdots & \vdots\\ u_{n-1}D_{n-1} & = & (u_0^{n-1}+u_1^{n-1}2+\cdots+u_A^{n-1}2^A)D_{n-1}\end{array}\right\}\qquad(2)$$

where $D_i = D_0^{\phi^i}$.

We observe that the right hand side has the structure of a matrix. So we can compute this sum row-wise or column-wise or block-wise. In algorithm 2 we are computing it column-wise. It is easy to see that the row-wise computation would not be efficient as it will require many more doublings. Look-up table based block-wise computation is a lucrative proposition. Further we can go for window method to compute these sums. In this work, we propose some new type of table-look-up methods to calculate the scalar multiplication.

For all possible combination of $a_0, a_1, \cdots, a_{n-1} \in \{0, \pm 1\}$ we precompute, $a_0 D_0 + a_1 \phi(D_0) + \cdots + a_{n-1} \phi^{n-1}(D_0)$ and store it in a look-up table T. Thus for every n-tuple $(a_0, a_1, \cdots, a_{n-1} \in \{0, \pm 1\}^n)$, the table entry $T(a_0, a_1, \cdots, a_{n-1})$ will be the divisor $a_0 D_0 + a_1 \phi(D_0) + \cdots + a_{n-1} \phi^{n-1}(D_0)$.

Mathematically, for a divisor D and for every $(a_0, a_1, \cdots, a_{n-1}) \in \{0, \pm 1\}^n$ let $\tau_{a_0 a_1 \cdots a_{n-1}}$ be the operator from $J_{F_{q^n}}$ to itself defined by $\tau_{a_0 a_1 \cdots a_{n-1}}(D) = a_0 D + a_1 \phi(D) + \cdots + a_{n-1} \phi^{n-1}(D)$. Our look-up table T contains for every $(a_0, a_1, \cdots, a_{n-1}) \in \{0, \pm 1\}^n$ the value of the operators $\tau_{a_0 a_1 \cdots a_{n-1}}$ at D_0. That is

$$T(a_0, a_1, \cdots, a_{n-1}) \leftarrow \tau_{a_0 a_1 \cdots a_{n-1}}(D_0)$$

The look-up table T will have entries as shown below :

$a_0 a_1 \cdots a_{n-1}$	Divisor
$00 \cdots 0$	$0D_0 + 0\phi(D_0) + \cdots + 0\phi^{n-1}(D_0)$
$00 \cdots 1$	$0D_0 + 0\phi(D_0) + \cdots + \phi^{n-1}(D_0)$
$\cdots$	$\cdots$
$\cdots$	$\cdots$

The look-up table will store 3^n divisors, which is manageable if the extension degree n of the field F_{q^n} is small. In the next subsection we will explore how we can work with a smaller table even for large n.

To compute $\sum_i u_i D_i$ we scan the rightmost column of coefficients $u_A^0, u_A^1, \cdots, u_A^{n-1}$ from (2) and find the corresponding divisor from the table T. We double this divisor and go to scan the column to the left of the column just considered. We compute the divisor corresponding to this column and add it to previously computed divisor and double again. We continue this until the leftmost column. We use the following algorithm to compute mD from the look-up table T.

Algorithm 3.

Input : reduced base-ϕ expansion $\sum_{i=0}^{n-1} u_i \phi^i$ of m and divisor D .

Output : mD.

1. Set $X = T[u_A^{(0)}, u_A^{(1)}, \cdots, u_A^{(n-1)}]$
2. For $j = A - 1$ down to 0
3. $\quad X = 2X$
4. $\quad X = X + T[u_j^{(0)}, u_j^{(1)}, \cdots, u_j^{(n-1)}]$
5. return X.

Proposition 4.1 *Algorithm 3 computes mD.*

Proof : The correctness of the algorithm follows immediately from the following observations. In algorithm 2, $D_i = sign(u_i)\phi(D_0)$. In step 5 of the algorithm 2 we calculate $X = \sum_{i=0}^{n-1} u_A^{(i)} D_i$, which is nothing but $T[u_A^{(0)}, u_A^{(1)}, \cdots, u_A^{(n-1)}]$, the value with which we initialise X in algorithm 3. In step 8 of the algorithm 2 we update X by adding $\sum_{i=0}^{n-1} u_j^{(i)} D_i$ to it. We have $\sum_{i=0}^{n-1} u_j^{(i)} D_i = u_j^{(0)} D_0 + u_j^{(1)} D_1 +$

$\cdots + u_j^{(n-1)} D_{n-1} = T[u_j^{(0)}, u_j^{(1)}, \cdots, u_j^{(n-1)}]$. In step 6, algorithm 3 updates X by adding this value in the table. Algorithm 3 takes care of the signs in the coefficients by allowing them to be in $\{0, \pm 1\}$. So algorithm 2 and algorithm 3 compute the same value. This completes the proof. ■

Proposition 4.2 *Algorithm 3 requires A additions and A doublings.*

Proof : There is only one addition step at step 7, which is in a loop with A iterations. Hence exactly A additions are required to compute mD by algorithm 3. Similarly one can check that exactly A doublings suffices. ■

5 Using Smaller Look-up Tables

In algorithm 3 we are using a table of 3^n divisors, which becomes unmanageable if n is large. In this section we will show how we can work with smaller tables.

