Software Implementations of Elliptic Curve Cryptography

Aneel Murari
ECE679
Oregon State university

June 9, 2003

Abstract

Elliptic Curve Cyrptography has gained a lot of significance in recent times. This is mostly due to the small key sizes associated with Elliptic Curve Cryptograhic systems. This paper presents a study of various algorithms for performing underlying field arithmetic and point representation useful in software implementations of Elliptic curve cyptography over prime fileds as well as binary fields.

1. Introduction

Elliptic curve cryptography was proposed independently by Neal Kolblitz and Victor Miller in 1985. Elliptic curves defined over $GF(p)$ and $GF(2^n)$ are generally used in cryptography. Elliptic curves over $GF(2^n)$ are used more widely due to existance of efficient algorithms for $GF(2^n)$ arithmetic.

An Elliptic Curve over $GF(p)$ is defined by the set of points (x,y) that satisfy the equation $y^2 = x^3 + ax + b$ where a , $b \in GF(p)$ and $p > 3$ and $4a^3 + 27b^2 \neq 0$ together with a special point O called the point at infinity. A point on an elliptic curve E can be seen as a point on the two-dimensional plane satisfying the elliptic curve equation which is of the above form. The set of such points satisfying an elliptic curve equation forms a group under addition operation. The point O or the point at infinity serves as an identity element in the group. The addition operations in the group are given the following rules:

$$O + O = O$$
$$(x,y) + O = (x,y)$$
$$(x,y) + (x,-y) = 0$$

Addition of two points with $x_1 \neq x_2$

$$(x_1,y_1) + (x_2,y_2) = (x_3,y_3)$$
$$\lambda = (y_2 - y_1)(x_2 - x_1)^{-1}$$
$$x_3 = \lambda^2 - x_1 - x_2$$
$$y_3 = \lambda(x_1 - x_3) - y_1$$

Doubling of a point with $x_1 \neq 0$

$$(x_1,y_1) + (x_2,y_2) = (x_3,y_3)$$
$$\lambda = (3x_1^2 + a)(2y_1)^{-1}$$
$$x_3 = \lambda^2 - 2x_1$$
$$y_3 = \lambda(x_1 - x_3) - y_1$$

An Elliptic Curve over $GF(2^p)$ is defined similar to the one in $GF(p)$. An Elliptic curve E over $GF(2^n)$ is given by set of points (x,y) that satisfy the equation $y^2 + xy = x^3 + ax^2 + b$ where a , $b \in GF(2^n)$ and $b \neq 0$ together with a special point O called the point at infinity. Similar to its counterpart in $GF(p)$, the set of points forms a group under addition operation. The addition rules for the group are as follows:

$$O + O = O$$
$$(x,y) + O = (x,y)$$
$$(x,y) + (x,x+y) = 0$$

Addition of two points with $x_1 \neq x_2$

$$(x_1,y_1)+(x_2,y_2)=(x_3,y_3)$$
$$\lambda=(y_2-y_1)(x_2-x_1)^{-1}$$
$$x_3=\lambda^2+\lambda+x_1+x_2+a$$
$$y_3=\lambda(x_1+x_3)+x_3+y_1$$

Doubling of a point with $x_1 \neq 0$

$$(x_1,y_1)+(x_2,y_2)=(x_3,y_3)$$
$$\lambda=x_1+(y_1)(x_1)^{-1}$$
$$x_3=\lambda^2+\lambda+a$$
$$y_3=x_1^2+(\lambda+1)x_3$$

The operation of adding a point several times to itself, i.e. the calculation of $Q = kP$ where k is an integer and $P \in E$, is called the elliptic curve scalar multipilcation. This scalar multiplication forms the basic for elliptic curve crytography. An elliptic curve discrete logarithm problem can be defined as follows. If E is an elliptic curve over F_{p^r} and P is on $E(F_{p^r})$, then the discrete logarithm problem on $E(F_{p^r})$ is the problem, given a point $P \in E(F_{p^r})$, of finding an integer $k \in Z$ such that $kP = Q$ if such a k exists.

