Scalar Multiplication in Elliptic Curve Cryptosystems: Pipelining with Pre-computations

Pradeep Kumar Mishra
Cryptographic Research Group
Indian Statistical Institute
203 B T Road
Kolkata - 700108
(INDIA)
e-mail: pradeep_t@isical.ac.in

Abstract

The pipelining scheme proposed in [24] is an efficient and secure scheme for computing scalar multiplication in Elliptic Curve Cryptosystems (ECC). The scheme proposed in [24] does not assume any pre-computation. In this work we extend the scheme to the situation where the system allows some pre-computation and is capable of storing some precomputed values. Like the scheme proposed in [24] our scheme uses an extra multiplier. On the performance front, it outperforms the best known sequential and parallel schemes. On the security front, our algorithm uses two countermeasures against SPA and hence are doubly secured against SPA. Also, it is secure against DPA using the Joye-Tymen's curve randomization countermeasure.

Keywords Elliptic Curve Cryptosystems, Pipelining, Scalar Multiplication, Jacobian coordinates, Comb methods, Window methods.

1 Introduction

Elliptic Curve Cryptosystems (ECC) since their inception (independently by Koblitz [18] and Miller [23] in 1985) is constantly gaining popularity due to intractability of the elliptic curve discrete logarithm problem (ECDLP). For a carefully chosen curve over a suitable underlying field there is no subexponential time algorithm to solve ECDLP. This fact enables ECC to provide a high level of security with much smaller keys in comparison to other popular cryptosystems based on integer factorisation or finite field based discrete logarithms.

Computationally the most expensive operation in ECC is scalar multiplication, namely, given an integer m and an elliptic curve point P, the computation of mP. The point P is generally refered to as the base point. It is computed by a series of doubling (ECDBL) and addition (ECADD) operation of the point P, depending upon the bit sequence representing m. Several methods have been proposed to perform the scalar multiplication securely and efficiently. For an excellent review the reader can refer to [13]. The performance of all these methods is dependent on the efficiency of the elliptic curve group operations: ECDBL and ECADD, which in turn depend upon the choice of the underlying field and the point representation. In the current work we will refer to ECADD and ECDBL as EC-operations. In affine coordinates EC-operations involve field inversion, an expensive operation particularly over fields of characteristic > 3. In the current work we will concentrate on such fields and use Jacobian coordinates for point representation.

The methods for computation of scalar multiplication can be broadly divided into two types: the ones which are suitable if the base point is fixed and the others are for variable base point. If the based point is fixed, then the some precomputation can be carried out and stored before the scalar multiplication actually begins. So the methods prescribed for fixed base point situation are generally more efficient. However, the methods require more memory and are less suitable for small devices with restricted resources.

The scalar multiplication is computed by a series of EC-operations. The pipelining scheme described in [24] is based on a key observation that instead of computing these operations one after another, they can be cascaded. The ECADD and ECDBL algorithms have their own set of inputs. These algorithms can be divided into parts some of which can be executed with only a part of the input. So one part of the algorithm can begin execution as soon as the corresponding part of the inputs is available to it. Thus two or more EC-operations can be executed in a pipeline. We discuss more about the pipelining scheme in Section 2.3.

In [19], [20], Paul Kocher et al. proposed Side-channel attacks (SCA), which are considered to be the most dangerous threat against mobile devices and ECC. The scheme proposed in the current work is doubly secure against simple power attacks, because it uses two countermeasures. First, like the pipelining scheme in [24] it uses side-channel atomic blocks to prevent SPA. Besides, the algorithms used for computing the scalar multiplication use a fixed pattern of EC-operations to compute the scalar multiplication. This makes the side-channel information uniform and does not leak out any data on the secret multiplier.

The comb method is an efficient method for computing the scalar multiplication. It uses a precomputed table and hence is considered suitable for fixed base point scenario of scalar multiplication. In the current work, we apply pipelining technique to comb method. The computation is naturally very efficient for fixed base point scenario. Also we provide an efficient method for computing the precomputated table online. When the precomputation is done online by our precomputation algorithm, the total complexity of precomputation and the comb method becomes quite affordable. Thus the proposed scheme is efficent for variable base point scenario as well. On the performance front our method compares favourably against all existing methods using similar amount of resources.

This has been established in Section 4.

The rest of the paper is organised as follows. In Section 2 we briefly deal with basics of elliptic cure cryptography, mainly highlighting the concepts we need in this work. In Section 3, we describe the proposed method. In Section 4 we examine the performance of the scheme and compare it with other known methods. Section 5 concludes the paper.

