International Journal of Computer & Information Technologies (IJOCIT)
www.ijocit.org & www.ijocit.ir
ISSN = 2345-3877

Classification and Comparison of Scalar Multiplication Algorithms in Elliptic Curve Cryptosystems

Saeed Rahimi[1], Abdolrasoul Mirghadri[2]
Department of cryptography, Emam Hosein university
Tehran, Iran
sae.rahimi@gmail.com[1] , mirghadri@yahoo.com [2]

Abstract: The most popular public-key cryptography systems nowadays are RSA and Elliptic Curve Cryptography (ECC). ECC is a type of public-key cryptosystem which uses the additive group of points on a nonsingular elliptic curve as a cryptographic medium. The basic operation in most elliptic curve cryptosystems is a scalar multiplication. Scalar Multiplication is the costliest operation among all in ECC which takes 80% of key calculation time on Elliptic curve calculation. Hence Scalar multiplication is the most time- consuming operation in ECC protocols. Scalar multiplication (or point multiplication) is the operation of calculating an integer multiple of an element in additive group of elliptic curve. in this paper, we classify and compare proposed scalar multiplication algorithms and compute their executing time.

Keywords: ECC, Cryptography, Elliptic Curve, Scalar Multiplication

1. Introduction

Compared with other approaches, such as the widely used Rivest-Shamir-Adleman (RSA) public key system, the Elliptic Curve Cryptography public key system is more appropriate for embedded systems with limited power and memory resources.

ECC is considered much more suitable than other public-key algorithms. It consumes lower power, has better performance and can be implemented in small areas that can be achieved by ECC. In this paper, the Elliptic Curve properties are carefully analyzed.

In fact, multi-scalar multiplications are the most time consuming operations for elliptic curve cryptosystems where implementations are mainly on devices with constrained computational power and memory (i.e. smart card), therefore, efficient operations are essential. It is denoted by kP where P is a curve point and k a scalar one. Basic scalar multiplication algorithm scans each bit of k and performs some curve-level operations based on the bit value. Scalar representation significantly impacts the number of point operations to be executed and overall computation time representation of elliptic curve elements.

Multi-scalar multiplication is required in many elliptic curve cryptosystems (ECC)such as the verification process of ElGamal digital signature, verification process of ECDSA, provable-secure digital signatures [3,4], multi-party protocols [1] and protocols of Brands [2].

Hence, efficiency of multi-scalar multiplications is essential in elliptic curve cryptosystems. The major building block of most elliptic curve cryptosystems is the computation of multi-scalar multiplication.

Scalar multiplication corresponds to group element exponentiation in a multiplicative group, i.e. x^k, for some x in the multiplicative group. Therefore, one can easily adapt classical exponentiation methods to scalar multiplication, replacing multiplication by addition as well as squaring by doubling. Mathematicians have dealt

with exponentiation methods for more than two thousand years and efficient methods have since developed.

In this paper, we will present numerous scalar multiplication methods starting from generic ones up to curve specific ones. There are two main conditions that occur in practice: where P is a prime number and where it is not there is also a sub case in which the scalar is used several times. Additionally, different coding methods of k result in different scalar multiplication methods. The expected running time and the worst case running time are important to analyze a scalar multiplication method. Thus, one of them is always given in details when algorithm is coded. Algorithms are constructed for the P points in E(Fq) for some $q = p^m$ where p is prime and m is in Z^+, since cryptosystems are designed on E(Fq).

The following sections of this paper are organized as follows. Section 2 contains the introduction to a number of generic methods. Different methods based on the concept of fixed point would be explained in section 3. Section 4 studies scalar multiplication methods on some specific curves. The notion of endomorphism would be described in section 5. Section 6 is consisted of the conclusion and results.

2. Unknown Point

In this section, both k and P are unknown until the run-time, i.e. they are placed into the program at the run time. Since k and P may vary, methods given in this section may be realized as generic methods.

2.1. Binary Method

Binary method is the first known exponentiation method; therefore, it is a scalar multiplication method, too. Binary representation of the scalar multiplication enables us to interpret the multiplication as the cumulative addition of non-zero components. Namely, if k has the binary representation $(k_{l-1}, k_{l-2}, \ldots, k_0)_2$ where $k_i \epsilon \{0,1\}$, then $k = \sum_{i=0}^{l-1} k_i 2^i$.

$$kP = \sum_{i=0}^{l-1} k_i 2^i P \qquad (2.1)$$

$$= k_0P + k_1 2^1 P + \cdots + k_{l-1} 2^{l-1} \qquad (2.2)$$

$$= k_0P + 2(k_1P + 2(k_2P + \cdots + (2(k_{l-2}P + 2(k_{l-1}P)) \cdots) \qquad (2.3)$$

Equation 2.2 can be interpreted as starting from $k_0$and summing up the terms $k_i 2^i P$ cumulatively for each non-zero $k_i$up to $k_{l-1}$to end up with kP. $2^i P$can be calculated by $2 * 2^{i-1}P$ if $2^{i-1}P$ is known. Thus, in order to speed up, $2^i P$ must be calculated for each $i\epsilon \{0, \ldots, l-1\}$ by doubling the previous one, and adding it to the cumulative sum and if ki is 1. Because of which, this case of the method is known as double-and-add. Also, it is called right-to left binary method since it starts from k0 and ends with $kl-1$. The pseudo-code of right-to-left binary method is Algorithm 2.1.

Equation 2.3 enables us to interpret the multiplication as starting from $kl-1$down to k_0 and adding P if ki is non-zero and continuously doubling whatever ki is. Contrary to the previous case, it is not necessary to keep at hand a doubled version of P. In other words, memory is not allocated for a doubled version of P. Because of similar reasons, this case of the method is known as add-and-double or left-to-right binary method. Its pseudo-code is given in the Algorithm 2.2.

Algorithm 2.1.[6]

Right-to-left binary method for scalar multiplication

Input: $k = (kl-1, kl-2, \ldots, k0)2, P\epsilon E(Fq)$.

Output: kP

1. $Q = \infty$.

2. for i from 0 to l-1 do

2.1. if ki= 1 then Q = Q + P.

2.2. P = 2P.

3. output(Q)

Algorithm 2.2.[7]

Left-to-right binary method for scalar multiplication

Input: $k = (kl-1, kl-2, \ldots, k0)2, P\epsilon E(Fq)$.

Output: k_P

1. $Q = \infty$.

