SCIS 2002 The 2002 Symposium on Cryptography and Information Security
Shirahama,Japan, Jan.29-Feb.1,2002
The Institute of Electronics, Information and Communication Engineers

A Fast Parallel Elliptic Curve Multiplication Resistant against Side Channel Attacks

Tetsuya Izu * Tsuyoshi Takagi †

Abstract— This paper proposes a fast scalar multiplication algorithm, which improves both on an addition chain and an addition formula, based on [Mon87]. Our addition chain is applicable for for any types of elliptic curves over finite fields $\mathbb{F}_q$, requires no table look-up (or a few pre-computed points) and can be implemented in parallel. The computing time for n-bit scalar multiplication is one ECDBL + $(n-1)$ ECADDs in the parallel case and $(n-1)$ ECDBLs + $(n-1)$ ECADDs in the single case. We also propose faster addition formulas which only use the x-coordinates of the points. We also show criteria which make our algorithm resistant against the side channel attacks (SCA). We establish a faster scalar multiplication resistant against the SCA in both single and parallel cases. The improvement is about 37% for two processors and 5.7% for a single processor.

Keywords: elliptic curve cryptosystem, scalar multiplication, parallelization, side channel attack

1 Introduction

Let $E(\mathbb{F}_p)$ be an elliptic curve over a finite field $\mathbb{F}_p$ (p a prime). The dominant computation in the elliptic curve based cryptography (ECC) is the scalar multiplication $d * P$, where d is an integer and $P \in E(\mathbb{F}_p)$. It is usually computed by combining adding $P+Q$ (ECADD) and doubling $2*P$ (ECDBL), where $P, Q \in E(\mathbb{F}_p)$. Several algorithms have been proposed to enhance the running time of the scalar multiplication [CMO98]. The choice of the coordinate system and the addition chain is the most important factor. A standard way in [IEEE] is to use the Jacobian coordinate system and the addition-subtraction chain.

This paper improves both the addition chain and the addition formula. Our addition chain requires no table look-up (or a very small table) and it can be implemented in parallel for any types of elliptic curves over finite fields $\mathbb{F}_q$. Recently, Smart proposed a fast implementation over a SIMD type processor, which allows to compute several operations in the definition field in parallel [Sma01]. Our paper is motivated by his technique and the computation time for a scalar multiplication is one ECDBL + $(n-1)$ ECADDs. We also propose addition formulas which only use the x-coordinates of the points for Weierstrass form elliptic curve over $\mathbb{F}_p$, whichi is motivated by [Mon87]. The computations of the ECADD and the ECDBL require $9M+3S$ and $6M+3S$, where M, S are the times for a multiplication and a squaring in $\mathbb{F}_p$.

The key length of ECC is currently chosen smaller than those of the RSA and the ElGamal-type cryptosystems. The small key size of ECC is suitable for implementing on low-power mobile devices like smart cards. However, the side channel attacks (SCA) allow an adversary to reveal the secret key in the device by observing the side channel information such as the computing time and the power consumption [Koc96]. The simple power analysis (SPA) only uses a single observed information, while the differential power analysis (DPA) uses a lot of observed information together with statistic tools. To resist the SPA, one uses the indistinguishable addition and doubling in the scalar multiplication [CJ01]. In the case of prime fields, Hesse and Jacobi form elliptic curves achieve the indistinguishability by using the same formula for both an addition and a doubling [LS01, JQ01]. Because of the specialty of these curves, they are not compatible to the standardized curves in [IEEE, SEC]. The other uses the add-and-double-always method to mask the scalar dependency. The Coron's algorithm [Cor99] and the Montgomery form [OKS00] are in this category. To resist the DPA, some randomizations are needed [Cor99] and an SPA-resistant scheme can be converted to a DPA-resistant scheme [Cor99, JT01].

In this paper, we discuss a criteria, which makes our algorithms to be resistant against both the SPA and the DPA by comparing the Coron's algorithm. We establish a faster scalar multiplication resistant against the SCA in both single and parallel computations. The improvement of our scalar multiplication over the previously fastest method is about 37% for two processors and 5.7% for a single processor.