Efficinet implementations of Elliptic curve cryptography can be classified at two levels. At the elliptic curve group level, fast implementations of scalar point multipliccation is crucial. This problem is analogous to the problem of exponentiation in multiplicative group. And at the underlying finite field level, fast implementations of multipication, Inversion and reduction are considered important. Field multiplication is the next costly operation after inversion. Various techniques like projective coordinates trade field multiplication to field inversion. Hence, it is desirable to have efficient implementations of field multiplication and reduction.

In the following sections we present various algorithms for performing field multiplications, inversion and reduction over $GF(P)$ and $GF(2^n)$. We also present algorithms for scalar point multiplication. We also explore representations of points on Elliptic curves using the projection coordinates and how certain type of projection coordinate suits to certain platforms for increased efficiency.

2. Arithmetic in Prime Field

This section presents algorithms for performing arithmetic in F_p in software. Without loss of generality, we assume the implementation platform for the algorithms is 32 bit. The bits a word W are numbered from 0 to 31, with the rightmost bit of W designated as bit 0. The elements of F_p are the integers between 0 and $p-1$, written in binary. Let $m = \lceil log_2 p \rceil$ and $t = \lceil m/32 \rceil$. In software, we can store a field element a in an array of t 32-bit words: $a = (a_{t-1}, ..., a_1, a_0)$.

2.1 Addition

The below given algorithm calculates the sum of two numbers mod p by first finding the sum word-by-word and then subtracting p if the result exceeds $p-1$.Each word addition produces a 32-bit sum and a 1-bit carry digit which is added to the next higher-order sum. The "Add" in step 1 and "AddWithCarry" in step 2 of the algorithm manage the carry digit. On processors such as the Intel Pentium family which offer an "Add with Carry" instruction as part of the instruction set, these may be fast single-instruction operations.

```
Modular Addition:
Input: A modulus $p$ and integers $a$, $b \in [0, p-1]$.
Output: $c = (a+b)$ mod $p$

1. $c_0 \leftarrow$ Add($a_0$,$b_0$)
2. For $i$ from 1 to $t-1$ do: $c_i \leftarrow$ AddWithCarry($a_i$,$b_i$)
3. If the carry bit is set, then subtract $p$ from $c = (c_{t-1}, ..., c_1, c_0)$
4. if $c \geq p$ then $c \leftarrow c-p$
5. Return($c$)
```

2.2 Subtraction

Implementation of modular subtraction is similar to the implementation of modular addition except the fact that the carry in addition is treated as borrow in subtraction. As with the case of addition, the first two steps are generally fast but are faster if the processors has specialized instructions for them.

```
Modular Subtraction
Input: A modulus $p$ and integers $a, b \in [0, p-1]$.
Output: $c = (a-b)$ mod $p$

1. $c_0 \leftarrow$ Subtract($a_0$,$b_0$)
2. For $i$ from 1 to $t-1$ do: $c_i \leftarrow$ SubtractWithBorrow($a_i$,$b_i$)
3. If the carry bit is set, then add $p$ from $c$ = $(c_{t-1}, ..., c_1, c_0)$
4. Return($c$)
```

2.3 Modular Multipication and squaring

Algorithm below specifies the standard multiplication algorithm for multiplying two integers a and b.

```
Integer Multiplication:
Input: Integers $a, b \in [0, p-1]$.
Output: $c = a \cdot b$

1. $r_0 \leftarrow 0, r_1 \leftarrow 0, r_2 \leftarrow 0,$
2. for $k$ from 0 to $2(t-1)$ do
    2.1 For each element of $\{(i,j)|i+j=k, 0 \leq i,j < t\}$ do
        $(uv) = a_i \cdot b_j$
        $r_0 \leftarrow$ Add($r_0, v$)
        $r_1 \leftarrow$ AddWithCarry($r_1, u$)
        $r_2 \leftarrow$ AddWithCarry($r_2, 0$)
    2.2 $c_k \leftarrow r_0, r_0 \leftarrow r_1, r_1 \leftarrow r_2, r_2 \leftarrow 0$
3. $c_{2t-1} \leftarrow r_0$
4. Return($c$)
```