2. for i from l-1 down to 0 do

2.1 Q = 2Q

2.1 if ki= 1 then Q = Q + P.

3. Output (Q)

The running time of an algorithm is determined by the number of operations being performed throughout its execution. In order to do that, it is essential to analyze each line of the algorithm in detail. After the analysis, the expected running time of Algorithm 2.1 is l/2Additions + l Doubling denoted as

$$\frac{l}{2A} + lD \qquad (2.4)$$

Algorithm 2.2 has the same operations as Algorithm 2.1 with the reverse order, so they have the same running time.

2.2. Non-Adjacent Form (NAF)

Using the information from previous section, we know that the inverse of $P = (x, y) \in E(Fq)$ is −P = (x, x + y) in binary fields and −P = (x,−y) in the fields of characteristic $\gg 3$. Thus, taking the inverse of an element on elliptic curve is very fast in terms of computational time. This brings up the question of representing k in the form

$$k = \sum_{i=0}^{l-1} k_i 2^i \; where k_i \in \{-1, 0, 1\} \qquad (2.5)$$

Algorithm 2.3 to get fast computation for kP. When minus one comes across, subtraction of P is performed in addition to binary method during the scalar multiplication kP. A representation whose set also consists of negative values is called Signed Digit Representation (SDR). If the representation set is {-1, 0, 1}, then it is the most trivial type of signed digit representation known as signed binary representation. In the binary method, we have noticed that the running time of algorithm increases proportional to the number of 1s in its representation. Hence, the aim is to form a representation of an integer k whose weight (number of nonzero elements) and length is as small as possible. There is a representation which satisfies this aim:

Definition 3.4.A. the non-adjacent form (NAF) of a positive integer k is an expression $k = \sum_{i=0}^{l-1} k_i 2^i$ where $ki \in \{-1, 0, 1\}$, $kl - 1 \neq 0$ and no two consecutive digits are nonzero. According to this definition, the length of NAF

is l. For any $k \geq 1$, NAF exists and has the following properties:

Theorem 3.5.Let k be a positive integer, then

1. K has a unique NAF denoted as NAF(k) [8].

2. NAF(k) has the fewest nonzero digits among the signed binary representations of k [8].

3. The expected value of the 1s of NAF(k) over the length of NAF(k) is 1/3 [9].

iv. The length of NAF(k) is at most "one" more than the length of the binary representation of k [9].

Algorithm 3.6 gives the recoding1 of a given integer k into NAF(k). Then, Algorithm 2.5, which takes NAF(k), computes the scalar multiplication kP using NAF representation of k and is called NAF method. We will, first, give the algorithms and then explain the steps involved.

Algorithm 2.3.[8]

Right-to-left NAF recoding

Input: k = (kl−1, kl−2, . . . , k0)2

Output: NAF(k)

1. C_0= 0

2. For i = 0 to l do

2.1. $ci + 1 = \lfloor (ci + ki + ki + 1)/2 \rfloor$

2.2. $k_i^{'} = c_i + k_i - 2c_{i+1}$

3. $output\,(k_l^{'}, \ldots, k_0^{'})$

However, as we have seen in binary method, left-to-right scalar multiplication is more efficient than right to left one in terms of memory consumption.

Algorithm 2.4.[10]

Left-to-Right NAF recoding

Input: (kl−1, . . . , k0)2

Output: NAF(k)

1. j= m, b = 0, kl = 0

2. For i from l-1 down to 0 do

2.1. If (ki+1 = ki) then

2.1.1.$k_j^{'} = k_i - b$

2.1.2. while (j >i+ 1) do

2.1.2.1.

$$j = j - 1, k_j^{'} = 1 - ki - b, b = 1 - b$$

2.1.3. b= ki, j = j − 1

3. $k_j^{'} = -b$

4. while (j >0) do

4.1. $j = j - 1, k_j^{'} = 1 - b, b = 1 - b$

5. $output\ (k_l^{'}, \ldots, k_0^{'})$

Algorithm 2.5. Left-to-right NAF multiplication

$input: NAF(k) = (kl - 1, kl - 2, \ldots, k0)\ and\ P \in E(Fq)$

$output: kP \in E(Fq)$

1. $Q = \infty$

2. *For i from l − 1 down to 0 do*

2.1. *Q = 2Q*

2.2. *If ki= 1 then Q = Q + P*

2.3. *If ki= −1 then Q = Q − P*

3. *Output (Q)*

Note that line 2.1 of Algorithm 2.5 performs exactly l time and by Theorem 3.5 (iii) and (iv). It is expected that lines 2.2 and 2.3 together perform approximately l/3 times. Therefore, expected running time of Algorithm 2.5 is:

$l/3A + lD$ (2.6)

2.3. Window Method

If the digits of k representation are allowed to be the elements of a larger set instead of only {−1, 0, 1}, then, the running time of the above algorithms decrease. In this case, not only P is added or subtracted, but also any small scalar multiple of P is added or subtracted. So, those values have to be calculated at the beginning of the scalar multiplication algorithm and saved to the memory. The window method may be interpreted as processing some consecutive digits of the scalar at a time. There are unsigned and signed versions of the window method. Unsigned width-w window representation of a positive integer k is $k = \sum_{i=0}^{l-1} k_i 2^i$ where ki is either zero or an odd integer smaller than 2^w and $kl - 1 \neq 0$. Similarly, signed width-w window representation of a positive integer k is $k = \sum_{i=0}^{l-1} k_i 2^i$ where |ki| is either zero or an odd integer smaller than $2^{w-1}, k_{l-1} > 0$and particularly width-w NAF representation of a positive integer k is $k = \sum_{i=0}^{l-1} k_i 2^i$ where |ki| is either zero or an odd integer smaller than $2^{w-1}, k_{l-1} > 0$. Since width-w NAF reduces nonzero terms fairly, we will only deal with properties and running time

of it. In order to ensure that w consecutive digits contain at most one nonzero digit, reduction module 2^w has to be conducted by choosing the least residue as represented in line 2.2 of Algorithm 2.6.

Algorithm 2.6.[11]

Computing the NAF_w of a positive integer

Input: window width w, positive integer k

Output: $NAF_w(k)$

1. *i= 0*

2. *While k >0 do*

2.1. *If k is even ki= 0*

2.2. *Else k_i= k mod s 2w, k = k − k_i*

2.3. *K= k/2, i= i+ 1*

3. *Output (kl−1, . . . , k1, k0)*

As in the binary method and NAF method, if one can do the recoding of $NAF_w(k)$ left-to-right, scalar multiplication operation using $NAF_w(k)$ can be performed on-the-fly. That is, scalar multiplication and recoding operations can be performed simultaneously.