2 Elliptic Curve

In this paper we assume that $K = \mathbb{F}_p$ ($p > 3$) be a finite field with p elements. Elliptic curves over K can

* Secure Computing Lab. (L22), FUJITSU LABORATORIES Ltd., 4-1-1, Kamikodanaka, Nakahara-ku, Kawasaki, 211-8588, Japan, izu@flab.fujitsu.co.jp
† Technische Universität Darmstadt, Fachbereich Informatik, Alexanderstr.10, D-64283 Darmstadt, Germany ttakagi@cdc.informatik.tu-darmstadt.de

be represented by the equation

$$E(K) := \{(x, y) \in K \times K \mid y^2 = x^3 + ax + b\} \cup \mathcal{O}, \quad (1)$$

where $a, b \in K$, $4a^3 + 27b^2 \neq 0$ and $\mathcal{O}$ is the point of infinity. Every elliptic curve is isomorphic to a curve of this form, and we call it the Weierstrass form. An elliptic curve $E(K)$ has an additive group structure. Let $P_1 = (x_1, y_1)$, $P_2 = (x_2, y_2)$ be two elements of $E(K)$ that are different from $\mathcal{O}$ and satisfy $P_2 \neq \pm P_1$. Then the sum $P_1 + P_2 = (x_3, y_3)$ is defined as follows:

$$x_3 = \lambda^2 - x_1 - x_2, \quad y_3 = \lambda(x_1 - x_3) - y_1, \qquad (2)$$

where $\lambda = (y_2 - y_1)/(x_2 - x_1)$ for $P_1 \neq P_2$, and $\lambda = (3x_1^2 + a)/(2y_1)$ for $P_1 = P_2$. We call $P_1 + P_2 (P_1 \neq P_2)$ the elliptic curve addition (ECADD) and $P_1 + P_2 (P_1 = P_2)$, that is $2 * P_1$, the elliptic curve doubling (ECDBL). Let d be an integer and P be a point on the elliptic curve $E(K)$. The scalar multiplication is to compute the point $d * P$. There are three types of enhancements of the scalar multiplication. The first one is to represent the elliptic curve $E(K)$ with a different coordinate system, whose scalar multiplication is more efficient. For examples, a projective coordinate and a class of Jacobian coordinate has been studied [CMO98]. The second one is to use an efficient addition chain. The addition-subtraction chain is an example. The third one is to use a special type of curve such as the Montgomery form elliptic curve [OS00].

Coordinate System: A point on an elliptic curve can be represented in several ways. The costs of computing an ECADD and an ECDBL depend on the representation. The detailed description of the coordinate systems is given in [CMO98]. The major coordinate systems are as follows: the affine coordinate ($\mathcal{A}$), the projective coordinate ($\mathcal{P}$), the Jacobian coordinate ($\mathcal{J}$), the Chudonovsky coordinate ($\mathcal{J}^C$), and the modified Jacobian coordinate ($\mathcal{J}^m$). We summarize the costs in Table 1, where M, S, I denotes the computation time of a multiplication, a squaring, and an inversion in the definition field K, respectively. The speed of ECADD can be enhanced when the third coordinate is $Z = 1$.

	ECADD		ECDBL
	$Z \neq 1$	$Z = 1$	
$\mathcal{A}$	$2M + 1S + 1I$	—	$2M + 2S + 1I$
$\mathcal{P}$	$12M + 2S$	$9M + 2S$	$7M + 5S$
$\mathcal{J}$	$12M + 4S$	$8M + 3S$	$4M + 6S$
$\mathcal{J}^C$	$11M + 3S$	$8M + 3S$	$5M + 6S$
$\mathcal{J}^m$	$13M + 6S$	$9M + 5S$	$4M + 4S$

Table 1: Computing times of ECADD/ECDBL

Addition Chain: Let d be an n-bit integer and P be a point of the elliptic curve $E(K)$. A standard way for computing the scalar multiplication $d * P$ is to use the binary expression $d = d_{n-1}2^{n-1} + d_{n-2}2^{n-2} + \cdots + d_1 2 + d_0$, where $d_{n-1} = 1$ and $d_i = 0, 1$ $(n = 0, 1, ..., n-2)$. Then Algorithm 1 and Algorithm 2 compute $d * P$ efficiently. We call these methods the binary method. On average they require $(n-1)$ ECDBLs + $(n-1)/2$ ECADDs. Because computing the inversion $-P$ of P is essentially free, we can relax the binary coefficient to a signed binary $d_i = -1, 0, 1$ $(i = 0, 1, ..., n-1)$, which is called the addition-subtraction chain. The NAF offers a way to construct the addition-subtraction chain, which requires $(n-1)$ ECDBLs + $(n-1)/3$ ECADDs on average [IEEE].

```
INPUT d, P, (n)
OUTPUT d*P
  Q[0] = P
  for i=n-2 down to 0
    Q[0] = ECDBL(Q[0])
    if d[i]==1
      Q[0] = ECADD(Q[0],P)
  return Q[0]
```