2 Elliptic Curve Preliminaries

In the current work we will concentrate on curves over large prime fields only. Over a finite field $F_q,\ q = p^r$ of odd characteristic $p > 3$, an elliptic curve has an equation of the form $y^2 = x^3 + ax + b$ where $a, b \in F_q$ and $4a^3 + 27b^2 \neq 0$. The set of points on an elliptic curve forms over a finite field forms a group under a specific operation written additively. A large number of elliptic curves over suitable underlying fields exist over which the discrete logarithm problem is believed to be hard. Elliptic curve cryptosystems are ElGamal type cryptosystems built over such groups. The most dominant operation in ECC is scalar multiplication. A lot of research has been carried out by research community for computing it securely and efficiently. Several algorithms have been proposed.

Scalar Multiplication is carried out by a series of EC-operations. In affine coordinates EC-operations require one field inversion each. Over prime fields the field inversion is a very expensive operation. To avoid these field inversions several point representations for the elliptic curve points have been proposed in literature. We will use Jacobian coordinates in the current work. Mixed additions are quite cheaper than general adition [7], so we will use mixed addition (Jacobian + affine = Jacobian) for addition steps in our scalar multiplication algorithm. Hence unless otherwise stated by ECADD we will generally mean this mixed addition in this paper.

2.1 Comb Methods

Among the scalar multiplication methods with precomputation two efficeint ones are the window method [25, 13] and the comb method [13]. The basic window method does not reduce the number of doublings. The efficiency is gained by reducing the number of additions. If the scalar is n bits in length, the Window method requires $(n-1)$ doublings. The binary method requires $n/2$ additions on average, which is lesser in window method. Applying pipelining technique to window method will surely yield a more efficient method. But comb methods are even more attractive as they require quite fewer doublings also. We devote this section to a brief description of the basic comb method.

The comb methods are very efficient in the fixed base point scenario of scalar multiplication.

Let m be the scalar multiplier. Let the binary representation of m be of n bits in length. Let w be a small integer and let $t = \lceil n/w \rceil$. We append

$tw-n$ bits in the left of the binary representation of m and we divide it into w bit strings of length t each. Let these bit strings be $K^{w-1}, K^{w-2}, \cdots, K^1, K^0$. Then the bit strings K^j can be written as the rows of an exponent array

$$\begin{bmatrix} K^0 \\ K^1 \\ \vdots \\ K^i \\ \vdots \\ K^{w-1} \end{bmatrix} = \begin{bmatrix} K^0_{t-1} & \cdots & K^0_0 \\ \vdots & & \vdots \\ K^i_{t-1} & \cdots & K^i_0 \\ \vdots & & \vdots \\ K^{w-1}_{t-1} & \cdots & K^{w-1}_0 \end{bmatrix} = \begin{bmatrix} k_{t-1} & \cdots & k_0 \\ \vdots & & \vdots \\ k_{t(i+1)-1} & \cdots & k_{ti} \\ \vdots & & \vdots \\ k_{wt-1} & \cdots & k_{(w-1)t} \end{bmatrix}$$

which is then processed columnwise, one after another. The computation is accelerated by precomputing the points

$$[a_{w-1}, \cdots, a_1, a_0]P = a_{w-1}2^{w-1}P + \cdots + a_1 2P + a_0 P$$

for all vectors $(a_{w-1}, \cdots, a_1, a_0) \in \{0,1\}^w$. That is, precompute the points $0P, 1P, \cdots, (2^w-1)P$ and store in a table $T[\,]$ such that $T[i] = iP$ for all $i \in \{0, 1, \cdots, 2^w-1\}$. The scalar multiplication is then computed by Algorithm 1.

Algorithm 1 (The Main Algorithm for Comb Method)

Input: *The point P and the integer m in the form specified and the table $T[]$.*
Output: mP.
1. Let $j = K^{w-1}_{t-1}2^{w-1} + \cdots + K^1_{t-1}2 + K^0_{t-1}$
2. Let $Q = T[j]$
3. For $i = t-2$ *down to 0*
4. $\quad j = K^{w-1}_i 2^{w-1} + \cdots + K^1_i 2 + K^0_i$
5. $\quad Q \leftarrow ECDBL(Q)$
6. $\quad Q = ECADD(Q, T[j])$
7. return Q

Proposition 1 *Algorithm 1 needs $t-1$ invocation of ECDBL and on average $\left(\frac{2^w-1}{2^w}t - 1\right)$ invocations of ECADD to compute the scalar multiplication.*