Avanzi[24], Muir et al.[12] and Okeya et al.[13] independently obtained similar results of left-to-right recoding of an integer k having the least Hamming weight. [12]'s left-to-right algorithm is optimal, and different from Avanzi's and it can output up to two different recordings of the same integer, one of which is equal to that of Avanzi's algorithm, whereas the other one differs in some of the less significant digits. Okeya et al. [14] also have a left-to-right algorithm, which is not equivalent to the w-NAF, but only gives asymptotic density estimates using Markov chains.

Algorithm 2.7.[11]

Left-to-right $NAF_w(k)$ multiplication

Input: $NAF_w(k)$, P ЄE(Fq)

Output: kP

0. Compute 2P

1. For i=3 up to 2^{w-1}– 1 do

1.1 if i is odd then compute Pi = Pi−1 + 2P

2. $Q = \infty$

3. for i = l-1 down to 0 do

3.1. Q = 2Q

3.2. *if $k_i \neq 0$ then*

3.2.1. if ki>0 then Q = Q + P_{ki}

3.2.2. else Q = Q − P_{-ki}

4. Output (Q)

Running time of line 0 and line 1 of Algorithm 2.7 is 1D and (2^{w-2} − 1) A respectively; pre-computation cost, therefore, is 1D+ (2^{w-2} −1) A. Next, the running time of line 3.1 is lD and the expected total running time of line 3.2.1 and line 3.2.2 is (l/(w + 1))A. To sum up, the expected running time of Algorithm 2.7 is

$$(l+1)D + (2^{w-2} + \frac{l}{w+1} - 1)A. \qquad (2.7)$$

2.4. Sliding Window Method

This method operates left-to-right over the digits of k with a maximum window width of w, at which the value in the window is odd. In contrast to window method, it has no exact window width; but similar to window method, it ignores zero digits. This method can be applied to binary or NAF representation of k. It may be applied to $NAF_w(k)$ with the maximum window width of w', but, the same algorithm of window NAF is obtained unless $w' > w$.

Algorithm of sliding window method applied to NAF_2 is given in Algorithm 2.8. So, it has to be computed first, and then placed into the algorithm. The first stage is the pre-computation of Pi for some i. Observe that a block of digits in an arbitrary window may have a maximum value of either 101010 . . . 10101(w-digits) or 101010 . . . 101001(w-digits) if w is odd or even respectively. Therefore, the value of the highest value block is either (2^{w+1} − 1)/3 or (2^{w+1} − 5)/3 which implies that the upper bound for the pre computation stage is :

$$2\,\frac{2^w-(-1)^w}{3} - 1 \qquad (2.7.1)$$

Algorithm 2.8.[15]

NAF Sliding window method for scalar multiplication

Input: window width w, NAF(k), P ЄE(Fq)

Output: kP

0. P1 = P

1. compute 2P

2. For i from 3 to $2(2^w-(-1)^w)/3-1$

2.1. $Pi = Pi-2 + 2P$

3. $Q = \infty, i = length(NAF(k)) - 1$

4. *while* $i \geq 0$ *do*

4.1. *if ki= 0 then t = 1, u = 0*

4.2. *else*

4.2.1. *t= 1; j = w*

4.2.2. *while (t = 1 and j >1) do*

4.2.2.1. *if (ki, . . . , ki−j+1) is odd t = j;*

u = (ki, . . . , ki−j+1)

4.3. $Q = 2^tQ$

4.4. *if u >0 then Q = Q + Pu; else if u <0 then*

Q = Q − P−u

4.5. i= i− t

5. Output (Q)

The average length of a run of zeros between windows in the NAF sliding window method is stated in [9] as

$$v(w) = \frac{4}{3} - \frac{(-1)^w}{3.2^{w-2}} \quad (2.8)$$

Pre-computation stage consists of lines 0, 1 and 2. Line 1 costs 1D and line 2costs $((2^w-(-1)^w)/3 - 1)A$. Thus, totally

$$\left(1D + \frac{2^w-(-1)^w}{3} - 1\right) \quad (2.8.1)$$

is done at the pre-computation stage. Next, lines 4.1, 4.2 are worthless. Line 4.3 is executed once, hence costs lD. In order to find the running time of line 4.4, we need to find the average number of nonzero terms and the number of zeros between the windows. The average length of a run of zeroes between windows in the NAF sliding window method is

$$v(w) = \frac{4}{3} - \frac{(-1)^w}{3.2^{w-2}} \quad (2.8.2)$$

The expected cost of line 4.4 is

$$\frac{l}{w+v(m)} \quad (2.8.3)$$

Therefore, the expected running time of Algorithm 2.8 is:

$$(1+l)D + \left(\frac{2^w-(-1)^w}{3} + \frac{l}{w+v(m)} - 1\right)A \quad (2.8.4)$$

2.5. Montgomery's Ladder

Montgomery [17] presented a ladder method to perform fast exponentiation (scalar multiplication). After that, [16] presented the elliptic curve version of Montgomery's ladder. However, their ideas were applicable only for

non-super singular curves over binary fields. [14] Extended Montgomery's ladder method for elliptic curves over non-binary fields. Hence, it became a generic method for computing scalar multiplication on elliptic curves.

The general idea of this method is to start from the left-most bit of the scalar and a pair (P, 2P) corresponding to the left-most bit. Then, iterate to the next left-most bit with a pair (2P, P + 2P) (i.e. doubling first component of the previous pair and adding the first and second component of previous pair) or (P +2P, 2(2P)) (i.e. addition of first and second component of previous pair and doubling second component of previous pair) if the next left-most bit of the scalar is 0 or 1, respectively. Continue this procedure until reaching the last bit and naturally the pair (kP, (k + 1)P). This idea is called scalar multiplication equivalence of exponentiation. In the above iteration, it is enough to compute x-coordinate of both components for each pair. Each iteration requires only an addition and a doubling. Also, there is a shortcut for the addition operation for elliptic curves over binary fields: For any given Qi = (xi, yi) i= 1, 2, 3, 4 satisfying Q3 = Q1 + Q2 and Q4 = Q1 − Q2. Then,

$$x_3 = x_4 + \frac{x_2}{x_1 + x_2} + \left(\frac{x_2}{x_1 + x_2}\right)^2$$

In our case, Q1 = (l + 1)P, Q2 = lP and Q4 = P are given and we are asked to compute (2l+1)P = Q3. Therefore, this shortcut could be applied at each iteration stage. Moreover, the y-coordinate of kP can be recovered, if needed, as: $y_1 = x^{-1}(x1 + x)[(x_1 + x)(x_2 + x) + x^2 + y] + y$, where $kP = (x_1, y_1)$, $(k + 1)P = (x_2, y_2)$ and P = (x, y).