Algorithm 1: Binary method from the most significant bit

```
INPUT d, P, (n)
OUTPUT d*P
  Q[0] = P, Q[1] = 0
  for i=0 to n-1
    if d[i]==1
      Q[1] = ECADD(Q[1],Q[0])
    Q[0] = ECDBL(Q[0])
  return Q[1]
```

Algorithm 2: Binary method from the least significant bit

The other enhancement technique is to utilize pre-computed tables. The Brickell's method and the sliding windows methods are two of the standard algorithms [BSS99]. These algorithms have been developed for the efficient modular multiplications over finite fields. In this paper we are interested in efficient algorithms without table look-up. Our goal is to propose an efficient algorithm that is suitable for smart cards, and the pre-computed table sometimes hinders to achieve the high efficiency because the memory spaces are expensive and an I/O interface to read the table is relatively slow.

Special Elliptic Curves: With a special class of elliptic curves, we can enhance the speed of a scalar multiplication. Okeya-Sakurai proposed to use the Montgomery form [OS01] because of the speed of its scalar multiplication. However, in all standards, the curves are defined by the Weierstrass form over $\mathbb{F}_p$ or $\mathbb{F}_{2^m}$. Every Montgomery form curve cannot be generally converted to the Weierstrass form, because the order of the Montgomery form curves is always divisible by 4.

3 Side Channel Attacks

The side channel attacks (SCA) are serious attacks against mobile devices like smart cards. An adversary can obtain a secret key from a cryptographic device without breaking its physical protection. We can achieve the attack by analyzing side channel information, i.e., computing time, or power consumption of the devices. The timing attack (TA) and the power analysis attack are examples of the SCA [Koc96]. The simple power analysis (SPA) only uses a single observed information, and the differential power analysis (DPA) uses a lot of observed information together with statistic tools. As the TA can be regarded as a class of the SPA, we are only concerned with the SPA and the DPA in this paper.

Countermeasures against SPA: The binary methods of Algorithm 1 and 2 compute ECADDs when the bit of the secret key d is 1. Therefore an attacker can easily detect the bit information of d by the SPA.

Coron proposed a simple countermeasure against the SPA by modifying the binary methods (Algorithm 1', 2') [Cor99]. These algorithms are referred as the add-and-double-always methods. In both algorithms, Step 3 and 4 compute both an ECDBL and an ECADD in every bits. Thus an attacker cannot guess the bit information of d by the SPA. A drawback of this method is their efficiency. Algorithm 1' requires $(n-1)$ ECADDs + $(n-1)$ ECDBLs and Algorithm 2' requires n ECADDs + n ECDBLs.

```
INPUT d, P, (n)
OUTPUT d*P
 1:  Q[0] = P
 2:  for i=n-2 down to 0
 3:     Q[0] = ECDBL(Q[0])
 4:     Q[1] = ECADD(Q[0],P)
 5:     Q[0] = Q[d[i]]
 6:  return Q[0]
```

Algorithm 1': Add-and-double-always method from the most significant bit (SPA-resistant)

```
INPUT d, P, (n)
OUTPUT d*P
 1:  Q[0] = P, Q[1] = 0
 2:  for i=0 to n-1
 3:     Q[2] = ECADD(Q[0],Q[1])
 4:     Q[0] = ECDBL(Q[0])
 5:     Q[1] = Q[1+d[i]]
 6:  return Q[1]
```

Algorithm 2': Add-and-double-always method from the least significant bit (SPA-resistant)

Möller proposed an SPA-resistant algorithm which is a combination of Algorithm 1' and the window method [Moe01]. However, his method requires extra table look-up (at least three points).

Another countermeasure is to establish the indistinguishability between an ECADD and an ECDBL. Joye-Quisquater and Liardet-Smart proposed to use the Jacobi and Hesse form elliptic curves, which use the same mathematical formulas for both an ECADD and an ECDBL [JQ01, LS01]. A drawback of this approach is that the Jacobi and Hesse form are special types of elliptic curves and they cannot be used for the standard Weierstrass form.

Converting SPA-resistance to DPA-resistance: Even if a scheme is SPA-resistant, it is not always DPA-resistant, because the DPA uses not only a simple power trace but also a statistic analysis, which has been captured by several executions of the SPA. Coron pointed out that some parameters of ECC must be randomized in order to be DPA-resistant [Cor99]. By the randomization we are able to enhance an SPA-resistant scheme to be DPA-resistant.

The key idea of Coron's 3rd countermeasure [1] is as follows. Let $P = (X : Y : Z)$ be a base point in the projective coordinate. Then P and $(rX : rY : rZ)$ $(r \in K)$ are same mathematically, but not in the computation. If we randomize a base point with r before starting the scalar multiplication, the side information for the scalar multiplication will be randomized. An extra cost for the countermeasure is small.