Proof : As ECDBL is invoked in the i-loop at Step 3 of the algorithm, the number of invocation of ECDBL is $(t-1)$. ECADD is also invoked same number of times. But actually an addition is not carried out if $j = 0$. Assuming all possibilities to be equally likely, the expected number of additions carried out at Step 6 is $\left(\frac{2^w-1}{2^w}t - 1\right)$. ■

2.2 Side-channel Attacks

Side-channel attacks (SCA), proposed by Paul Kocher et al. [19], [20] are very serious threat to ECC. SCA reveals the secret information by sampling and analyzing the side-channel information like timing, power consumption and EM radiation traces of an electronic computation. ECC is a very cryptosystem suitable for mobile and hand held devices, which are used in hostile outdoor environments. Hence an implementation must be side-channel resistant. Power attacks subsumes timing attacks [14] and can be divided into simple power attacks (SPA) and differential power attacks (DPA). Simple power attacks use information from one observation to break the secret. Differential power attacks use data from several observations and reveal the secret information by statistically analyzing them. Power analysis is the most serious of all side-channel attacks as these can be launched with very simple and easily available hardwares.

Several countermeasures have been proposed in literature to guard ECC against SPA and DPA (see [3], [8], [14], [17], [5] for example). The pipelining scheme proposed in [24] uses the side-channel atomicity to immunize against SPA. In the current work we use the same countermeasure. Also to facilitate pipelining, we use the same division of the EC-operations into atomic blocks as proposed in [24]. We provide the tables depicting the EC-operations as sequences of atomic blocks in the Appendix.

2.2.1 Curve Randomisation Countermeasure Against DPA

To immunize ECC from DPA, many countermeasures have been proposed. Most of them involve randomization of the processed data, such as the representation of the point or of the curve or of the scalar. For a details discussion on the topic the reader can refer to [5]. In the proposed scheme, we use Joye-Tymen's countermeasure to resist DPA [17]. The countermeasure is based on the following principle. Let C be the elliptic curve and let C' be a randomly chosen curve isomporphic to C. Let P' be the point on C' corresponding to the base point P. The countemeasure shifts the computation of scalar multiplication to the curve C' and computes mP'. After the computation the result is transformed back to the original curve. Put more succinctly,

Let z be a random nonzero field element. The steps are as follows.

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

The additional cost of obtaining DPA resistance is $4[m]$ for Step 1; $2[m]$ for Step 2; $2[m]$ for Step 3 and finally $1[i] + 2[m]$ for Step 5. We resort to this countermeasure as the base point can be kept in affine coordinates and efficiency can be gained from mixed additions. Any other DPA countermeasure which allows mixed additions can easily be incorporated into our scheme. Also, we use a variation of this scheme as our method uses a precomputed table (see Section 3.3).

2.3 The Pipelining Scheme

The scalar multiplication is generally computed by a sequence of ECDBL and ECADD, computed one after another in a sequential execution. The pipelining scheme proposed in [24] is based on a simple observation. Both of these operations are a sequence of field operations. ECDBL takes as input a point P of the elliptic curve and computes it double, $2P$. ECADD takes as input 2 points P_1, P_2 and returns their sum $P_1 + P_2$. In the scalar multiplication algorithm whenever ECADD is invoked one of the addend is fixed, the base point. Hence in the context of scalar multiplication, ECADD also has only one input point. Thus, input to both the EC-operations are three field elements, the x, y and z-coordinates of the input point. As mentioned earlier, we use the mixed addition for the addition operations in the scalar multiplication algorithm.

In both the EC-operations there are some operations which can be computed with the z-coordinate only, some can be computed with only the x and z coordinates. Also, the computation of the output z-coordinate needs minimum amount of computation and hence it can be computed first. Next, the output of x-coordinate needs lesser amount of computation than y coordinates and hence can be computed next. The computation of y-coordinates can be done at the last. This observation leads to a very efficient scalar multiplication method. In the left-to-right scalar multiplication algorithm, the first EC-operation is always an ECDBL. It is followed by an ECADD or an ECDBL according as the corresponding bit is 1 or 0. In a sequential execution, the subsequent EC-operation is invoked when the current one exits. In the pipelining scheme, the current operation outputs its first output as soon as it is computed and the next EC-operation begins in parallel. Both the operation continue execution, the earlier one producing its outputs and the subsequent operation using it as input. As the first process exits, the second process reaches the stage of producing its outputs and a third process starts execution in parallel with the second using the outputs of the second as its inputs.