Algorithm 2.9.[16]

Montgomery scalar multiplication

Input: k = (kl−1, . . . , k1, k0), P ЄE(Fq)

Output: kP

1. *X = P and Y = 2P*

2. *for i from l − 2 down to 0 do*

   2.1. *if ki= 0 then X = 2X and Y = X + Y*

   2.2. *else X = X + Y and Y = 2Y*

3. *Output(X)*

Algorithm 2.9 performs a doubling and an addition for each i whatever k_i is. Thus, the expected running-time of the algorithm is l(D + A). However, it is not necessary to compute the y-coordinate of corresponding points until the last iteration. Hence, it is not possible to compare the running-time of this algorithm with the running-time of the others.

2.6. Non-Adjacent Form (2-NAF) Method

In [26], NAF is a signed representation that uses elements from the set {-1, 0, 1}. It is a canonical representation with the smallest number of non-zero digits for any scalar *d*, denoted by NAF(*d*)and it contains at most *one* non-zero digit among any two successive digits. Moreover, the length of NAF(*d*) is at most *one* more bit than its binary representation.

To compute scalar multiplication of a scalar d, NAF performs two algorithms. The First algorithm converts the scalar to a signed representation while the second algorithm computes scalar multiplication.

The first algorithm (algorithm 2.10) computes the NAF of a scalar *d* if *w* is fixed to two and the second (algorithm 2.11) uses obtained NAF for scalar multiplication when $w = 2$.

Algorithm 2.10 Computing the *2*-NAF of a positive integer

Input: window *w* (*w* = 2), scalar *d*

Output: *NAFw(d)*

1. $i = 0$
2. *While* $d \geq 1$ *do*
   2.1. If *d* is odd, then *di = fct(d,2w), d = d - di*
   2.2. else di = 0
   2.3. $d = d/2, i = i+1$
3. *Return (di-1, …, d1,d0)NAFw*

Algorithm 2.11 *2*-NAF method for scalar multiplication

Input: Window *w* (*w* = 2), scalar d =(di-1, …, *d1,d0)*

PE (Fp)

Output: *dP*

1.Compute *Pi = iP for i{1, 3, 5, …, (2-1) }*

2. Q = O

3. For i = l – 1 down to 0 do

3.1. Q = 2Q

3.2. if di> 0 then *Q = Q + Pdi*

3.3.*Q = Q - Pdi*

4. Return (*Q*)

Function *fct(d,2^w)*

If d mod 2^W>2^W/2 then di = (d mod 2^w)-2^w

Else *di = d mod 2^w*

Let l be the bit-length of a scalar d, the expected number of doublings and additions using algorithm 2.2 is approximately (*l* – 1) and *l*/3, respectively. Thus the cost of the NAF method is:

*(l-1)*D + (l/3)A* (2.9)

3. Fixed Point

It was mentioned earlier in this paper that fixed point is the point for which a scalar multiplication is to be computed, and so the algorithm of scalar multiplication can be designed according to this privilege. For instance, if point P is fixed and some storage is available, then some of the multiples of P can be pre-computed and saved to memory, and then used during the computation of scalar multiplication directly by memory call.

3.1. Fixed-Base Windowing Method

If P is fixed, then the simplest idea that can be applied to the scalar multiplication is pre-calculation of all doublings of P up to 2^tP i.e. 2P, 4P, 8P, . . . ,$2^{t-1}P$ where t is equal to approximately extension degree m of our finite field. Then, for any given scalar k, one can compute kP by summing up only 2^iP for which k_i is nonzero. Hence, all doublings are removed, and expected running time of binary algorithms decreases to (l/2) A. A refinement to the above idea is first described in [18] and more refinement is given in section 4.6.3 of [6]. Finally, [19] proposed a patented version of previous ideas. The basic idea behind these refinements is the following equality:

$$KP = \sum_{i=0}^{l-1} k_i P = \sum_{j=1}^{L-1} \left(j \sum_{i:k_i=j} 2^{wi} P \right) where k = \sum_{i=0}^{d-1} K_i 2^{wi}$$

We now define the BGMW's algorithm. Let

$$k = (k_{l-1}, \ldots, k_1, k_0)_2 = (k_{d-1} \parallel \cdots \parallel k_0)$$

$$= (k_{d-1}, k_{d-2}, \ldots, k_0)_{2^w} \sum_{i=0}^{d-1} k_i \, 2^{wi} \qquad (3.2)$$

Then

$$KP = \sum_{i=0}^{d-1} k_i \, 2^{wi} P$$

$$= \sum_{i=0}^{d-1} k_i \, (2^{wi} P)$$

$$= \sum_{j=0}^{2^w-1} (j \sum_{i:k_i=j} 2^w P)$$

$$= \sum_{j=1}^{2^w-1} j \, Qj \qquad (3.3)$$

$$= Q2w-1 + (Q2w-1 + Q2w-2) + \ldots + (Q2w-1 + Q2w-2 + \ldots + Q1). \qquad (3.4)$$

Then, the corresponding algorithm is coded as, first, calculate Qj by cumulative addition of K_is and add Q_js together cumulatively to reach kP.

Algorithm 3.1.[19]

Binary BGMW's algorithm

$$input: w, d = \lceil l/w \rceil, k = (Kd-1, Kd-2, \ldots, K0)2^w, P \in E(Fq)$$

$$precomputed\ values: Pi = 2^{wi} P_i, 0 \leq i \leq d-1$$

Output: kP (3.1)

1. $A = \infty, B = \infty$
2. For j from $2^w - 1$ down to 1 do
   2.1. for i from 0 up to d − 1 do
      2.1.1. if j= Ki do B = B + Pi
   2.2. A = A + B
3. Output (A)

The pre-computation stage is not included in the running-time. The Expected running-time of line 2.2.1 of Algorithm 3.1 is $(d/(2^w - 1))(2^w - 1)$ since the equation j = Ki occurs expectedly $d/(2^w - 1)$ times and the outer loop performs this expectancy $2^w - 1$ times. Consequently, the addition at line 2.2.1 performs d times. In fact, it costs (d − 1)A if we discard the trivial first addition which is the identity plus a point. line 2.2 performs $2^w - 1$ times, but costs $(2^w - 2)$A by discarding the first trivial addition. Therefore, the expected running time of Algorithm 3.1 is $(2^w + d - 3)$A, where $d = \lceil l/w \rceil$

Algorithm 3.2 uses the same argument presented above with NAF representation of k instead of binary representation. NAF(k) = (Kd−1, . . .