Direct modification of the above Multiplication algotirhm leads to the following squaring algorithm. The number of multiplications are roughly half from the above algorithm.

```
Squaring Algorithm:
Input: Integer $a \in [0, p-1]$.
Output: $c = a^2$

1. $r_0 \leftarrow 0, r_1 \leftarrow 0, r_2 \leftarrow 0,$
2. for $k$ from 0 to $2(t-1)$ do
    2.1 For each element of $\{(i,j)|i+j=k, 0 \leq i,j < t\}$ do
        $(uv) = a_i \cdot a_j$
        If ($i$ ¡ $j$) then $(uv) \ll 1$, $r_2 \leftarrow$ AddWithCarry($r_2, 0$)
        $r_0 \leftarrow$ AddWithCarry($r_0, v$)
        $r_1 \leftarrow$ AddWithCarry($r_1, u$)
        $r_2 \leftarrow$ AddWithCarry($r_2, 0$)
    2.2 $c_k \leftarrow r_0, r_0 \leftarrow r_1, r_1 \leftarrow r_2, r_2 \leftarrow 0$
3. $c_{2t-1} \leftarrow r_0$
4. Return($c$)
```

2.4 Modular Reduction

Modular reduction using Barrets method [1] is generally considered to be fast reduction technique for modular reduction. The algrithm is as follows:

```
Barrett Reduction:
Input: $b > 3$, $p, k = \lfloor log_b p \rfloor + 1$, $0 \leq x < b^{2k}, \mu = \lfloor b^{2k}/p \rfloor$
Output: $x$ mod $p$
```

```
1. $q \leftarrow \lfloor \lfloor x.b^{k-1} \rfloor \cdot \mu / bk+1 \rfloor$
2. $r \leftarrow$ ($x$ mod $b^{k+1}$ ) - ($q \cdot p$ mod $b^{k+1}$).
3. If $r < 0$ then $r \leftarrow r + b^{k+1}$
4. While $r \geq p$ do: $r \leftarrow r - p$
5. Return(r)
```

2.5 Binary Inversion Algorithm

The following algorithm is a varient of Extended Euclidian Algorithm which calculates the inverse of field element.

```
Input: Prime $p$, $a \in [1, p-1]$
Output: $a^{-1}$ mod $p$
1. $u \leftarrow a, v \leftarrow p, A \leftarrow 1, C \leftarrow 0$
2. While $u \neq 0$ do
    2.1 While $u$ is even do:
        $u \leftarrow u/2$. If $A$ is even then $A \leftarrow A/2$; else $A \leftarrow (A+p)/2$
    2.2. While $v$ is even do:
        $v \leftarrow v/2$. If $C$ is even then $C \leftarrow C/2$; else $C \leftarrow (C+p)/2$
    2.3 If $u \geq v$ then: $u \leftarrow u - v$, $A \leftarrow A - c$; else: $v \leftarrow v - u, C \leftarrow C - A$
3. return($C$ mod $p$)
```

3. Arithmetic in Binary Field

In this section, we present algorithms for preforming Binary filed arthimetic $GF(2^m)$ where m is prime. Much like the previous section we assume that underlying implementation platfrom has a 32-bit architecture. Of the many representations of $GF(2^m)$ where m is prime, it appears that a polynomial basis representation with a trinomial or pentanomial as the reduction polynomial yields the simplest and the fastest implementation in software. So, we can use the polynomial based representation here in this paper.

Let $f(x) = x^m + r(x)$ be an irreducible binary polynomial of degree m. The elements of $GF(2^m)$ are binary polynomials of degree at most $m-1$ with addition and multiplication performed modulo $f(x)$. A field element $a(x) = a_{m-1}x^{m-1} + ...a_2x^2 + a_1x + a_0$ is associated with the binary vector $a = (a_{m-1}, ..., a_2, a_1, a_0)$ of length m. The addition and subtraction in $GF(2^m)$ is identical to that of prime field addition and subtraction. We will give the reamining Arithmetic algorithms below.