Let $w \geq 2$ be a small positive integer. Let $n = s \times w + r$. We observe,

$$
\begin{aligned}
&a_0 D_0 + a_1 \phi(D_0) + \cdots + a_{n-1} \phi^{n-1}(D_0) \\
&\quad = \; a_0 D_0 + a_1 \phi(D_0) + \cdots + a_{w-1} \phi^{w-1}(D_0) \\
&\qquad + a_w \phi^w(D_0) + a_{w+1} \phi^{w+1}(D_0) + \cdots + a_{n-1} \phi^{2w-1}(D_0) \\
&\qquad \vdots \\
&\qquad + a_{(s-1)w} \phi^{(s-1)w}(D_0) + a_{(s-1)w+1} \phi^{(s-1)w+1}(D_0) + \cdots + a_{sw-1} \phi^{sw-1}(D_0) \\
&\qquad + a_{sw} \phi^{sw+1}(D_0) + a_{sw+2} \phi^{sw+2}(D_0) + \cdots + a_{sw+r-1} \phi^{sw+r-1}(D_0)
\end{aligned}
$$

So instead of creating one table of size 3^n, we create $s+1$ tables $T_0, T_1, \cdots T_s$, each of size 3^w. In T_0 we store $a_0 D_0 + a_1 \phi(D_0) + \cdots + a_{w-1} \phi^{w-1}(D_0)$ for all possible choices of $a_0, a_1, \cdots, a_{w-1}$ in $\{0, \pm 1\}$. Similarly in T_1 we store $a_0 \phi^w(D_0) + a_1 \phi^{w+1}(D_0) + \cdots + a_{w-1} \phi^{2w-1}(D_0)$ for all possible choices of $a_0, a_1, \cdots, a_{w-1}$ in $\{0, \pm 1\}$ and so on. In general, the table T_i stores $a_0 \phi^{(i-1)w}(D_0) + a_1 \phi^{(i-1)w+1}(D_0) + \cdots + a_{w-1} \phi^{(i+1)w-1}(D_0)$ for all $a_0, a_1, \cdots, a_{w-1} \in \{0, \pm 1\}^w$. Note that, the combined storage required for all the tables is $(s+1)3^w$ which is much less than $3^n = 3^{ws+r}$. Also, note that the table T_s is not required if $r = 0$. If $r \neq 0$ then it can be made smaller by making it to contain $a_0 \phi^{sw}(D_0) + a_1 \phi^{sw+1}(D_0) + \cdots + a_{r-1} \phi^{sw+r-1}(D_0)$ for all possible choices of $a_0, a_1, \cdots, a_{r-1}$ in $\{0, \pm 1\}$. In that case table T_s will contain only 3^r divisors thereby reducing the cumulative table size. In the following algorithm, we have not incorporated this modification and we will refer to table T_s by choosing the $a_i = 0$ for $i \geq r+1$. If T is the table used by algorithm 3, we have,

$$
\begin{aligned}
T[a_0, a_1, \cdots, a_{n-1}] &= T_0[a_0, a_1, \cdots, a_{w-1}] + T_1[a_w, a_{w+1}, \cdots, a_{2w-1}] + \cdots + \\
&\quad T_{s-1}[a_{(s-1)w}, a_{(s-1)w+1}, \cdots, a_{sw-1}] + T_s[a_{sw}, a_{sw+1}, \cdots, a_{sw+r-1}, 0, \cdots, 0]
\end{aligned}
$$

After construction of the tables in a precomputation step we can compute the scalar multiplication by the following algorithm.

Algorithm 4.

Input : reduced base-ϕ expansion $\sum_{i=0}^{n-1} u_i \phi^i$ of m, n, w, s, r and divisor D .

Output : mD.

1. Set $X = T_s[u_A^{sw}, u_A^{sw+1}, \cdots, u_A^{sw+r-1}, 0, \cdots, 0]$
2. For $i = s-1$ down to 0
3. $\qquad X = X + T_i[u_A^{iw}, u_A^{(i+1)w+1}, \cdots, u_A^{iw-1}]$
4. For $j = A-1$ down to 0
5. $\qquad X = 2X$
6. $\qquad X = X + T_j[u_j^{sw}, u_j^{sw+1}, \cdots, u_j^{sw+r-1}, 0, \cdots, 0]$

```
    For i = s − 1 down to 0
        X = X + T_j[u_j^{iw}, u_j^{iw+1}, ..., u_j^{(i+1)w−1}]
return X.
```

Proposition 5.1 ***Proposition*** *Algorithm 4 computes the scalar multiplication.*