,K1,K0) where each Ki is in non-adjacent form. Thus, Ki would be maximum if w is even or odd, respectively.

$$(1010 \dots 1010)_{NAF2} = \frac{2^{w+1}-2}{3} or (1010 \dots 101)_{NAF2} = \frac{2^{w+1}-1}{3}$$

(3.5)

Algorithm 3.2.[20]

NAF BGMW's algorithm

input: w,NAF(k), P 2 E(Fq)

$pre - computed\ values: Pi = 2(wi)P, 0 \leq i \leq \lceil (l+1)/w \rceil$

output: kP

1. $d = \lceil l/w \rceil$

2. if w is even S = $(2^{w+1} - 2)/3$ and else S = $(2^{w+1} - 1)/3$

3. $A = \infty, B = \infty$

4. for j from S to down to 1 do

    4.1. for i from 0 up to d-1 do

        4.1.1. if j= Ki do B = B + Pi

        4.1.2. else if −j = Ki do B = B − Pi

        4.1.3. A = A + B

5. output(A)

, the expected running time of Algorithm 3.2 is

$(2^{w+1}/3 + d - 2)A$ *where* $d = \lceil l/w \rceil$.

(3.6)

3.2. Fixed-Base Comb Method

This method is also known as Lim-Lee Method. Let d = $\lceil l/w \rceil$. If necessary, pad on the left side of the representation with zeros and then, split k into w concatenated parts: k = $K^{w-1}, \dots, K^0$.

$$KP = \sum_{i=0}^{i-1} k_i 2^i P$$

= k0P + k12P + k24P + ... + kt−12^{l-1}P

= k0P + ... + kd−12^{d-1}P + k(w−1)d$2^{(w-1)d}$P+ ... + kwd−12^{wd-1}P

= k0P + ... + (kd−1P)2^{d-1}+ k(w−1)d$2^{(w-1)d}$P+ ... + (kwd−$12^{(w-1)d}$P)2^{d-1}.

The general idea of the method is to operate on k column by column. First, kd−1P + . . . + kid2^{id}+ . . . + kwd−12^{wd-1} is calculated and doubled. Second, kd−2P + . . . + kid−12^{id-1}+ . . . + kwd−22^{wd-2} is calculated and added to the results of the first calculation and the final result is doubled. the procedure continues like this.

Finally, k0P + . . . + kid2^{i}+ . . . +k(w−1)d2$^{(w-1)d}$ is calculated and added to the previous sum. The obtained result represents kP. In order to accelerate the computation process, one can compute [aw−1, . . . , a2, a1, a0]P = aw−12$^{(w-1)d}$+. . .+a12^{d}P+a0P for all possible values of string (aw−1, . . . , a2, a1, a0). The exact algorithm is given below.

Algorithm 3.3.[21]

Fixed-base comb method point multiplication

$input: w, d = \lceil t/w \rceil, k = (k_{t-1}, \ldots, k_0)2 = K^{w-1} || \ldots || K^0 where K^i is d - bit integer and k_s^i is s^{th} bit of k^i, P \epsilon E(F_q)$

Pre computed values: [a$_{w-1}$, . . . , a$_2$, a$_1$, a$_0$] P for all possible strings (a$_{w-1}$, . . . , a0)

output: kP

1. $Q = \infty$

2. For i from d − 1 down to 0 do

2.1. Q = 2Q

2.2. $Q = Q + [K_i^{w-1}, K_i^{w-2}, \ldots, K_i^2, K_i^1]P$

3. Output (Q)

Probability of $[K_i^{w-1}, K_i^{w-2}, \ldots, K_i^2, K_i^1]$ is a zero. Hence, it is a non-zero array with a probability of 1 − 1/2^{w}. the non-infinite addition in line 2.2 occurs expectedly $(\frac{2^w-1}{2^w} d - 1)$ times (−1 comes from the first infinity addition).

Also, in line 2.1 a non-infinity doubling is executed (d − 1) times (−1 comes from the first infinity doubling). Therefore, the expected running time of Algorithm 3.3 is:

$$\left(\frac{2^w-1}{2^w} d - 1\right) A + (d-1) D \qquad (3.7)$$

In the case of additional memory being available for the algorithm, two columns can be executed simultaneously. The idea is to divide the columns of k into two parts. Apply the previous procedure simultaneously for both sides. This idea is illustrated in Algorithm 3.4

Algorithm 3.4.[21]

Fixed-base comb method point multiplication with two tables

$input: w, d = \lceil l/w \rceil, e = \lceil d/2 \rceil, k = (kl - 1, \ldots, k0)2 = K^{w-1} || \ldots || K^0 where K^i is$

$d - bit - integer and K_s^i is s^{th} bit of K^i, P \epsilon E(Fq)$

Pre-computed values: [aw−1, a2, a1, a0]P and 2^{e}[aw−1, . . . , a2, a1, a0]P for all

Possible string (aw−1. . . a2, a1, a0)

Output: kP

1. $Q = \infty$

2. for i from e-1 down to 0 do

2.1. Q = 2Q

2.2. $Q = Q + [K_i^{w-1}, K_i^{w-2}, \dots, k_i^2, k_i^1]P + 2^e[a_{w-1}, \dots, a_2, a_1, a_0]P$

3. Output (Q)

Similar to Algorithm 3.3, the expected time of Algorithm 3.4 can be calculated as $(\frac{2^w-1}{2^w}2e-1)A + (e-1)D$. On the other hand, Algorithm 3.4 requires twice as much storage for pre computation as Algorithm 3.3. If memory is limited, the two algorithms can be compared for a given fixed amount of pre computation.

4. Curve Specific Methods

In this section, we will study scalar multiplication methods on some specific curves. These methods developed from special properties of these curves. Specific scalar multiplication methods are included into many standards, e.g. NIST, IEEE, ISO, ANSI. The methods, studied in the previous two sections, are also applicable to the curves in this section. Moreover, generic methods are used with curve specific methods in order to decrease the computation time of the scalar multiplication.