The other enhancement against the DPA was proposed by Joye-Tymen [JT01]. This countermeasure uses an isomorphism of an elliptic curve. The base point $P = (X : Y : Z)$ and the definition parameters a, b of an elliptic curve can be randomized in its isomorphic classes like $(r^2X : r^3Y : Z)$ and r^4a, r^6b, respectively. In this countermeasure, Z can be 1 during the scalar multiplication and it improves the efficiency of the scalar multiplication in some cases. An extra cost for the countermeasure is small, too.

4 Proposed Algorithm

We describe our proposed algorithm in the following. The algorithm improved on the addition chain and the addition formula. Both improvements are based on the scalar multiplication by Montgomery [Mon87]. However, we firstly point out that the addition chain is applicable for not only Montgomery form curves but any type of curves. We enhance it to be suitable for implementation and study the security against the SCA. We also establish the addition formulas, which only use the x-coordinate of the points, for the Weierstrass form curves.

4.1 Addition Chain

The improved addition chain is as follows:

```
INPUT d, P, (n)
OUTPUT d*P
 1: Q[0] = P, Q[1] = 2*P
 2: for i=n-2 down to 0
 3:    Q[2] = ECDBL(Q[d[i]])
 4:    Q[1] = ECADD(Q[0],Q[1])
 5:    Q[0] = Q[2-d[i]]
 6:    Q[1] = Q[1+d[i]]
 7: return Q[0]
```

Algorithm 3: Proposed addition chain (SPA resistant)

For each bit $d[i]$, we compute $Q[2] = \mathrm{ECDBL}\ (Q[d[i]])$ in Step 3 and $Q[1] = \mathrm{ECADD}\ (Q[0], Q[1])$ in Step 4. Then the values are assigned $Q[0] = Q[2], Q[1] = Q[1]$ if $d[i] = 0$ and $Q[0] = Q[1], Q[1] = Q[2]$ if $d[i] = 1$. The

[1] As Coron proposed three countermeasures, Okeya-Sakurai showed the bias in his 1st and 2nd countermeasures [OS00].

correctness of our algorithm is given in Theorem 4.1 [IT02].

Theorem 4.1. Algorithm 3, on input a point P and an integer $d > 2$, outputs the correct value of the scalar multiplication $d * P$.

Algorithm 3 requires one ECDBL in the initial Step 1, and $(n-1)$ ECDBLs and $(n-1)$ ECADDs in the loop. The computation time of the loop is same as that of Algorithm 1'.

Remark: Algorithm 3 does not depend on the representation of elliptic curves, and it is applicable to execute a modular exponentiation in any abelian group. Therefore the RSA cryptosystem, the DSA, the ElGamal cryptosystem can use our proposed algorithm.

Parallel Computation: We discuss the parallelizion of ECADD and ECDBL in the addition chains. Algorithm 1' cannot be parallelized, because ECADD requires the output of ECDBL. Algorithm 2' and Algorithm 3 can be parallelized in this sense.

In the right side of Figure 1 we show an architecture of the parallel computation of the loop of Algorithm 3. It has two registers: Register 1 and Register 2, which are initially assigned $Q[0] = P$ and $Q[1] = 2 * P$, respectively. In Step 3, we choose the value $Q[d[i]]$ based on the bit information $d[i]$, then compute ECDBL($Q[d[i]]$) from $Q[d[i]]$. In Step 4, we compute ECADD($Q[0], Q[1]$) from the value $Q[0]$ in Register 1 and $Q[1]$ in Register 2. In both Step 3 and Step 4, they do not need the output from Step 3 nor Step 4, and they are excused independently. After finishing to compute both ECDBL($Q[d[i]]$) and ECADD($Q[0], Q[1]$), we assign the values in Register 1 and Register 2 based on the bit $d[i]$. If $d[i] = 0$, we assign the ECDBL($Q[d[i]]$) in Register 1 and the ECADD($Q[0], Q[1]$) in Register 2. If $d[i] = 1$, we swap the two variables, then we assign the ECADD($Q[0], Q[1]$) in Register 1 and the ECDBL($Q[d[i]]$) in Register 1.

In general the computation of an ECADD is slower than that of an ECDBL, so that the latency of the loop in Algorithm 3 depends on the running time of ECADDs. Thus the total running time of Algorithm 3 is one ECDBL and $(n-1)$ ECADDs, where n is the bit-length of d.

Security Consideration: We discuss the security of Algorithm 3 against the SCA. Algorithm 1' is commonly believed secure against the SPA [OS00]. The relation between Algorithm 1' and Algorithm 3 is as follows [IT02].