Proof : Comparing algorithm 3 and 4, we observe that they are the same except the table look-up part. Algorithm 3 computes $T[u_j^{(0)}, u_j^{(1)}, \cdots, u_j^{(n-1)}]$ from the table T. Algorithm 4 computes the same thing part-wise, by adding divisors obtained from look-up tables of smaller sizes. Hence both the algorithms basically compute the same thing. This proves the correctness of algorithm 4. ∎

Proposition 5.2 *Algorithm 4 requires $(s+1)A$ additions and A doublings (sA additions and A doublings if $r = 0$) .*

Proof : We know that algorithm 3 requires A additions and A doublings. Besides there are $A+1$ table look-ups. In algorithm 4 each of these table look-ups is broken down to $s+1$ smaller table look-ups and s additions. So, algorithm 4 will require $(A+1) \times s$ more additions than algorithm 3. Also, note that if $r = 0$, then the last table and the last addition in each iteration would not be required. So, only sA addition will suffice.Hence algorithm 4 needs $A + s(A+1)$ additions. The number of doublings in algorithm 4 remains the same as that in algorithm 3, which can be easily checked. ∎

5.1 Advantage of Normal basis

If the underlying field is represented by a normal basis, then we can work with a still smaller table. Let q be a prime. A field F_{q^m} is said to have a normal basis if it has a basis of the form $\{\alpha, \alpha^q, \cdots, \alpha^{q^{m-1}}\}$. Any element of the field can be represented as $x = \sum_{j=0}^{m-1} a_j \alpha^{q^j}$ or briefly as an ordered m-tuple $x = (a_0, \cdots, a_{m-1})$. In the field F_{q^m}, which is of characteristic q, for any $y, z \in F_{q^m}$, $(y+z)^q = y^q + z^q$. Hence we have,$\phi(x) = x^q = (\sum_{j=0}^{m-1} a_j \alpha^{q^j})^q = (\sum_{j=0}^{m-1} a_j \alpha^{q^{j+1}})$. Thus if x is represented by the tuple $(a_0, \cdots, a_{m-1})$ then $\phi(x)$ can be represented by $(a_{m-1}, a_0, \cdots, a_{m-2})$, as $\alpha^{q^m} = 1$. With a normal basis, $\phi(x) = x^q$ can be computed from x by a shift operation only and hence can be efficiently computed online. We observe that,

$$\begin{aligned} T_1[a_0, a_1, \cdots, a_{w-1}] &= \phi^w(T_0[a_0, a_1, \cdots, a_{w-1}]; \\ T_2[a_0, a_1, \cdots, a_{w-1}] &= \phi^w(T_1[a_0, a_1, \cdots, a_{w-1}]) = \phi^{2w}(T_0[a_0, a_1, \cdots, a_{w-1}]) \end{aligned}$$

and so on. Hence we only need to precompute and store the table T_0. Others can be efficiently computed from it during run-time.

Let T' be a table consisting of the values of $a_0 D_0 + a_1 \phi(D_0) + \cdots + a_{w-1}\phi^{w-1}(D_0)$ for all possible choices of $(a_0, a_1, \cdots, a_{w-1})$ in $\{0, \pm 1\}^w$. Then we have,

$$\begin{aligned} a_0 D_0 + a_1 D_1 + \cdots + a_{n-1} D_{n-1} &= T'[a_0, a_1, \cdots, a_{w-1}] + \phi^w(T'[a_w, a_{w+1}, \cdots, a_{2w-1}]) \\ &\quad + \cdots + \phi^{(s-1)w}(T'[a_{(s-1)w}, a_{(s-1)w+1}, \cdots, a_{sw-1}]) \\ &\quad + \phi^{sw}(T'[a_{sw}, a_{sw+1}, \cdots, a_{sw+r-1}, 0, \cdots, 0]). \end{aligned}$$

In other words, we can compute $a_0D_0 + a_1D_1 + \cdots + a_{n-1}D_{n-1}$ as follows. Let

$$\left.\begin{array}{lcl} X_1 & = & \phi^w(T'[a_{sw}, \cdots, a_{r+sw-1}, 0, \cdots, 0]) \\ X_2 & = & \phi^w(X_1 + T'[a_{(s-1)w}, \cdots, a_{sw-1}]) \\ \vdots & \vdots & \vdots \\ X_s & = & \phi^w(X_{s_1} + T'[a_w, \cdots, a_{2w-1}]) \\ X_{s+1} & = & X_s + T'[a_0, \cdots, a_{w-1}] \end{array}\right\} \quad (3)$$

Let $a_0D_0 + a_1D_1 + \cdots + a_{n-1}D_{n-1} = X_{s+1}$. Thus, when the underlying field admits a normal basis, the following modified version of algorithm 4 can be used.

Algorithm $4'$.

Input : reduced base-ϕ expansion $\sum_{i=0}^{n-1} u_i\phi^i$ of m, n, w, s, r and divisor D .
Output : mD.