3.1 Multiplication

The classical shift-and-add method for field multiplication is based on the fact that $a \cdot b$ can be rewritten as $a_{m-1}x^{m-1}b + ... + a_1xb + a_0b$. The ith iteration of the algorithm computes x^ib mod $f(x)$ and add the result to the accumlator if $a_i = 1$. It must be noted that $b \cdot x$ mod $f(x)$ can be calculated easily by a left-shift of the vector representation of b, followed by addition of $r(x)$ to b is $b_m = 1$

```
Shift-and-add Field Multiplication:
Input: Binary polynomials $a(x)$ and $b(x)$ of degree at most $m-1$.
Output: $c(x) = a(x) \cdot b(x)$ mod $f(x)$.

1. if $a_0 = 1$ then $c \leftarrow b$; else $c \leftarrow 0$.
2. For $i$ from 1 to $m-1$ do
    2.1 $b \leftarrow b \cdot x$ mod $f(x)$
    2.2 If $a_i = 1$ then $c \leftarrow c + b$.
3. Return(c).
```

while the above stated algorithm is good for hardware implementations in which vector shift can be performed in on clock cycle, it may be not suitable very much for software implementations due to multiple shifts.Hence, we now consider algorithms for field multiplication which multiplies the two polynomial vectors and later reduction mod $f(x)$ is applied to the result. The following algorithm is called the comb method for polynomial multiplication and is based on the observation that if $b(x) \cdot x^k$ has been computed for some $k \in [0, 31]$, then $b(x) \cdot x^{32j+k}$ can be obtained by appending j zero words to the right of the vector representation of $b(x) \cdot x^k$.

```
Left-to-right comb method:
Input: Binary polynomials $a(x)$ and $b(x)$ of degree at most $m-1$.
Output: $c(x) = a(x) \cdot b(x)$.

1. $C \leftarrow 0$
2. For $k$ from 31 downto 0 do
    2.1 For $j$ from 0 to $t-1$ do
        If the $k$th bit of $A[j]$ is 1 then add $B$ to $c\{j\}$.
    2.2 If $k \neq 0$ then $C \leftarrow C \cdot x$. 3. Return(C).
```

```
Left-to-right comb method with window size = 4:
Input: Binary polynomials $a(x)$ and $b(x)$ of degree at most $m-1$.
Output: $c(x) = a(x) \cdot b(x)$.

1. Compute $B_u = u(x) \cdot b(x)$ for all polynomials $u(x)$ of degree 3.
2. $C \leftarrow 0$.
3. For $k$ from 7 downto 0 do
    3.1 For $j$ from 0 to $t-1$ do
        Let $u = (u_3, u_2, u_1, u_0)$, where $u_i$ is bit $(4k+i)$ of $A[j]$. Add $B_u$ to $C\{j\}$.
    3.2 If $k \neq 0$ then $C \leftarrow C \cdot x^4$.
4. Return(C).
```

Once Multiplication is done, we trun our attention now to reducing the multiplied polynomials by mod $f(x)$. the following algorithm computes $c(x)$ mod $f(x)$ where $c(x)$ is a polynomial of degree atmost $2m-2$. The algorithm is based on the observation that $x^i = x^{i-m}r(x) \pmod{f(x)}$ for $i \geq m$. The idea is that the polynomilas $x^k r(x)$ can be precomputed and hence the algorithm can be faster.

```
Modular Reduction - One bit at a time:
Input: A binary polynomial $c(x)$ of degree atmost $2m-2$.
Output: $c(x)$ mod $f(x)$

1. Compute $u_k(x) = x^k r(x), 0 \leq k \leq 31$.
2. For $i$ from $2m-2$ downto $m$ do
    2.1 If $c_i$ = 1 then
        Let $j = \lfloor (i-m)/32 \rfloor$ and $k = (i-m) - 32j$.
        Add $u_k(x) to C\{j\}$.
3. Return(($c[t-1], \ldots C[1], C[0]$)).
```

The reduction algorithm can further be modified to process it one word at a time customizing it according to the reduction polynomial that is being used.

3.2 Squaring

Unlike polynomial multiplication, squaring a given polynomial can be done much faster because squaring is a linear operation in $GF(2^m)$. i.e., if $a(x) = \sum_{i=0}^{m-1} a_i x^i$, then $a(x)^2 = \sum_{i=0}^{m-1} a_i x^{2i}$. The binary representation of $a(x)^2$ is obtained by inserting a 0 bit between consecutive bits of binary representation of $a(x)$

```
Squaring:
Input: $a \in GF(2^m)$
Output:$a^2$ mod $f(x)$

1. For each byte $v$ = ($v_7, \ldots v_1, v_0$), compute the 16-bit quantity $T(v)$ = ($0, v_7, \ldots 0, v_1, 0, v_0$).
2. For $i$ from 0 to $t-1$ do
    2.1 Let $A[i]$ = ($u_3, u_2, u_1, u_0$) where each $u_j$ is a byte
    2.2 $C[2i] \leftarrow (T(u_1), T(u_0))$, $C[2i+1] \leftarrow (T(u_3), T(u_2))$.
3. Compute $b(x) = c(x)$ mod $f(x)$
4. Return(b).
```