In [24], the author has shown that the cascading of the EC-operations is nearly perfect in the sense that once an EC-operation starts executing it seldom waits for an input. In the refered work the author has used the side-channel atomic blocks countermeasure against SPA. In the Appendix we have produced the division of atomic blocks into atomic blocks conducive to the pipelining scheme. Also, we have presented the table describing how the EC-operations take part in the pipelining scheme in different scenarios of the scalar multiplication (e.g. when an ECDBL is followed by an ECADD or ECDBL; or when an ECDBL follows an ECADD).

In [24], the author has shown how the pipelining scheme can be implemented with just one extra multiplier and some more memory. The author has used the computation time of one atomic block as one unit of computation time. In fact, computation time of an atomic block is roughly same as that of a multiplication. The author has shown that except for the first EC-operation, which is always an ECDBL, each subsequent EC-operation can be computed in 6 units of time in the pipelining scheme. The first ECDBL takes 7 or 10 units of time according as the base point is in affine or Jacobian coordinates. Following [24], we assume that the cost of a squaring to be the same as that of a multiplication. This is however, not true in general.

3 Pipelined Comb Method

In this section we will apply the pipelining technique of [24] to comb method of computing the scalar multiplication described in Section 2.1.

3.1 Precomputations

We need to precompute the table $T[\,]$ described in Section 2.1. That is we have to precompute

$$[a_{w-1}, \cdots, a_1, a_0]P = a_{w-1}2^{w-1}P + \cdots + a_1 2P + a_0 P$$

for all vectors $(a_{w-1}, \cdots, a_1, a_0) \in \{0,1\}^w$. As the vector $(a_{w-1}, \cdots, a_1, a_0)$ ranges over all vectors in $\{0,1\}^w$, the point $[a_{w-1}, \cdots, a_1, a_0]P$ ranges over all points in $\{0.P, 1P, \cdots, (2^w - 1)P\}$. Hence we precompute iP for all $i \in \{0, 1, \cdots, 2^w - 1\}$ and store the precomputed values in a table $T[\,]$ such that $T[i] = iP$.

Let us denote $T[i]$ by T^i. As each T^i is a point on the elliptic curve, it has three coordinates, say, T^i_x, T^i_y and T^i_z, each of which is a field element. We have $T^0 = 0$ and $T^1 = P$. As we keep the base point in affine coordinates, we have $T^1_x = x$, $T^1_y = y$ and $T^1_z = 1$, where (x, y) are the coordinates of the base point P. We can use the Algorithm 2 for precomputation.

Algorithm 2 (Precomputation for Comb Method)

Input: *The point P and the integer w.*
Output: *The table $T[\,]$ such that $T[j] = jP \forall j \in \{0, 1, \cdots, 2^w - 1\}$.*
1. Let $T[0] = 0$
2. Let $T[1] = P$
3. For $i = 1$ to $2^{w-1} - 1$
4. $\quad T[2i] = ECDBL(T[i])$
5. $\quad T[2i+1] = ECADD(T[2i] + T[1])$

Observe that the ECADD abd ECDBL in Agorithm 2 can be computed in a pipeline. As the ECDBL of one iteration exits, the ECDBL of the next operation can enter the pipeline. In the pipeline scheme of [24], the EC-operations get their input and write back their outputs to the same locations, namely, T_6, T_7 and T_8. In this precomputation scheme a slight change to that has to be made. The ECDBL gets its input from T^i_x, T^i_y and T^i_z and writes back its output to T^{2i}_x, T^{2i}_y and T^{2i}_z. The ECADD gets its input from T^{2i}_x, T^{2i}_y and T^{2i}_z and writes back its output to T^{2i+1}_x, T^{2i+1}_y and T^{2i+1}_z.

The precomputation algorithm invokes ECADD and ECDBL $2^{w-1}-1$ times each. Hence it computes $(2^w - 2)$ EC-operations. The first doubling takes 7 units of time and the subsequent ones take 6 units of time each. Hence we have;

Proposition 2 *Algorithm 2 takes* $7 + 6 \times (2^w - 3) = 6 \times 2^w - 11$ *units of time to compute the table* $T[\,]$.