Theorem 4.2. Algorithm 3 is as secure as Algorithm 1' against the SPA, if we use a computing architecture whose swapping power of two variables is negligible.

Corollary 4.3. Algorithm 3 with Coron's 3rd or Joye-Tymen's countermeasure is as secure as Algorithm 1' against the DPA.

It is possible to implement the swapping of two variables in hardware using a few logic gates. Its power is usually negligible. In software we can implement it just to swap two pointer assignments. The swapping of the pointer assignments in software can be executed in several clocks, whose time or power trace is negligible. Therefore, our proposed method is secure against the DPA in many computing environments.

Further Parallelization: If a table of pre-computed points is allowed to be used, we can construct a scalar multiplication, which can be computed in parallel with more than two processors. For example, the algorithm for 4 processors in [IT02] computes at most $(n/2+3)$ ECADDs. Moreover, the security against the SCA can be discussed in the same way.

4.2 Addition formula

Let E be an elliptic curve defined by the standard Weierstrass form (1) and $P_1 = (x_1, y_1)$, $P_2 = (x_2, y_2)$, $P_3 = P_1 + P_2 = (x_3, y_3)$ be points on $E(K)$. Moreover, let $P_3' = P_1 - P_2 = (x_3', y_3')$. Then we obtain the following relations:

$$x_3 \cdot x_3' = \frac{(x_1x_2 - a)^2 - 4b(x_1 + x_2)}{(x_1 - x_2)^2}, \quad (3)$$

$$x_3 + x_3' = \frac{2(x_1 + x_2)(x_1x_2 + a) + 4b}{(x_1 - x_2)^2}. \quad (4)$$

On the other hand, letting $P_4 = 2 * P_1 = (x_4, y_4)$ leads to the relation

$$x_4 = \frac{(x_1^2 - a)^2 - 8bx_1}{4(x_1^3 + ax_1 + b)}. \quad (5)$$

With these relations, the x-coordinates of P_3 and P_4 can be computed just form the x-coordinates of the points P_1, P_2, P_3'. The scalar multiplication can be computed with these relations, and we call the method combined with (3), (5) ((4),(5)) the multiplicative (additive) x-coordinate-only method. The x-coordinate-only methods were originally introduced by Montgomery [Mon87]. However, his main interest was to find a special form of elliptic curves on which the computing times are optimal. The additive method was not discussed in his paper.

When we use the x-coordinate-only methods, we need the difference $P_3' = P_1 - P_2$. This may be a problem in general, but not in Algorithm 3. In each loop of Algorithm 3, the two points $(Q[0], Q[1])$ are simultaneously computed and they satisfy the equation $Q[1] - Q[0] = P$, where P is a base point of the scalar multiplication. Therefore, the difference P_3' for ECADD(P_1, P_2) in Algorithm 3 are always known. On the contrary, in Algorithm 2', we need extra computation to know P_3'.

When we apply the x-coordinate-only methods to Algorithm 3, the output is only the x-coordinate of $d * P$. This is enough for some cryptographic applications, but other applications also require the y-coordinate of $d * P$ [SEC]. However, the y-coordinate of $d * P$ is easily obtained in the following way: The final values of $Q[0], Q[1]$ in Algorithm 3 are related by $Q[1] = Q[0] + P$. Let $P = (x_1, y_1), Q[0] = (x_2, y_2), Q[1] = (x_3, y_3)$. Here known values are x_1, y_1, x_2, x_3 and the target is y_2. Using a standard addition formula (2), we obtain the equation $y_2 = (2y_1)^{-1}(y_1^2 + x_2^3 + ax_2 + b - (x_1 - x_2)^2(x_1 + x_2 + x_3))$. This y-recovering technique was originally introduced by Agnew et al. for curves over $\mathbb{F}_{2^m}$ [AMV93]. The computing time for y-recovering is $16M + 4S + 1I$.

The projective coordinate system offers a faster computation for the x-coordinate-only methods. The computing times for (3),(4),(5) in the projective coordinate are $\mathtt{ECADD}_m^{(x)} = 9M + 2S$, $\mathtt{ECADD}_a^{(x)} = 10M + 2S$, $\mathtt{ECDBL}^{(x)} = 6M + 3S$. If $Z_3' = 1$, the computing times deduce to $\mathtt{ECADD}_{m(Z_3'=1)}^{(x)} = \mathtt{ECADD}_{a(Z_3'=1)}^{(x)} = 8M + 2S$. The concrete algorithms are in [IT02].