3.3 Inversion

The inverse of a non zero element $a \in GF(2^m)$ can be calculated using a variation of Extended Euclidean Algorithm which is given below:

```
Extended Euclidian Algorithm for Inverse:
Input: $a \in GF(2^m), a \neq 0$
Output: $a^{-1} \bmod f(x)$

1. $b \leftarrow 1, c \leftarrow 0, u \leftarrow a, v \leftarrow f$
2. While degree($u$) $\neq 0$ do
    2.1 $j \leftarrow$ degree($u$) - degree($v$)
    2.2 If $j < 0$ then: $u \leftrightarrow v$, $b \leftrightarrow c$, $j \leftarrow -j$
    2.3 $u \leftarrow u + x^j v$, $b \leftarrow b + x^j c$
3. Return(b)
```

A varient of Almost inverse Algorithm from is given in [6]. For $a \in GF(2^m), a \neq 0$, a pair (b,k) is returned where $ba = x^k \bmod f(x)$. A reduction is then applied to obtain $a^{-1} = bx^{-k} \bmod f(x)$. Below is the algorithm for this.

```
Variation of Almost Inverse Algorithm for inversion:
Input: $a \in GF(2^m), a \neq 0$
Output: $a^{-1} \bmod f(x)$

1. $b \leftarrow 1$, $c \leftarrow 0$, $u \leftarrow a$, $v \leftarrow f$
2. While $x$ divides $u$ do:
    2.1 $u \leftarrow u/x$
    2.2 If $x$ divides $b$ then $b \leftarrow b/x$; else $b \leftarrow (b+f)/x$
3. If $u = 1$ then return(b)
4. If degree($u$) $<$ degree($v$) then:$u \leftrightarrow v$, $b \leftrightarrow c$
5. $u \leftarrow u + v$, $b \leftarrow b + c$
6. Goto step 2.
```

4. Point Representation

Elliptic curve point representation given in section 1 of this paper are called the Affine coordinates of a point. While it is easy to understand the addition and doubling rules of elliptic curve points in Affine coordinates, it is not always the efficient way of representing the points and performing arithmatic in Affine coordinates. For instance, when two points P_1 and P_2 are not equal, general addition of points P_1 and P_2 requires 1 inversion, 2 multiplications and 1 squaring operations. This computing effort can be reduced if the Elliptic curve points are represented using another form of representation called the Projective Coordinates.

Representation of an Elliptic curve point in projective coordinate replaces the inversion operation which is considerably expensive than multiplication with multiplication Operation. There are many forms of projective coordinates proposed. Some of them gives improved efficiency while using with certain type of elliptic curves. in standard projective coordinates, the projective point $(X : Y : Z)$, $Z \neq 0$, corresponds to the affine point $(X/Z, Y/Z)$. The corresponding elliptic curve equation would be $Y^2Z = X^3 - 3XZ^2 + bZ^3$. In jacobian projective coordinates, the projective point $(X : Y : Z)$ corresponds to the affine point $(X/Z^2, Y/Z^3)$ and the corresponding elliptic curve equation is $Y^2 = X^3 - 3XZ^4 + bZ^6$. And in chudnovsky Jacobian coordinates, the point $(X : Y : Z : Z^2 : Z^3)$ corresponds to the jacobian point $(x : y : Z)$.

Point doubling and addition can be done by first converting the projective coordinates to affine coordinates and then the point doubling and addition formulas may be applied which would not neccesiate Inversion operation and finally clearing the dinominators to get back the projective coordinates. Table below shows number of various operations required to perform point doubling and addition in various forms of point representation [1]. It can be observed that Doubling is fastest in Jacobian form where as addition is fastest in mixed affine-jacobian form.