4.1. Koblitz Curves

The non-supersingular curves defined over F2 are called Koblitz curves [22], also known as anomalous binary curves. Any non-supersingular curve over F2 is isomorphic to one of the following two curves:

E0 :$y^2 + xy = x^3 + 1$

E1 :$y^2 + xy = x^3 + x^2 + 1$.

Hence, there exist only two non-isomorphic Koblitz Curves: E0, E1. By simply counting, #E0(F2) = 4 and #E1(F2) = 2. Then, #E0($F2^m$) = 4n and $\#E1(F2^m) = 2n'$, for some n and $n' \epsilon Z$. In cryptography, n and n' are desired to be prime. N and n' can only be prime if m is prime; otherwise,

there exists a subgroup $Ea(F2^d)$ of Ea($F2^m$) for any d|m.

4.2. Z[τ] and τ – Representation

Let $a \epsilon \{0,1\}$, Then, the Frobenius map on Ea($F2^m$) is

$\tau : Ea(F2^m) \rightarrow Ea(F2^m)$

$\tau(\infty) = \infty$

$\tau(x,y) = (x^2, y^2)$.

Squaring in a binary field, when polynomial base is used, requires only insertion of zeros between the components and then reduction. This is very easy compared to other operations. Furthermore, when normal base is used instead of polynomial base, it is just a circular shift. Therefore, computing Frobenius map for any binary field element is very fast. From the previous chapter, it is known that Frobenius map is an endomorphism over $Ea(F2^m)$. Its characteristic polynomial is $x^2 - tx + 2$ where t is the trace of Frobenius and equals to $2+1-\#\ Ea(F2^m)$. Explicitly, the characteristic polynomial of Frobenius map over $E0(F2^m)$ is x^2+x+2, and similarly, over $E1(F2^m)$ is x^2-x+2. Let $\lambda = (-1)^{1-a}$,then, the characteristic polynomial of Frobenius over Ea is $x^2 - \lambda x + 2$. Hence,

$(\tau^2 - \lambda\tau + 2)P = 0 \ for\ all\ P \epsilon\ E_a(F_{2^m})$

It can be observed that $\tau = (\lambda + \sqrt{7})/2$ is one of the roots of the characteristic polynomial of Frobenius map over $E_a(F_{2^m})$. We can naturally lift the action of Frobenius map to the action of the commutative ring

$Z[x]/(x2 - \lambda x + 2) \cong Z[\tau]\ by\ identifying\ x\ with\ \tau : E_a(F_{2^m}) \rightarrow E_a(F_{2^m})$

This shows that the natural action of $Z[\tau]$ over Ea(F2m) induces the action of the ring $Z[\tau]$ on $E_a(F_{2^m})$:

$(s_l\tau^l + \cdots + s_1\tau + s_0)P = s_l\tau^l(P) + \cdots s_1\tau(P) + s_0P$

In general, for a scalar multiplication to be efficient, it is preferred to constitute the representation of the scalar as short as possible and small-sparse digits are surely desired. In the following sections, the length and digits of the representation of a scalar are investigated. Any element $\alpha \in Z[\tau]$ can be written uniquely in the

form $\alpha_0 + \alpha_1\tau$ for some integers α_0 and α_1 since $\tau^2 = \lambda\tau - 2$. In order to investigate the digits, we need to know the norm of an element in $Z[\tau]$.

Theorem and proof at [11] enables us to represent any positive integer k in terms of τ Similar to binary representation of $k = \sum_{i=0}^{l'-1} k_i 2$ where $k_i \epsilon \{0,1\}$, $\tau - adic$ representation of k can be obtained by repeatedly dividing k by τand the digits ui are remainders of the division steps. Since $N(\tau) = 2$, remainders are −1, 0 or 1.Then, any positive integer k can be represented in $\tau - adic$ representation uniquely as $k = \sum_{i=0}^{l-1} u_i\tau^i$ where each digit $u_i \epsilon 0, \pm 1$. Any generic method studied in the previous sections can be applied to $\tau - adic$ representation of k. in order to decrease the number of point additions, namely, to decrease the number of non-zero digits, NAF method can be applied to this representation. It becomes so called $\tau - adic$ NAF or TNAF $\tau - adic$ NAF is obtained in a similar way of 2-adic NAF.

Definition 3.27.TNAF of an element $k \in Z[\tau]$ *is* $k = \sum_{i=0}^{l-1} u_i\tau^i$ where each $u_i \in \{0, \pm 1\}, u_{l-1} \neq 0$ and no two consecutive digits are nonzero. The length of the TNAF is l. Computation of TNAF(τ) is similar to computation of NAF of an integer.

4.3. Scalar Multiplication on Koblitz Curves

Since a specific coding of k (TNAF(k)) is obtained , generic methods are applicable in this case. Algorithm 4.1 performs a scalar multiplication by using previously observed properties of the curve and TNAF..

Algorithm 4.1.[11]

TNAF scalar multiplication on Koblitz curves

Input: integer k Є[1, n − 1], P 2 E(F2^m) of order n

Output: kP

1. *compute* $\delta' = k\ partmod\ \rho$ *by Algorithm* 4.3

2. *compute* $TNAF(\delta') = \sum_{i=0}^{l-1} u_i\tau^i$ *by Algorithm* 4.2

3. $Q = \infty$

4. for i from l − 1 down to 0

4.1 $Q = \tau Q$

4.2 if ui= 1 Q = Q + P

4.3 if ui= −1 Q = Q − P

5. Output (Q)

Running-time of Algorithm 4.1 is l/3 A, since τQ is a very fast calculation and the weight of TNAF(k) is 1/3.