5 Comparison

In this section, we compare the computing times of a scalar multiplication resistant against the SCA. We show that our proposed algorithm establishes a faster scalar multiplication. The improvement of our method over the previously fastest method is about 37% for two processors and 5.7% for a single processor.

Estimation: We compare the computing times of a scalar multiplication with Algorithm 1', 2', and 3 using different coordinate systems. All algorithms are DPA-resistant using Coron's 3rd or Joye-Tymen's countermeasure. We estimate the total times to output a scalar multiplication $d * P = (x_d, y_d)$ on input d, $P = (x, y)$ and elliptic curve (a, b, p). The times are given in terms of the numbers of the arithmetic in the definition field, i.e., the multiplication M, the squaring S, and the inverse I. In the estimation, we include the times for randomization by Coron's 3rd or Joye-Tymen's countermeasure, and the times for recovering the y-coordinate in the x-coordinate-only method are also included. In the estimation, we also give the estimated running time for a 160-bit scalar. The last numbers in the brackets are the estimation for $1S = 0.8M, 1I = 30M$ [OS01].

Single Case: The estimated running timesa using a single processor are in Table 2. Algorithm 3/Joye-Tymen with the x-coordinate-only methods are the fastest (2929.0M). The previously fastest algorithm was Algorithm 1'/Joye-Tymen with the Jacobian coordinate $\mathcal{J}$ (3095.0M). The improvement is about 5.7%.

Parallel Case: The estimated running timesa using two parallel processors are in Table 3. Algorithm 1' cannot be computed in parallel and Algorithm 2' has no computational advantage to use the x-coordinate-only methods. Therefore, the previously fastest algorithm was Algorithm 2'/Coron's 3rd with the Chudonovsky coordinate $\mathcal{J}^C$ (2181.6M). Algorithm 3/Joye-Tymen with x-coordinate-only methods provides the fastest multiplication (1593.4M). The improvement is about 37%.

References

[AMV93] G.Agnew, R.Mullin and S.Vanstone, "An implementation of elliptic curve cryptosystems over $F_{2^{155}}$", *IEEE J. on Selected Areas in Communications*, vol.11, pp.804-813, 1993.

[BSS99] I.Blake, G.Seroussi and N.Smart, *Elliptic Curves in Cryptography*, Cambridge University Press, 1999.

[Cor99] J.Coron, "Resistance against differential power analysis for elliptic curve cryptosystems", *CHES'99*, LNCS 1717, pp.292-302, 1999.

[CMO98] H.Cohen, A.Miyaji and T.Ono, "Efficient elliptic curve exponentiation using mixed coordinates", *Asiacrypt'98*, LNCS 1514, pp.51-65, 1998.

[CJ01] C.Clavier and M.Joye, "Universal exponentiation algorithm – A first step towards provable SPA-resistance –", *CHES2001*, LNCS 2162, pp.300-308, 2001.

[IEEE] IEEE P1363, Standard Specifications for Public-Key Cryptography, 2000. Available from `http://groupe.ieee.org/groups/1363/`

[IT02] T.Izu and T.Takagi, "A fast parallel elliptic curve multiplication resistant against side channel attacks", *PKC2002*, to appear.

[JQ01] M. Joye and J. Quisquater, "Hessian elliptic curves and side-channel attacks", *CHES2001*, LNCS 2162, pp.402-410, 2001.

[JT01] M.Joye and C.Tymen, "Protections against differential analysis for elliptic curve cryptography", *CHES2001*, LNCS 2162, pp.377-390, 2001.

[Koc96] C.Kocher, "Timing attacks on Implementations of Diffie-Hellman, RSA, DSS, and other systems", *Crypto'96*, LNCS 1109, pp.104-113, 1996.

[LS01] P.Liardet and N.Smart, "Preventing SPA/DPA in ECC systems using the Jacobi form", *CHES2001*, LNCS 2162, pp.391-401, 2001.

[Moe01] B.Möller, "Securing elliptic curve point multiplication against side-channel attacks", *ISC 2001*, LNCS 2200. p.324-334, 2001.

[Mon87] P.Montgomery, "Speeding the Pollard and elliptic curve methods for factorizations", *Math. of Comp*, vol.48, pp.243-264, 1987.

[OKS00] K.Okeya, H.Kurumatani and K.Sakurai, "Elliptic curves with the Montgomery form and their cryptographic applications", *PKC2000*, LNCS 1751, pp.446-465, 2000.

[OS00] K.Okeya and K.Sakurai, "Power analysis breaks elliptic curve cryptosystems even secure against the timing attack", *Indocrypt 2000*, LNCS 1977, pp.178-190, 2000.