Doubling		General Addition		Mixed coordinates	
$2A \leftarrow A$	$1I, 2M, 2S$	$A + A \leftarrow A$	$1I, 2M, 1S$	$J + A \leftarrow J$	$8M, 3S$
$2P \leftarrow P$	$7M, 3S$	$P + P \leftarrow P$	$12M, 2S$	$J + C \leftarrow J$	$11M, 3S$
$2J \leftarrow J$	$4M, 4S$	$J + J \leftarrow J$	$12M, 4S$	$C + A \leftarrow C$	$8M, 3S$
$2C \leftarrow C$	$5M, 4S$	$C + C \leftarrow C$	$11M, 3S$		

A = Affine, P=Standard Projective, J = jacobian, C = Chudnovsky, I = Inverse, M = Multiplication, S = Subtraction.

5. Point Multiplication

This section describes various scalar point multiplication algorithms. Point multiplication is the most expensive operation in the elliptic curve cyptography schema. The algortims given in this section calculate kP where k is an integer and P is a point on the elliptic curve.

The below given algorithm is the addition version of a basic square and add version of the exponentition algorithm.

```
Binary Method for point multiplication:
Input: $k = (k_{t-1}, ..., k_1, k_0)_2$, $P = GF(2^m)$
Output: $kP$

1. $Q \leftarrow O$
2. For $i$ from $t-1$ downto 0 do
    2.1 $Q \leftarrow 2Q$
    2.2 if $k_i = 1$ then $Q \leftarrow Q+P$
3. Return($Q$).
```

The approximate number of 1's in the binary representation is $m/2$. Hence the above algorithm would take $0.5m$ point additions and m point doublings. If the affine coordinates are used, it would require $3m$ Multiplications and $1.5m$ Inversions. If projective coordinates are used to store Q, it requres only $8.5m$ multiplications and a little overhead in converting the coordinates back to affine coordinates. This is an example where projective coordinates are more efficeint to implement than their affine counterparts.

In elliptic curves if $P = (x, y)$ is a point on the curve, the point $-P$ is $(x, -y)$ if $P \in GF(p)$ or it is equal to $(x, x+y)$ if $P \in GF(2^m)$. Hence point subtraction is as expensive as point addition on an elliptic curve. This property of elliptic curves can be exploited to efficiently record the point multiplication of a point P. In order to exploit this property, we first need to represent the elliptc curve point P in form called the canocnical recording. This representation takes digits from the set $\{1, 0, \overline{1}\}$ to represent a point P. The following table also called the Reitwiesners Algorithm [3] is used to get an efficient canonical recording for a given binary representation of the point.

e_{i+1}	e_i	a_i	f_i	a_i
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	0	0
1	0	1	$\overline{1}$	1
1	1	0	$\overline{1}$	1
1	1	1	0	1

In the above table f gives the final canonical reading for the given point. Once the canonical recording is done, the point multiplication is done by the following algorithm.

```
Binary canonical recording Algorithm:
Input:$P, -P, e$
Output: $Q = eP$

1. Obtain the signed digit recording $f$ of $e$.
2. if $f_k = 1$ then $Q = P$ else $Q = O$
3. for $i = k-1$ downto 0
    3.1 $Q = Q + Q$
    3.2 if $f_i = 1$ then $Q = Q + P$
     if $f_i = \overline{1}$ then $Q = Q - P$
4. Return($Q$)
```

One can further increase the efficiency of the algorithm by considering more than 1 bit at a time using window techniques rather than binary method. Lot of other point multiplication algorithms are also in existance. Among them, one important and efficient algorithm is Montgomery point multiplication algorithm.