■

All entries in the table $T[\,]$, except $T[0]$ and $T[1]$ computed by Algorithm 2 are in Jacobian coordinates. We wish to store them in affine coordinates as we wish to take advantage of mixed addition in the scalar multiplication algorithm. To convert the entries into affine we need to compute inverse of z-coordinate of each of $T[i], 2 \le i \le 2^w - 1$. By Montgomery's trick it will take 1 inversion and $3 \times 2^w - 3$ multiplications. After computing the inverses, computing their squares and cubes and multiplying them with the corresponding x and y coordinates will take 4 multiplications per point of the table $T[\,]$ (recall that we are assuming the costs of multiplication and squaring to be the same). Hence, the conversion to affine stage of the precomputation will take $1[i] + (7 \times 2^w - 17)[m]$ computation, where $[i]$ and $[m]$ denote the computation time of an inversion and a multiplication respectively.

3.2 The Main Algorithm

The main algorithm for comb method has been presented in Section 2.1. Observe that the algorithm can again be processed in a pipelined manner. The ECDBL at Step 5 and ECADD at step 6 can be cascaded. When ECDBL exits, the ECADD can be cascaded with the ECDBL of next iteration. As the points stored in Table $T[\,]$ are in affine coordinates, one can take advantage of mixed additions.

3.3 Security Against SCA

The pipelined scheme we use is the same as the one proposed in [24]. In the scheme the author has used side-channel atomic blocks for security against SPA. So our scheme is also secure against SPA. At Step 6 of Algorithm 1, if a whole column of the exponent array is zero (i.e. $j = 0$) then no addition takes place. If this can be detected by the adversary from the side channel information, then he/she can obtain a partial information about the secret key. Although this is least likely, we can add further security measure by computing a dummy addition there. Thus the computation pattern becomes more uniform and the algorithm is now doubly secure against SPA. Proposition 1 states that the Algorithm 1 needs $\left(\frac{2^w-1}{2^w}t - 1\right)$ invocations of ECADD on average to compute the scalar multiplication. If dummy additions are carried out the computational requirement will be $t-1$ ECDBL and $t-1$ ECADD. Thus, the computational overhead for this extra security is only $t/2^w$ additions on average.

To prevent DPA, we can use Joye-Tymen curve randomisation method. We shift the whole computation to a random isomorphic curve and pull back the result to the original curve after the computation is over. We have to transform all points in the table $T[\,]$ to the corresponding points on the random curve.

3.4 Resistance Against DPA

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

1. Choose a random nonzero field element z.
2. Compute the relevant powers of z. (see Subsection 2.2.1)
3. Transform the curve parameters.
4. Transform each of the $2^w - 1$ nonzero points of $T[\,]$.
5. Perform scalar multiplication using Algorithm 1.
6. Transform the result back to the original curve.

Step 2 needs $3[m]$ (z^6 is not required), Step 3 requires $1[m]$ (transformation of b is not required), Step 4 requires $2\times(2^w-1)[m]$ and Step 6 requires $1[i]+5[m]$. Hence total cost is $1[i] + (2^{w+1} + 7)[m]$ taking $[m] = [s]$.

3.5 Cost of the Main Algorithm

The main algorithm invokes ECADD and ECDBL $t - 1$ times each. So it computes $2t - 2$ EC-operations. The first doubling takes 7 units of tie as the input point is in affine coordinates. Then each subsequent operation takes 6 units of time. Hence the whole computation takes $6(2t - 3) + 7 = 12t - 11$ units of time. The DPA countermeasure takes $1[i] + (2^{w+1} + 7)[m]$ amount of computation. Assuming $[i] = 30$ units of time, the DPA countermeasure takes $2^{w+1} + 37$ units of time. Hence computation time of the scalar multiplication is $12t + 2^{w+1} + 26$ time units. In Table 4 we present the computational cost for some typical values of w and t assuming $n = 160$.

4 Performance

In this section we will see the performance of the scheme for some representative values of n and w and compare performance of the scheme with other known methods requiring the same amount of resources. In Table 4 we have presented the cost of computing the scalar multiplication for $n = 160$ and different values of w. The values in Columns 2, 3 and 4 refer to the cost of precomputation, main and total amount of computation required for scalar multiplication in time units. Neglecting the field additions the unit of time is equal to the time of a field multiplication. The Storage-column in Table 4 refers to the number of points to be stored.

w	t	Precomputation	Main	Total	Storage
2	80	54	994	1048	3
3	54	106	660	766	7
4	40	210	538	758	15
5	32	418	474	892	31

If the base point is fixed, so that the precomputation can be done offline, then the scalar multiplication can be carried out with only 474 units of time with a storage of 31 points or 538 units of time with a storage of 15 points. The computational requirement can be further reduced if storage of more points is allowed. If the base point is not fixed and the precomputation is done offline, then the computation takes 758 units of time with a storage of 15 points and 766 units of time with a storage of 7 points only. The original pipeline scheme [24] with similar storage requirement requires 1152 units of time.