4.4. Window TNAF Method

Window method can be applied to TNAF(k) in order to increase the speed of the algorithm and to secure it against simple power attacks. Similar to width-w NAF method, width-w TNAF method processes w digits of. δ^' at a time. However, obtaining width-w TNAF is different. [10]. Algorithm 4.2.[11]

Window TNAF scalar multiplication method for Koblitz curves

input: window width w, integer k in [1, n − 1], P Є

Ea($F2^m$) of order n

Output: kP

1. use Algorithm 4.2 to compute δ'= k part mod ρ

2. Use Algorithm 4.3 to compute $TNAFw(\delta') = \sum_{i=0}^{l-1} u_i \tau^i$

3. Compute Pu= αuP, u Є{1, 3, 5, . . . , 2^{w-1}-1}

4. $Q = \infty$

5. For i= l − 1 down to 0

5.1 $Q = \tau Q$

5.2 $if\ ui \neq 0$

5.2.1 Let u be such that αu = uiorα−u = −ui

5.2.2 if u >0 then Q = Q + Pu

5.2.3 else Q = Q − P−u

6. Output (Q)

Pre-computation stage of Algorithm 3.38 costs (2^{w-2} − 1)A, while loop costs approximately $\frac{m}{w+1}A$. Thus, the expected running-time of Algorithm 3.4 is

$$(2^{w-2} - 1 + \frac{m}{w+1})A. \qquad (4.1)$$

5. Endomorphism

Endomorphism has a crucial role in the theory of elliptic curves. An endomorphism α of an elliptic curve E over a field K is defined as homomorphism on $E(\bar{K})$ given by rational functions. That is,

$\alpha: E(\bar{K}) \rightarrow E(\bar{K})$

$P \rightarrow (g_1(P), g_2(P))$

Where g1 and g2 are rational functions on E (quotient of polynomials) and

$\alpha(P1 + P2) = \alpha(P1) + \alpha(P2)$

for all P1 and $P2 \in E(\bar{K})$. The set of all endomorphism of E over K forms a ring under addition and composition [25], called the endomorphism ring of E over K. The characteristic polynomial of an endomorphism α is defined to be the least degree polynomial

$f(x) \in Z[x]$ *satisfying* $f(\alpha)(P) = \infty$ *for all* $P \in E$,

if it exists.

5.1. Endomorphism-Based Scalar Multiplication

Scalar multiplication can be performed faster by using a special endomorphism of the curves. In general, let $\emptyset$ be an endomorphism of an elliptic curve E(Fq) and let #E(Fq) be divisible by a prime r, but not by r^2. Then, by the properties of commutative groups, there exists only one subgroup of order r of E(Fq). Having prime order implies being a cyclic subgroup. Let it be generated by P, then, $\emptyset(P)$ has order r, since $\emptyset$ is an endomorphism. Hence, $\emptyset(P) \in < P >$, that is, $\emptyset(P) = \lambda P$ for some $\lambda \in [1, r-1]$. In fact, λ is a root modulo r for characteristic polynomial of $\emptyset$. We can apply above observations to scalar multiplication kP as follows: we find the λ expansion of k. However, since λ is too large, it is desirable to represent k as k = k1 + k2 λ(mod r) where k1 and k2 have approximately half the length of k. Then, kP= k1P + λ k2P = k1P + ϕ(k2P). Computing ϕ (k2P) is easy; it is just a one field multiplication. So, it is reduced to computing k1P and k2P. After finding k1 and k2, applying an interleaved right-to-left scalar multiplication method to k1P and k2P reduces the number of doubling operations approximately to half. if k1 and k2for a given k and λ (decomposition of k,)can be computed efficiently, The gain is considerable.

5.2. Decomposition of Scalar

It is necessary to find k1 and k2 satisfying k = f(k1, k2) = k1 + k2λ (mod r) and number of bits

of k1 and k2 are approximately half the number of bits of k, which means that k1 and k2 are small or $\sqrt{k_1^2 + k_2^2}$ is small. Thus, the aim is to find a short vector u such that f(u) = k. the Trivial solution is v = (k, 0), but this is not short. The approach is as following:

(1) find two vectors v1 = (a1, b1) and v2 = (a2, b2) in Z × Z satisfying

(i) v1 and v2 are linearly independent over R.

(ii) f(v1) = f(v2) = 0.

(iii) v1 and v2 are short themselves, that is, $\sqrt{a_i^2 + b_i^2}$ is small since ai and bi have half the bits of k which can be at least r, then it is approximately $\sqrt{r}$

(2) find a vector v in the integer lattice generated by v1 and v2 that is close to (k, 0). Then, u = (k, 0) − v is a short vector with

f(u) = f((k, 0)) − f(v) = k − 0 = k.

Note that sub-problems (1) and (2) may be solved using lattice basis reduction algorithms. However, [23]'s method is faster.

In Table 4.1, there is a comparison of scalar multiplications in terms of their number of additions, doublings and memory consumption.

5.3. Mutual Opposite Form (Mof)

The left-to-right recoding method eliminates the need for recoding and storing the multiplier in advance. Joye and Yen [27] first proposed the left-to-right recoding algorithm in the year 2000. In CRYPTO2004, Okeya [28] proposed a new efficient left-to-right recoding scheme called *mutual opposite form* (MOF).The unique property of the new singed binary representation is that Signs of adjacent non-zero bits (without considering 0 bits) are opposite. The maximum non-zero bit and the least non-zero bit

are 1 and -1, respectively. All the positive integers can be represented by a unique MOF.

5.4. Joint Sparse Form (Jsf)

Solinas presented a right-to-left method called Joint Sparse Form (JSF) for computing the signed binary representation of a pair of integers, which results in a minimal joint weight

compared to shamir.s method. The property of JSF is that the average joint weight among all JSF representations of two n-bit integers is n/2.Of any three consecutive positions, at least one is a double zero. Adjacent terms do not have opposite

- signs, that is Xj,Xj+1 # -1 and YjYj+1 # -1
- If X , Xj+1 # - 1, then Yj+1 = ±1 and Yj = 0
- If Yj, Yj+1 # - 1, then Xj+1 = ±1 and Xj = 0

The Algorithm is used for generating joint sparse form of a pair of integers called JSF of integers.

5.5. Direct Doubling

Among the various elliptic curve arithmetic operations, point doubling is quite costlier than point addition in the scalar multiplication over coordinate system. Sakai and Sakurai proposed a scalar multiplication algorithm using direct computation of several doublings (which computes 2kP directly from P) without computing the intermediate points 2P, 22P, 23P.2k-1P. The concept of direct computation of 2kP was first suggested by Guajardo and Paar. The new doubling formula was reconstructed as below.

A1=x1

B1=3x1^2 + a

C1= -y1

D1=12A1 C1^2- B1^2

x2= B1^2 . 8A1 C1^2 / (2C1)^2

y2 = 8C1^4 . B1 D1 / (2C1)^3

The computational complexity of this formula is (5S + 5M + I) and the existing method has the complexity of (6S + 4M + I).