1. Set $X = \phi^w(T'[u_A^{sw}, u_A^{sw+1}, \cdots, u_A^{sw+r-1}, 0, \cdots, 0])$
2. For $i = s-1$ down to 0
3. $\quad\quad X = \phi^w(X + T'[u_A^{iw}, u_A^{iw+1}, \cdots, u_A^{(i+1)w-1}])$
4. For $j = A-1$ down to 0
5. $\quad\quad X = 2X$
6. $\quad\quad Y = \phi^w(T'[u_j^{sw}, u_j^{sw+1}, \cdots, u_j^{sw+r-1}, 0, \cdots, 0])$
7. $\quad\quad$ For $i = s-1$ down to 0
8. $\quad\quad\quad\quad Y = Y + T'[u_j^{iw}, u_j^{iw+1}, \cdots, u_j^{(i+1)w-1}]$
9. $\quad\quad X = X + Y$
10. return X.

The following result is obvious from the discussions given above.

Proposition 5.3 *Algorithm $4'$ is correct and it requires $(A+1)s + A$ additions and A doublings.* ∎

6 Window Methods

Let $\lambda \geq 2$ be a positive integer. Let $k = \lfloor A/\lambda \rfloor + 1$. We express all u_i occurring in the base-ϕ expansion of m in the base 2^λ. In such expansion we will use the numbers $\{0, \pm 1, \pm 2, \cdots, \pm 2^{\lambda-1}\}$ as digits. Note that to represent an integer we use the set of digits $\{0, 1, \cdots, 2^\lambda - 1\}$, which contains 2^λ elements. The set of digits we consider has $2^\lambda + 1$ elements i.e. we are taking one more element. That is to ensure that the set of digits contains additive inverse of all the digits and zero. The advantage of taking that extra digit will be clear from the following results.

Lemma 6.1 *Any non-negative integer can be represented in base 2^λ using the set $\{0, \pm 1, \pm 2, \cdots, 2^{\lambda-1}\}$ as digits.*

Proof : Let m be any integer. Dividing m by 2^λ we obtain $m = q_0 2^\lambda + a_0$. Since $S = \{0, \pm 1, \pm 2, \cdots, \pm 2^{\lambda-2}, 2^{\lambda-1}\}$ form a residue class modulo 2^λ we can have $a_0 \in S$. Then we divide q_0 by 2^λ to obtain the remainder a_1 in S. We continue this division process until the quotient becomes 0. Let $a_0, \ldots, a_t$ be the remainders so obtained. Then $\sum_{i=0}^{t} a_i 2^{\lambda i}$ is the representation of m in the base 2^λ. ∎

Corollary 6.1 *Any integer can be represented in base 2^λ using the digits $\{0, \pm 1, \pm 2, \cdots, \pm 2^{\lambda-1}\}$.*

Thus, the advantage of including the $-2^{\lambda-1}$ in the set of digits is that if $m = \sum_{i=0}^{t} a_i 2^{\lambda i}$ then $-m = \sum_{i=0}^{t}(-a_i)2^{\lambda i}$, where $-a_i$ is also a digit.

Now, we can write u_i as $(c_0^i + c_1^i 2^{\lambda} + ... + c_k^i 2^{\lambda k})$, where each of $c_i^j \in \{0, \pm 1, \pm 2, \cdots, \pm 2^{\lambda-1}\}$. Thus we have to compute the sum:

$$\left.\begin{array}{lll} u_0 D_0 & = & (c_0^0 + c_1^0 2^{\lambda} + ... + c_k^0 2^{\lambda k}) D_0 \\ u_1 D_1 & = & (c_0^1 + c_1^1 2^{\lambda} + ... + c_k^1 2^{\lambda k}) D_1 \\ \vdots & \vdots & \vdots \\ u_{n-1} D_{n-1} & = & (c_0^{n-1} + c_1^{n-1} 2^{\lambda} + ... + c_k^{n-1} 2^{\lambda k}) D_{n-1} \end{array}\right\} \tag{4}$$

where, $c_j^i \in \{0, \pm 1, \pm 2, \cdots, \pm 2^{\lambda-1}\}$, $D_0 = D$ and $D_i = D_0^{\phi^i}$.

Here again we will use a look-up table. For all choices of $c_0, c_1, \cdots, c_{n-1} \in \{0, \pm 1, \pm 2, \cdots, \pm 2^{\lambda-1}\}$, we compute $c_0 D_0 + c_1 \phi(D_0) + \cdots + c_{n-1}\phi^{n-1}(D_0)$ and store the result in a table $T[]$. That is, given any $(c_0, c_1, \cdots, c_{n-1}) \in \{0, \pm 1, \pm 2, \cdots, \pm 2^{\lambda-1}\}$, $T[c_0, c_1, \cdots, c_{n-1}] = c_0 D_0 + c_1 \phi(D_0) + \cdots + c_{n-1}\phi^{n-1}(D_0)$. The table size will be $(2^{\lambda}+1)^n$.