Our sheme requires one additional multiplier. Let us compare its performance vis-a-vis other known parallel methods.

Computation of scalar multiplication on ECC is not a new concept. Koyama and Tsuruoka [21] had proposed one such method as early as 1992. A special hardware was used to carry out the computation in their proposal. We compare our scheme with some of the recent proposals which are SCA resistant. The scheme proposed in [9], uses a parallelized encapsulated-add-and-double algorithm using Montgomery arithmetic. This algorithm uses two multipliers and takes $10[m]$ computations per bit of the scalar. So computation of the scalar multiplication for a 160 bit scalar will take 1600[m] or 1600 time units. In [11], the authors propose a parallel scheme, which computes the scalar multiplication with two multipliers in time equivalent to n doulings and $n/4$ additions in a sequential implementation. If we translate that into time units it is more than 2000[m]. In [16], the authors have proposed several schemes for parallel computation of scalar multiplication in ECC. Their best scheme requires 1592.4 units of time. Ofcourse their method does not require a precomputed table and hence requires less memory. In [1], the authors have proposed efficient algorithms for computing the scalar multiplication with SIMD (*Single Instruction Multiple data*). Similar and more efficient algorithms are also proposed in [15]. In [15] the authors have given two proposals. The first proposal, like our scheme, does not use precomputations and takes $1629[m]$ to compute the scalar multiplication. They have taken $[s] = 0.8[m]$. Their second proposal uses precomputed points, applies signed window expansions of the scalar and is quite efficient. Their best scheme requires storage of 16 points and computes the scalar multiplication in 942.4 units of time. Our proposal with similar memory requirement needs only 758 units of time.

5 Conclusion

In the current work we extend the simple pipelining scheme proposed in [24] for scalar multiplication in ECC to include precomputation. The method combines the pipelining technique with comb method of computing the scalar multiplication. The resultant method is secure against SCA and preforms better than many existing sequential and parallel schemes.

References

[1] K. Aoki, F. Hoshino, T. Kobayashi and H. Oguro. *Elliptic Curve Arithmetic Using SIMD*, In ISC, 2001, LNCS 2200, pp. 235-247, Springer-Verlag, 2001

[2] R. M. Avanzi. *On Multi-exponentiation in Cryptography*, To appear in *J. Cryptology*. Available at IACR eprint Archive, Technical Report No 2002/154, http://www.iacr.org.

[3] E. Briér and M. Joye. *Weierstrass Elliptic Curves and Side-Channel Attacks.* In PKC 2002, LNCS 2274, pages 335-345, Springer-Verlag,2002.

[4] B. Chevallier-Mames, M. Ciet and M. Joye. *Low-cost Solutions for Preventing Simple Side-Channel Analysis: Side-Channel Atomicity*, IEEE Trans. on Computers

[5] M. Ciet. *Aspects of Fast and Secure Arithmetics for Elliptic Curve Cryptography*, Ph. D. Thesis, Louvain-la-Neuve, Belgique.

[6] C. Clavier and M. Joye. *Universal Exponentiation Algorithm – A First Step Towards SPA Resistance*, In CHES, 2001, LNCS 2162, pp. 300-308, Springer-Verlag, 2001.

[7] H. Cohen, A. Miyaji, and T. Ono. *Efficient Elliptic Curve Exponentiation Using Mixed coordinates*, In ASIACRYPT'98, LNCS 1514, pp. 51-65, Springer-Verlag, 1998.

[8] J. -S. Coron. *Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems*, In *CHES 1999*, pages 292-302.

[9] W. Fischer, C. Giraud, E. W. Knudsen, J. -P. Seifert. *Parallel Scalar Multiplication on General Elliptic Curves over* $\mathbf{F}_p$ *hedged against Non-Differential Side-Channel Attacks*, Available at IACR eprint Archive, Technical Report No 2002/007, http://www.iacr.org.

[10] K. Fong and D. Hankerson and J. López and A. Menezes. *Field inversion and point halving revisited*, Technical Report, CORR 2003-18, Department of Combinatorics and Optimization, University of Waterloo, Canada, 2003.

[11] J. M. G. Garcia, R. M. Garcia. Parallel Algorithm for Multiplication on Elliptic Curves. Cryptology ePrint Archive, Report 2002/179, (2002), Available at `http://eprint.iacr.org`

[12] D. Gordon. A survey of fast exponentiation methods, *J. Algorithms*, 27(1):129–146, 1998.