The methods mentioned above in this section correspond to point multiplication algorithms for a given arbitrary point. But if we know the point before hand and the point is fixed and if we can spare some memory for precomputation, more efficient point multilication algorithms can be devised. The following two algorithms given are good example of these category of algorithms.

```
Fixed-Base Windowing Method:
Input: Window width $w$, $d = \lceil t/w \rceil$, $k = (k_{d-1}, \ldots, k_1, k_0)2^w$, $P \in GF(2^m)$
Output: $kP$

1. Precompute $P_i = 2^{wi}P$, $0 \leq i \leq d-1$
2. $A \leftarrow O$, $B \leftarrow O$
3. For $j$ from $2^w - 1$ downto 1 do
    3.1 For each $i$ for which $k_i = j$ do: $B \leftarrow B + P_i$
    3.2 $A \leftarrow A + B$
4. Return(A)
```

```
Fixed-Base comb Method:
Input: Window width $w$, $d = \lceil t/w \rceil$, $k = (k_{d-1}, \ldots, k_1, k_0)2^w$, $P \in GF(2^m)$
Output: $kP$

1. Precompute $[a_{w-1}, \ldots, a_1, a_0]P \forall\ (a_{w-1}, \ldots, a_1, a_0) \in Z_2^w$
2. By padding $k$ on the left with 0's if necessary, write $k = K^{w-1} \| \ldots \| K^1 \| K^0$, where each $K^j$ is a bit string of length d.
Let $K_i^j$ denote the $i$th bit of $K^j$
3. $Q \leftarrow O$
4. For $i$ from $d-1$ downto 0 do
    4.1 $Q \leftarrow 2Q$
    4.2 $Q \leftarrow Q + [K_i^{w-1}, \ldots, K_i^1, K_i^0]P$
5. Return(Q)
```

5. Summary

It can be seen from the previous section that for point multiplication, the Fixed-base comb method is the fastest given that the point to be multiplied is know before hand. For a variable point, the point multiplication using the Window based canonical reading method is the fastest. Also, according to the practical results that are provided in [1], using projective coordinates for these methods yields the most optimized algorithms. For window based canonical reading method, using mixed coordinates of Jacobian and Chudnovsky yields the fastest algorithm where as using mixed coordinates with Jacobian and Affine yields fastest algorithm for Fixed-Base comb method. For integer multiplication in prime fields, the algorithm presented in section 2.3 performs well. It is considered faster than the Karasthubha algorithm for integer multiplication. In binary field multiplication, Left-to-Right comb method with windowing yields the fastest algorithm followed by Karatsuba algorithm. For Inversion in binary field, the extended Euclidian algorithm performs well when compared to other algorithms.

[1] and [6] gives the exact timings of the above algorithms implemented on a 32 bit architecture for NIST recommended curves. I can be observed that using projective coordinates instead of affine coordinates can gain significant amount of efficiency mainly due to the ovehead reduced by replacing Inversion with multiplication. It can also be noted that faster algorithms for specific kinds of curves like the kolblitz curves can be disigned and also, enhancing the mentioned algorithms depending on the specific architecture, leads to some performance improvement.

References

[1] M. Brown, D. Hankerson, J. Lopez, and A. Menezes. Software implementations of nist elliptic curves over prime fields. Technical report, 2001.

[2] Ç. K. Koç. Classnotes - elliptic curve cryptosystems, oregon state univerity.

[3] Ç. K. Koç. Classnotes - high-speed implementations of rsa and elliptic curve cryptosystems, oregon state univerity.

[4] Kenneth Giuliani. Elliptic curves over finite fields. Technical report, 6 December 1999.

[5] Yongfei Han, Peng-Chor Leong, Peng-Chong Tan, and Jiang Zhang. Fast algorithms for elliptic curve cryptosystems over binary finite field. Technical report, 1999.

[6] H. Hankerson, J. L. Hernandez, and A. Menezes. *Software implementation of elliptic curve cryptography over binary fields*. Second International Workshop, Worcester, MA, USA. Springer Verlag, LNCS Nr. 1965, August 17–18 2000.

[7] G.-A. Kamendje J. Groschadl, E. Oswald, and R. Posch. Elliptic curve cryptography in practice the concept of the austrian citizen card for e-government applications. Technical report, 2000.

[8] BBN Technologies Jeffrey L. Vagle. A gentle introduction to elliptic curve cryptography. Technical report, Nov 21 2000.

[9] Don. B. Johnson and Alfred J Menezes. Elliptic curve dsa (ecdsa): an enhanced dsa. Technical report, 1997.

[10] M.MATSUI T.HASEGAWA, J.NAKAGIMA. A small and fast software implementation of elliptic curve cryptosystems over gf(p) on a 16 bit microcomputer. Technical report, January 1999.

[11] E. De Win, A. Bosselears, S. VAndenberghe, P. De Gersem, and J. Vandewalle. A fast software implementation for arithmetic operations in $gf(2^n)$. Technical report, 1996.