[OS01] 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", *CHES2001*, LNCS 2162, pp.126-141, 2001.

[Sma01] N.Smart, "The Hessian form of an elliptic curve", *CHES2001*, LNCS2162, pp.118-125, 2001.

[SEC] Standards for Efficient Cryptography Group (SECG), Specification of Standards for Efficient Cryptography. Available from `http://www.secg.org`

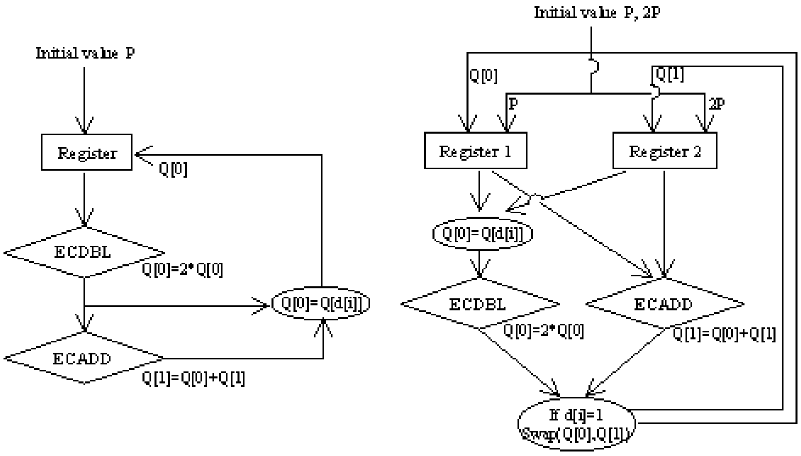

Figure 1: Algorithm 1' (left), the parallel implementation of Algorithm 3 (right)

	Addition formula	Computing Time	
		Total	$n = 160$
Algorithm 1' /Coron 3rd	$\mathcal{P}$	$(19n-15)M + (7n-7)S + 1I$	$3025M + 1113S + 1I$ $(3945.4M)$
	$\mathcal{J}$	$(16n-10)M + (10n-8)S + 1I$	$2550M + 1592S + 1I$ $(3853.6M)$
	$\mathcal{J}^{\mathcal{C}}$	$(16n-10)M + (9n-7)S + 1I$	$2550M + 1433S + 1I$ $(3726.4M)$
	$\mathcal{J}^m$	$(17n-10)M + (10n-7)S + 1I$	$2710M + 1593S + 1I$ $(4014.4M)$
Algorithm 1' /Joye-Tymen	$\mathcal{P}$	$(16n-7)M + (7n-4)S + 1I$	$2553M + 1116S + 1I$ $(3475.8M)$
	$\mathcal{J}$	$(12n-3)M + (9n-5)S + 1I$	$1917M + 1435S + 1I$ $(3095.0M)$
	$\mathcal{J}^{\mathcal{C}}$	$(13n-4)M + (9n-5)S + 1I$	$2076M + 1435S + 1I$ $(3254.0M)$
	$\mathcal{J}^m$	$(13n-4)M + (9n-5)S + 1I$	$2076M + 1435S + 1I$ $(3254.0M)$
Algorithm 2' /Coron 3rd	$\mathcal{P}$	$(19n+4)M + 7nS + 1I$	$3085M + 1120S + 1I$ $(4011.0M)$
	$\mathcal{J}$	$(16n+6)M + (10n+2)S + 1I$	$2566M + 1602s + 1I$ $(3877.6M)$
	$\mathcal{J}^{\mathcal{C}}$	$(16n+16)M + (9n+2)S + 1I$	$2566M + 1442S + 1I$ $(3749.6M)$
	$\mathcal{J}^m$	$(17n+7)M + (10n+3)S + 1I$	$2727M + 1603S + 1I$ $(4039.4M)$
Algorithm 2' /Joye-Tymen	$\mathcal{P}$	$(19n+9)M + (7n+3)S + 1I$	$3049M + 1123S + 1I$ $(3977.4M)$
	$\mathcal{J}$	$(16n+9)M + (10n+4)S + 1I$	$2569M + 1604S + 1I$ $(3882.2M)$
	$\mathcal{J}^{\mathcal{C}}$	$(16n+9)M + (9n+4)S + 1I$	$2569M + 1444S + 1I$ $(3754.2M)$
	$\mathcal{J}^m$	$(13n+9)M + (9n+4)S + 1I$	$2089M + 1444S + 1I$ $(3274.2M)$
Algorithm 3 /Coron 3rd	$\mathcal{P}$	$(19n-8)M + (7n-2)S + 1I$	$3032M + 1118S + 1I$ $(3956.4M)$
	$\mathcal{J}$	$(16n-6)M + (10n-2)S + 1I$	$2554M + 1598S + 1I$ $(3862.4M)$
	$\mathcal{J}^{\mathcal{C}}$	$(16n-5)M + (9n-1)S + 1I$	$2555M + 1439S + 1I$ $(3736.2M)$
	$\mathcal{J}^m$	$(17n-6)M + (10n-3)S + 1I$	$2714M + 1597S + 1I$ $(4021.6M)$
	x (mul)	**$(15n+8)M + (5n+2)S + 1I$**	**$2408M + 802S + 1I$ $(3079.6M)$**
	x (add)	**$(16n+7)M + (5n+2)S + 1I$**	**$2567M + 802S + 1I$ $(3238.6M)$**
Algorithm 3 /Joye-Tymen	$\mathcal{P}$	$(19n-4)M + 7nS + 1I$	$3036M + 1120S + 1I$ $(3962.0M)$
	$\mathcal{J}$	$(16n-4)M + (10n-1)S + 1I$	$2556M + 1599S + 1I$ $(3865.2M)$
	$\mathcal{J}^{\mathcal{C}}$	$(16n-3)M + 9nM + 1I$	$2557M + 1440S + 1I$ $(3739.0M)$
	$\mathcal{J}^m$	$(17n-5)M + (10n-3)S + 1I$	$2715M + 1597S + 1I$ $(4022.6M)$
	x (mul)	**$(14n+15)M + (5n+5)S + 1I$**	**$2255M + 805S + 1I$ $(2929.0M)$**
	x (add)	**$(14n+15)M + (5n+5)S + 1I$**	**$2255M + 805S + 1I$ $(2929.0M)$**