Table 1 – Comparison of Different Algorithms

Algorithm	Addition	Doublings	Memory
3.2	$l/2$	l	-
2.5	$l/3$	l	-
2.7	$2^{w-2} + \frac{l}{w+1} - 1$	$l+1$	2^{w-2}

2.8	$\frac{2^w-(-1)^w}{3}+\frac{1}{w+v(w)}-1$	$l+1$	2^{w-2}
3.1	$(2^w+d-3), d=\left\lceil\frac{l}{w}\right\rceil$	-	$\left\lceil\frac{l}{w}\right\rceil$
3.2	$\left(\frac{2^{w+1}}{3}+d-2\right), d=\left\lceil\frac{l+1}{w}\right\rceil$	-	$\left\lceil\frac{l+1}{w}\right\rceil$
3.3	$\left(\frac{2^w-1}{2^w}d-1\right), d=\left\lceil\frac{l}{w}\right\rceil$	*(d-1)*	2^w
3.4	$\left(\frac{2^w-1}{2^w}2e-1\right), e=\left\lceil\frac{l}{2w}\right\rceil$	*(e-1)*	2^{w+1}
4.1	$l/3$	-	-

6. Conclusion

In this paper, we studied the scalar multiplication on elliptic curves. We made mathematical and computational analysis of many scalar multiplication methods.

In order to develop a scalar multiplication method or to improve an existing one, we first need to analyze the properties of elliptic curves carefully and then consider whether a recoding method of an integer corresponds to an efficient scalar multiplication method, and finally, use some algebraic methods on elliptic curves.

References

[1] E Karthikeyan, "Survey of Elliptic Curve Scalar Multiplication Algorithms Int. J. Advanced Networking and Applications Vol.4 Issue:2 Pp: 1581-1598

[2] A. C. Yao, "On the Evaluation of Powers," SIAM J. Comput. 5, pp. 100-103, 1976

[3] A. Enge, "Elliptic Curves and Their Applications to Cryptography: An Introduction," Kluwer Academic Publishers, 2009.

[4] C. H. Lim and P. J. Lee, "More flexible exponentiation with pre-computation," Advances in Cryptography, Crypto'94, LNCS 839, pp. 95-107, Springer-Verlag, 1994.

[5] D. Bressoud,"A Course in Computational Number Theory," (Key Curriculum Press), Springer-Verlag, 2008.

[6] D. M. Gordon, "A survey of fast exponentiation methods," J. Algorithms, 27(1):pp. 129-146, 1998.

[7] D.E. Knuth, "Semi numerical Algorithms," vol. 2 of The Art of Computer Programming, Addison Wesley, 2nd edition, 1997.

[8] E. F. Brickell, D. M. Gordon, K. S. McCurley, and D. B.Wilson, "Fast exponentiation with pre-computation," Advances in Cryptology Proceedings of Eurocrypt 92, LNCS 658, 200207. Springer-Verlag, 1992.

[9] F. Morain and J. Olivos, "Speeding up the computations on an elliptic curve using addition-subtraction chains," Inform. Theor. Appl., 24:531543, 1990.

[10] G. Reitwiesner, "Binary Arithmetic," Adv. Comput. 1:pp. 231-308, 1960.

[11] H. Silverman, "the Arithmetic of Elliptic Curves," Springer, 2009.

[12] J. Ho_stein, J. Pipher, and J. H. Silverman, "An introduction to mathematical cryptography," Springer-Verlag, New York, 2008.

[13] J. Lopez and R. Dahab, "Fast multiplication on elliptic curves over GF(2^m) without pre-computation," CHES 1999 LNCS 1717 316327. Springer-Verlag, 2000.

[14] J. Solinas, "Efficient arithmetic on Koblitz curves," Designs, Codes and Cryptography 19, 195249, 2000.

[15] J.A. Muir, and D. R. Stinson, "New Minimal Weight Representations for Left-to-Right Window Methods," Technical Report CACR 2004-03, Centre for Applied Cryptographic Research.

[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," CHES 2001, LNCS 2162 126141. Springer-Verlag,2001.

[17] K. Okeya, K. Schmidt-Samoa, C. Spahn, and T. Takagi, "Signed Binary Representations Revisited," Proceedings of Crypto 2004.

[18] K.Okeya,"Signed binary representation revisited," Proceedings of CRYPTO.04,

[19] M. Bellare, J.A. Garray, and T. Rabin, "Fast Batch verification for modular exponentiation and digital signatures," Advances in Cryptology-EUROCRYPTO'98, vol. 1403 of Lecture Notes in Computing Science, pp. 236-250. Springer-Verlag, 1998.

[20] M. Joye and S. M. Yen, "Optimal left-to-right binary signed-digit recoding," IEEE Transactions on Computers, vol.49, issue: 7):pp. 740-748, 2000.

[21] M.Joye and S.Yen, ""Optimal Left-to-Right binary signed digit recoding," IEEE Transactions on Computers, Vol. 49, pp. 740-748, 2000

[22] N. Koblitz, "CM-curves with good cryptographic properties," in: Advances in Cryptology, CRYPTO 91, LNCS 576, 279287, 1992.

[23] P. L. Montgomery, "Speeding the Pollard and Elliptic curve Methods of Factorization, "Mathematics of Computation, vol.48, issue: 170,:243264, 1987.

[24] R. Gallant, R. Lambert and S. Vanstone, "Faster Point Multiplication on Elliptic Curves with Efficient Endomorphisms," LNCS 2139, CRYPTO 2001.

[25] R. M. Avanzi, "A Note on the Sliding Window Integer Recoding and its Left-To-Right Analogue, "Workshop on Selected Areas in Cryptography SAC2004, LNCS 3357, pp. 130-143, Springer-Verlag, 2005.

[26] M. Lehsaini, M. Feham, Ch. T. Hellel, "Improvement of Scalar Multiplication Time for Elliptic Curve Cryptosystems",978-1-4799-1153-0/13/©2013IEEE.

[27] S. Brands, "Rethinking Public Key Infrastructures and Digital Certificates- Building in Privacy," MIT Press, p.356, 2000.

[28] T. Okamoto, "Practical identification schemes as secure as the DL and RSA problems," http://grouper.ieee.org/groups/1363/addendum.html# Okamoto, March 1999.

[29] T. Okamoto, "Provably secure and practical identification schemes and corresponding signature schemes," Advances in Cryptology-CRYPTO'92, vol.740 of Lecture Notes in Computing Science, pp. 31-53, Springer-Verlag, 1993.

Authors Profile:

Saeed Rahimi is graduated in shahed university and Emam Hosein university and his main activity is in cryptography algorithms and e voting systems.