We can compute the sum $\sum_{i=0}^{n-1} u_i D_i$ by computing the sum of the terms in the right hand side of the equations in (4). Algorithm 5 precisely does that using the look-up table T described above.

Algorithm 5.

Input : reduced base-ϕ expansion $\sum_{i=0}^{n-1} u_i \phi^i$ of m,each u_i expressed in base 2^{λ} and the divisor D .
Output : mD.

1. Set $X = T[c_0^{(k-1)}, c_1^{(k-1)}, \cdots, c_{n-1}^{(k-1)}]$
2. For $j = k-2$ down to 0
3. $\quad X = 2^{\lambda} X$
4. $\quad X = X + T[c_0^{(j)}, c_1^{(j)}, \cdots, c_{n-1}^{(j)}]$
5. return X.

Proposition 6.1 *Algorithm 5 computes mD.*

Proof : Algorithm 5 is the same as algorithm 3 except that instead of scanning one column as in algorithm 3, algorithm 5 scans λ columns at a time. Algorithm 3 initialises X with the value $T[u_A^{(0)}, u_A^{(1)} \cdots, u_A^{(n-1)}]$ which is equal to the value in table T corresponding to the bits in last column of (2). Observe that the last λ columns of (2) are same as last column of (4). For example $a_{A-\lambda-1}^0 + a_{A-\lambda}^0 2 + \cdots + a_{A-1}^0 2^{\lambda} = c_{k-1}^0$. Algorithm 5 initialises X with the value $T[c_0^{(k-1)}, c_1^{(k-1)}, \cdots, c_{n-1}^{(k-1)}]$, which is the value in the corresponding table corresponding to last column of (4). This intialisation takes care of the initialisation step and λ iterations in steps 4,5,6 of algorithm 3 with all computations being done during the precomputation stage. Similarly it is easy to see that each iteration (steps 2,3,4) of algorithm 5 is equivalent to λ iterations(steps 2,3,4) of algorithm 3. Thus algorithm 3 and 4 compute basically the same thing. Hence follows the proposition. ■

Proposition 6.2 *Algorithm 5 requires $k-1$ additions and $(k-1) \times \lambda$ doublings .*

Proof : In algorithm 5 there is only one addition step at step 7, which is in a loop with $k-1$ iterations. Hence the algorithm requires $k-1$ divisor additions. The doubling step (step 6) is also in the same loop. In each step there are λ doublings. Hence the number of doubling required is $(k-1) \times \lambda$. ■

6.1 Using smaller table

We can adopt the same technique in algorithm 3 to reduce the table size. We choose a small integer μ. If $k = t \times \mu + r$ we can work with a look-up table of size $(2^\lambda + 1)^\nu$ and each table look-up of algorithm 5 can be broken down as above to $t + 1$ smaller table look-ups and t additions. As there are k table look-ups in algorithm 5, the number of additions in the modified algorithm will be $(k-1) + kt$ along with $(k-1) \times \lambda$ doublings.

Let $T_0, T_1, \cdots, T_t$ be tables containing $(2^\lambda + 1)^\nu$ entries each. In T_0 we store $c_0 D_0 + c_1\phi(D_0) + \cdots + c_{\mu-1}\phi^{\mu-1}(D_0)$ for all possible choices of $c_0, c_1, \cdots, c_{\mu-1}$ in $\{0, \pm 1, \pm 2, \cdots, \pm 2^{\lambda-1}\}$. Similarly in T_1 we store $c_0\phi^{\mu}(D_0) + c_1\phi^{\mu+1}(D_0) + \cdots + c_{\mu-1}\phi^{2\mu-1}(D_0)$ for all possible choices of $c_0, c_1, \cdots, c_{w-1}$ in $\{0, \pm 1, \pm 2, \cdots, \pm 2^{\lambda-1}\}$ and so on. In general, the table T_i stores $c_0\phi^{(i-1)\mu}(D_0) + c_1\phi^{(i-1)\mu+1}(D_0) + \cdots + a_{\mu-1}\phi^{i\mu-1}(D_0)$ for all $c_0, c_1, \cdots, c_{\mu-1} \in \{0, \pm 1, \pm 2, \cdots, \pm 2^{\lambda-1}\}^w$. Note that, the combined storage required for all the tables is $(t+1)(2^\lambda + 1)^\nu$ which is much less than $(2^\lambda + 1)^k = (2^\lambda + 1)^{t\mu+r}$.

If T is the table used in algorithm 5, then we have,

$$
\begin{aligned}
T[c_0, c_1, \cdots, c_{k-1}] &= T_0[c_0, c_1, \cdots, c_{\mu-1}] + T_1[c_\mu, c_{\mu+1}, \cdots, c_{2\mu-1}] \\
&\quad + \cdots + T_{t-1}[c_{(t-1)\mu}, c_{(t-1)\mu+1}, \cdots, c_{t\mu-1}] + T_t[c_{t\mu}, c_{t\mu+1}, \cdots, c_{t\mu+r-1}, 0, \cdots, 0].
\end{aligned}
$$

Again, we note here that, the table T_t is not required if $r = 0$.