[13] D. Hankerson, A. Menezes and S. Vanstone. *Guide to Elliptic Curve Cryptography*, Springer-Verlag, 2004.

[14] T. Izu, B. Möller and T. Takagi. Improved Elliptic Curve Multiplication Methods Resistant Against Side Channel Attacks, Proceedings of Indocrypt 2002, LNCS 2551, pp 296-313, Springer-Verlag.

[15] T. Izu and T. Takagi. Fast Elliptic Curve Multiplications with SIMD operation, ICICS 2002, LNCS, pp 217-230, Springer-Verlag.

[16] T. Izu and T. Takagi. A Fast Parallel Elliptic Curve Multiplication Resistant against Side Channel Attacks, ICICS 2002, LNCS, pp 217-230, Springer-Verlag.

[17] M. Joye and C. Tymen. *Protection against differential attacks for elliptic curve cryptography*, CHES 2001, LNCS 2162, pp 402-410, Springer-Verlag.

[18] N. Koblitz. *Elliptic Curve Cryptosystems*, Mathematics of Computations, 48:203-209, 1987.

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

[20] P. Kocher, J. Jaffe and B, Jun. *Differential Power Analysis,* CRYPTO'99, LNCS 1666, pp. 388-397, Springer-Verlag, 1999.

[21] K. Koyama, Y. Tsuruoka. *Speeding up elliptic Curve Cryptosystems Using a Signed Binary Windows Method*, In CRYPTO'92, LNCS 740, pp 345-357, Springer-Verlag, 1992.

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

[23] V. S. Miller. Use of Elliptic Curves in Cryptography. In *CRYPTO'85*, LNCS 218, pp. 417-426, Springer-Verlag, 1985.

[24] P. K. Mishra. Pipelined Computation of Scalar Multiplication in Elliptic Curve Cryptosystems. To appear in *CHES 2004*.

[25] B. Möller. Securing Elliptic Curve Point Multiplication against Side-Channel Attacks. In *Proc. of ISC 2001*, pages 324-334, 2001.

[26] J. Solinas. *Efficient arithmetic on Koblitz curves*, in Designs, Codes and Cryptography, 19:195-249, 2000.

A ECADD and ECDBL in Atomic Blocks

ECDBL Algorithm in Atomic Blocks

Input: $P_i(X_i, Y_i, Z_i)$

Input: $P_i = (X_i, Y_i, Z_i)$

Output: $2P_i = (X_{i+1}, Y_{i+1}, Z_{i+1})$

Δ_1	$R_1 = T_8 \times T_8\ (Z_i^2)$ * * * *	Δ_6	$R_4 = T_7 \times T_7\ (Y_i^2)$ $R_2 = R_4 + R_4\ (2Y_i^2)$ $R_2 = R_4 + R_4\ (2Y_i^2)$ * *
Δ_2	$R_1 = R_1 \times R_1\ (Z_i^4)$ * * *	Δ_7	$R_4 = T_6 \times R_2\ (2X_iY_i^2)$ $R_4 = R_4 + R_4\ (S)$ $R_4 = -R_4\ (-S)$ $R_5 = R_4 + R_4\ (-2S)$
Δ_3	$R_1 = a \times R_1\ (aZ_i^4)$ * * *	Δ_8	$R_3 = R_1 \times R_1\ (M^2)$ $T_6 = R_3 + R_5\ (\underline{X_{i+1}})$ * $R_4 = T_6 + R_4\ (X_{i+1} - S)$
Δ_4	$R_2 = T_6 \times T_6\ (X_i^2)$ $R_3 = R_2 + R_2 (2X_i^2)$ * $R_2 = R_3 + R_2\ (3X_i^2)$	Δ_9	$R_2 = R_2 \times R_2\ (4Y_i^4)$ $R_2 = R_2 + R_2 (8Y_i^4)$ * *
Δ_5	$T_8 = T_7 \times T_8\ (Y_iZ_i)$ $T_8 = T_8 + T_8\ (\underline{Z_{i+1}})$ * $R_1 = R_1 + R_2\ (M)$	Δ_{10}	$T_7 = R_1 \times R_4\ (M(X_{i+1} - S))$ $T_7 = T_7 + R_2\ (-Y_{i+1})$ $T_7 = -T_7\ (\underline{Y_{i+1}})$ *

ECADD Algorithm in Atomic Blocks
Input: $P = (T_x, T_y), P_i = (X_i, Y_i, Z_i)$
Output: $P + P_i = (X_{i+1}, Y_{i+1}, Z_{i+1})$.