With this set of tables at our disposal we can compute the scalar multiplication by the algorithm 6.

Algorithm 6.

Input : reduced base-ϕ expansion $\sum_{i=0}^{n-1} u_i\phi^i$ of m, n, k, μ, t, r and divisor D .

Output : mD.

1. Set $X = T_t[c_k^{t\mu}, u_k^{t\mu+1}, \cdots, u_k^{t\mu+r-1}, 0, \cdots, 0]$
2. For $i = t-1$ down to 0
3. $\quad X = X + T_i[c_k^{i\mu}, c_k^{i\mu+1}, \cdots, c_k^{(i+1)\mu-1}]$
4. For $j = k-1$ down to 0
5. $\quad X = 2X$
6. $\quad X = X + T_t[c_j^{t\mu}, c_j^{t\mu+1}, \cdots, c_j^{t\mu+r-1}, 0, \cdots, 0]$
7. $\quad$ For $i = t-1$ down to 0
8. $\quad\quad X = X + T_i[c_j^{iw}, u_j^{iw+1}, \cdots, u_j^{(i+1)w-1}]$
9. return X.

From the above discussion the following results are obvious.

Proposition 6.3 *Algorithm 6 is correct and it needs $(k-1)+kt$ divisor additions along with $(k-1)\times\lambda$ doublings($(k-1)(t+1)$ additions if $r = 0$).*

6.2 Using Fields with Normal Bases

If the underlying field admits a normal basis, here also we can save a lot in terms of table size. That is because in a field with a normal basis, computation of $\phi(.)$ is almost free and by constructing just the table T_0 we can construct the entries of the other tables from it online. As in the discussion before Algorithm 4$'$, we have,

$$
\begin{aligned}
T_1[c_0, c_1, \cdots, c_{\mu-1}] &= \phi^\mu(T_0[c_0, c_1, \cdots, c_{\mu-1}]) \\
&= T_2[c_0, c_1, \cdots, c_{\mu-1}] \\
&= \phi^\mu(T_1[c_0, c_1, \cdots, c_{w-1}]) \\
&= \phi^{2\mu}(T_0[c_0, c_1, \cdots, c_{\mu-1}])
\end{aligned}
$$

and so on. Hence we only need the table T_0 . We name it as T'. Thus in table T' we needto store only $(2^\lambda+1)^\mu$ divisors.

We can compute the divisor corresponding to $c_0D_0 + c_1D_1 + \cdots + c_{n-1}D_{n-1}$ as follows. Let

$$\left.\begin{array}{lcl} X_1 & = & \phi^\mu(T'[c_{t\mu}, \cdots, c_{r+t\mu-1}, 0, \cdots, 0] \\ X_2 & = & \phi^\mu(X_1 + T'[c_{(t-1)\mu}, \cdots, c_{t\mu-1}] \\ \vdots & \vdots & \vdots \\ X_t & = & \phi^\mu(X_{t_1} + T'[c_\mu, \cdots, c_{2\mu-1}] \\ X_{t+1} & = & X_t + T'(c_0, \cdots, c_{\mu-1} \end{array}\right\} \quad (5)$$

Let $a_0D_0 + a_1D_1 + \cdots + a_{n-1}D_{n-1} = X_{t+1}$

Hence for a field with a normal basis we can resort to the following algorithm to compute mD.

Algorithm $6'$.

Input : reduced base-ϕ expansion $\sum_{i=0}^{n-1} u_i\phi^i$ of m, n, λ, μ, t, r and divisor D .

Output : mD.

1. Set $X = \phi^\mu(T'[c_k^{t\mu}, c_k^{t\mu+1}, \cdots, c_k^{t\mu+r-1}, 0, \cdots, 0])$
2. For $i = t-1$ down to 0
3. $\quad X = \phi^\phi(X + T'[c_k^{i\mu}, c_A^{i\mu+1}, \cdots, c_k^{(i+1)\mu-1}])$
4. For $j = k-1$ down to 0
5. $\quad X = 2X$
6. $\quad Y = \phi^\mu(T'[c_j^{t\mu}, c_j^{t\mu+1}, \cdots, c_j^{t\mu+r-1}, 0, \cdots, 0])$
7. $\quad$ For $i = t-1$ down to 0
8. $\quad\quad Y = Y + T'[c_j^{i\mu}, c_j^{i\mu+1}, \cdots, c_j^{(i+1)\mu-1}]$
9. $\quad X = X + Y$
9. return X.

The following proposition follows from the discussion at the beginning of this subsection.