Γ_1	$R_1 = T_8 \times T_8 \ (Z_i^2)$ $*$ $*$ $*$	Γ_7	$R_2 = R_2 \times R_4 \ (-U_1 W^2)$ $R_5 = R_2 + R_2 \ (-2U_1 W^2)$ $*$ $*$
Γ_2	$R_2 = T_x \times R_1 \ (U_1)$ $*$ $R_2 = -R_2 \ (-U_1)$ $*$	Γ_8	$R_1 = R_4 \times R_1 \ (W^3)$ $R_1 = R_1 + R_5 \ (W^3 - 2U_1 W^2)$ $R_3 = -R_3 \ (-S_1)$ $R_5 = R_3 + T_7 \ (S_2 - S_1 = -R)$
Γ_3	$R_3 = T_y \times T_8 \ (YZ_i)$ $*$ $*$ $*$	Γ_9	$T_6 = R_5 \times R_5 \ (R^2)$ $T_6 = T_6 + R_1 \ (\underline{X_{i+1}})$ $R_2 = T_6 + R_2 \ (X_{i+1} - U_1 W^2)$
Γ_4	$R_3 = R_3 \times R_1 \ (S_1)$ $R_1 = R_2 + T_6 \ (-W)$ $R_1 = -R_1 \ (W)$ $*$	Γ_{10}	$R_2 = R_5 \times R_2 \ (-R(X_{i+1} - U_1 W^2))$ $*$ $*$
Γ_5	$T_8 = R_1 \times T_8 \ (\underline{Z_{i+1}})$ $*$ $*$ $*$	Γ_{11}	$T_7 = R_3 \times R_4 \ (-S_1 W^2)$ $T_7 = T_7 + R_2 \ (\underline{Y_{i+1}})$ $*$ $*$
Γ_6	$R_4 = R_1 \times R_1 \ (W^2)$ $*$ $*$ $*$		

EC-operations in the Pipeline

	DBL-DBL		DBL-ADD		ADD-DBL	
Time	PS1	PS2	PS1	PS2	PS1	PS2
k	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$k+1$	$\Delta_1^{(i)}$	-	$\Delta_1^{(i)}$	-	$\Gamma_1^{(i)}$	-
$k+2$	$\Delta_2^{(i)}$	-	$\Delta_2^{(i)}$	-	$\Gamma_2^{(i)}$	-
$k+3$	$\Delta_3^{(i)}$	-	$\Delta_3^{(i)}$	-	$\Gamma_3^{(i)}$	-
$k+4$	$\Delta_4^{(i)}$	-	$\Delta_4^{(i)}$	-	$\Gamma_4^{(i)}$	-
$k+5$	$\Delta_5^{(i)}$	-	$\Delta_5^{(i)}$	-	$\Gamma_5^{(i)}$	-
$k+6$	$\Delta_1^{(i+1)}$	$\Delta_6^{(i)}$	$\Gamma_1^{(i+1)}$	$\Delta_6^{(i)}$	$\Delta_1^{(i+1)}$	$\Gamma_6^{(i)}$
$k+7$	$\Delta_2^{(i+1)}$	$\Delta_7^{(i)}$	$\Gamma_2^{(i+1)}$	$\Delta_7^{(i)}$	$\Delta_2^{(i+1)}$	$\Gamma_7^{(i)}$
$k+8$	$\Delta_3^{(i+1)}$	$\Delta_8^{(i)}$	$\Gamma_3^{(i+1)}$	$\Delta_8^{(i)}$	$\Delta_3^{(i+1)}$	$\Gamma_8^{(i)}$
$k+9$	$\Delta_4^{(i+1)}$	$\Delta_9^{(i)}$	$\Gamma_4^{(i+1)}$	$\Delta_9^{(i)}$	*	$\Gamma_9^{(i)}$
$k+10$	*	$\Delta_{10}^{(i)}$	$\Gamma_5^{(i+1)}$	$\Delta_{10}^{(i)}$	$\Delta_4^{(i+1)}$	$\Gamma_{10}^{(i)}$
$k+11$	$\Delta_5^{(i+1)}$	*	$\vdots$	$\Gamma_6^{(i+1)}$	*	$\Gamma_{11}^{(i)}$
$k+12$	$\vdots$	$\Delta_6^{(i+1)}$	$\vdots$	$\Gamma_7^{(i+1)}$	$\Delta_5^{(i+1)}$	*