Proposition 6.4 *Algorithm $6'$ is correct and it needs $(k-1)+kt$ divisor additions along with $(k-1)\times\lambda$ doublings.* ∎

7 Comparison

In this section, we will summarize the theoretical efficiency of the discussed algorithms. Let C be a hyperelliptic curve of genus g(=2,3 or 4) defined over a finite field F_q. It is known that $\#J(F_{q^n})$ is approximately q^{ng}. So we may assume that $m \approx q^{ng}$. Under suitable assumptions (see [2] for details) in the reduced base-ϕ expansion of $m = \sum_{i=0}^{n-1} u_i\phi^i$, each u_i lies in $(-q^g, q^g]$ and $A = max\{\lfloor \log_2 |u_i| \rfloor\}_{i=0}^{n-1} \leq g\log_2 q$. Let $A = g\log_2 q$.

In the worst case the generic double-and-add algorithm requires $nA+1$ additions and nA doublings. Let us look at the complexity of these algorithms under that condition.

Table 1	Algorithm	ADDITION	DOUBLING	TABLE SIZE
	Double-and-add	$nA+1$	nA	...
	Algorithm 2	$nA+n$	A	...
	Algorithm 3	A	A	3^n
	Algorithm 4	$(s+1)A$	A	$(s+1)3^w$
	Algorithm 4′	$(s+1)A$	A	3^w
	Algorithm 5	$k-1$	$(k-1)\lambda$	$(2^\lambda+1)^n$
	Algorithm 6	$k-1+kt$	$(k-1)\lambda$	$(t+1)\times(2^\lambda+1)^\mu$
	Algorithm 6′	$k-1+kt$	$(k-1)\lambda$	$(2^\lambda+1)^\mu$

To appreciate the results better let us look at some concrete examples.

The following example appears in [2]. Let $q = 131$ and $n = 13$. Consider the hyperelliptic curve of genus 2 given by the equation $C : y^2 = x^5 + x^3 + 1$ defined over F_{131}. The characteristic polynomial of the q^{th} power endomorphism of $J(F_{q^n})$ is

$$\chi(T) = T^4 - 7T^3 + 42T^2 - 917T + 17161$$

Using this polynomial we can find $\#J(F_{q^n})$, which is seen to be divisible by a 159-bit prime. Hence we assume m to be an integer of 159 bits. Also, as stated above the coefficients u_i's in the reduced base-ϕ expansion of m will lie in $(-131^2, 131^2]$= (-17161, 17161]. So $|u_i|$'s are of at most 14 bits. Also $A \leq g\log_2 q = 2\log_2 131 = 14$.

A F_{q^n}-divisor of the curve is of the form $(x^2 + u_1x + u_0, v_1x + v_0)$, where u_1, u_0, v_1, v_0 are in F_{q^n}. Each field element of F_{q^n} is of 91 bits. Hence a divisor will occupy $91 \times 4 = 364$ bits in memory.

Let [A] stand for amount of computation required for a divisor addition and [D] for that for a divisor doubling. Recently, Lange in [8] has developed inversion-free addition and doubling formula for hyperelliptic curve of genus 2. It is reported in [8] that, in the most general and most frequent case, a divisor addition takes 47 multiplications and 4 squarings of the underlying field elements, where as a divisor doubling takes 40 multiplications and 6 squarings. We assume that time complexity of a squaring is 0.8 times that of a multiplication. That is if [M] stands for the time taken for multiplication of a field element with another and [S] stands for that of a squaring, then [M] = 0.8[S]. So, we can say [A] = 50.2[M] and [D] = 44.8[M].

For a scalar multiplication, the generic double-and-add algorithm(Algorithm 1) takes $nA + 1$ additions and nA doublings. In the present example it translates into 183 additions and 182 doublings i.e. 183[A]+182[D].Expressing the cost of additions and doublings in term of cost of field multiplication it becomes 17340.2[M].

In algorithm 3 we can compute the scalar multiplication with exactly A additions and A doublings i.e. 14 additions and 14 doublings, which is equivalent to 14[A]+14[D] = 1330[M]. However, the look-up table size is discouraging. One divisor takes 364 bits of storage. This method requires storage for 3^n divisors for $n = 13$. So, it will need 69.15 Mb of space.

From Table 1 we see that the speed-up achieved by algorithm 2 over the generic double-and-add is really significant only when the extension degree is large. Then the difference between nA and A will be huge. The generic double-and-add method and algorithm 2 take almost the same number of additions. But algorithm 2 takes only A doublings when the former takes nA doublings. So the speed-up obtained in algorithm 2 is almost proportional to this difference. To appreciate this difference in performance let us consider another example where the underlying field has a small extension degree. For example, let us consider a curve over the field, F_{q^3}, where q is a 31 bit prime. Note that elements of this field can be of 93 bits, about the same as the previous case. Here a divisor of a genus 2 curve will occupy $4 \times 93 = 372$

bits. $A \leq g \log_2 q = 62$ and $n = 3$. *For a hyperelliptic curve over such a field, the performance of the algorithms is given by the parenthesized entries in Table 2.*

We observe that in fields with large extension degree algorithm 5 is the most efficient; the amount of storage required is prohibitive. Also, in such a situation the table look-up will become an expensive operation. Under a general set up, algorithm 4 and 6 are more attractive. They require less storage but are significantly faster than algorithm 2. Particularly algorithm 6 with $\mu = 2$ and $\lambda = 2$ is certainly the most attractive choice. If the underlying field has a normal basis representation then algorithms $4'$ and $6'$ are certainly much ahead of the others. The look-up tables are small enough to be implemented on smart cards.

If the extension degree of the underlying field is small, then we can see that many of the above conclusions will cease to hold good. First, we observe that, in that scenario algorithm 2 is not as efficient as it seemed in the previous case (the speed up is only 29 percent compared to 41 percent in previous case). Further, Algorithm 4 which required discouraging amount of space when n is large now takes only 1.28 KB of space. In efficiency, although it has become costlier than the previous situation, still it is 52 percent faster than algorithm 2.

When the extension degree n is small, algorithm 4,$4'$, 6, $6'$ are no more relevant. However, algorithm 5, which in the previous scenario required prohibitive amount of memory now takes only 5.7 kilobytes of memory and still emerges as the fastest one.

To sum up, if the extension degree of the field is high, algorithm 4, $4'$, 6 or $6'$ are the best performers. One should choose one of them based on one's speed requirement and available resources. If the extension degree is small, then one of the algorithms 3 or 5 should be chosen. We summarize this discussion in the following table.

Table 2

Algorithm	ADDITION	DOUBLING	COMPUTATIONS	TABLE SIZE
Double-and-add	183 (187)	182(186)	17340[M] (17720)	...
Algorithm 2	195 (189)	14 (62)	10416[M](12265)	...
Algorithm 3	14 (62)	14 (62)	1330[M](5890)	69.15MB(1.2KB)
Algorithm 4(w=2)	98	14	5547[M]	2.79KB
Algorithm 4(w=3)	70	14	4141[M]	5.99KB
Algorithm $4'$(w=2)	98	14	5547[M]	0.39KB
Algorithm $4'$(w=3)	70	14	4141[M]	1.19KB
Algorithm 5($\lambda = 2$)	6 (30)	12 (60)	838[M] (4194)	50GB(5.7KB)
Algorithm 6($\mu = 2, \lambda = 2$)	27	12	1893[M]	4.4KB
Algorithm 6($\mu = 3, \lambda = 2$)	24	12	1742[M]	16.6KB
Algorithm $6'$($\mu = 2, \lambda = 2$)	27	12	1893[M]	1.1KB
Algorithm $6'$($\mu = 3, \lambda = 2$)	24	12	1742[M]	5.55KB

8 Conclusion

In this article, we have devised several look-up table based algorithms for Frobenius map based scalar multiplication in the Jacobian of a hyperelliptic curve. For a field with large extension degree, the new table look-up methods are much more efficient efficient than the existing algorithms. Particularly, if the underlying field is represented by a normal basis, the storage requirement is low enough to be implemented on smart cards. For fields with low extension degree the window method is found to be most lucrative in terms of speed and the general table-look up method required the least amount of storage. To sum up, the article offers a spectrum of opportunity to the implementor, who can choose one of the algorithms based on his/her requirement and available resources.

References

[1] D. G. Cantor. Computing in the Jacobian of a Hyperelliptic curve. In *Mathematics of Computation*,volume 48, pages 95-101, 1987.

[2] Y. J. Choie, J. W. Lee. Speeding Up theScalar Multiplication in the Jacobian of Hyperelliptic Curves Using Frobenius Map, *INDOCRYPT 2002*, LNCS 2551, Springer Verlag 2002, pp 285-295.

[3] C. Gunther, T. Lange, A. Stein. Speeding Up the Arithmetic on Koblitz Curves of Genus Two, *Selected Areas in Cryptography, SAC 2001*, LNCS, pp. 106-117, 2001.

[4] N. Koblitz. Hyperelliptic Cryptosystems. In Ernest F. Brickell, editor, *Journal of Cryptology*, pages 139–150, 1989.

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

[6] N. Koblitz. CM Curves with Good Cryptographic Properties *Advances in Cryptology - Crypo'91*, LNCS 576, pp. 279-287,Springer Verlag 1992.

[7] T. Lange. Efficient Arithmetic on Hyperelliptic Koblitz Curve, Preprint 2001.

[8] 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/* . Also, see Report 2002/147, 2002/153.

[9] W. Meier, O. Staffelbach. Efficient Multiplication on Certain Nonsupersingular Elliptic Curves. In *Advances in Cryptology, Crypto'92* ,LNCS 740, pp.333-344, Springer Verlag.

[10] 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

[11] 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.

[12] J. Solinas. An improved Algorithm on a Family of Elliptic Curves. In *Advances in Cryptology - Crpto'97*,LNCS 1294 , pp.357-371 ,Springer-Verlag.

[13] J. Solinas Efficient Arithmetic on Koblitz Curves. Technical Report 99-09, University of Waterloo. Available at *http://www.cacr.math.uwaterloo.ca* .