Table 2: Computing times of a scalar multiplication (a single processor)

	Addition formula	Computing Time	
		Total	$n = 160$
Algorithm 2' /Coron 3rd	$\mathcal{P}$	$(12n+4)M + 2nS + 1I$	$1924M + 320S + 1I$ $(2210.0M)$
	$\mathcal{J}$	$(12n+6)M + (4n+2)S + 1I$	$1926M + 642S + 1I$ $(2469.6M)$
	$\mathcal{J}^{\mathcal{C}}$	$(11n+6)M + (3n+2)S + 1I$	$1766M + 482S + 1I$ $(2181.6M)$
	$\mathcal{J}^m$	$(13n+7)M + (6n+3)S + 1I$	$2087M + 963S + 1I$ $(2887.4M)$
Algorithm 2' /Joye-Tymen	$\mathcal{P}$	$(12n+9)M + (2n+3)S + 1I$	$1929M + 323S + 1I$ $(2217.4M)$
	$\mathcal{J}$	$(12n+9)M + (4n+4)S + 1I$	$1929M + 644S + 1I$ $(2474.2M)$
	$\mathcal{J}^{\mathcal{C}}$	$(11n+9)M + (3n+4)S + 1I$	$1769M + 484S + 1I$ $(2186.2M)$
	$\mathcal{J}^m$	$(13n+9)M + (6n+4)S + 1I$	$2089M + 964S + 1I$ $(2890.2M)$
Algorithm 3 /Coron 3rd	$\mathcal{P}$	$(12n-1)M + (2n+3)S + 1I$	$1919M + 323S + 1I$ $(2207.4M)$
	$\mathcal{J}$	$(12n-2)M + (4n+4)S + 1I$	$1918M + 644S + 1I$ $(2463.2M)$
	$\mathcal{J}^{\mathcal{C}}$	$11nM + (3n+5)S + 1I$	$1760M + 485S + 1I$ $(2178.0M)$
	$\mathcal{J}^m$	$(13n-2)M + (6n+1)S + 1I$	$2078M + 961S + 1I$ $(2876.8M)$
	x (mul)	**$(9n+14)M + (2n+5)S + 1I$**	**$1454M + 325S + 1I$ $(1744.0M)$**
	x (add)	**$(10n+13)M + (2n+5)S + 1I$**	**$1613M + 325S + 1I$ $(1903.0M)$**
Algorithm 3 /Joye-Tymen	$\mathcal{P}$	$(12n+3)M + (2n+5)S + 1I$	$1923M + 325S + 1I$ $(2213.0M)$
	$\mathcal{J}$	$12nM + (4n+5)S + 1I$	$1920M + 645S + 1I$ $(2466.0M)$
	$\mathcal{J}^{\mathcal{C}}$	$(11n+2)M + (3n+6)S + 1I$	$1762M + 486S + 1I$ $(2180.8M)$
	$\mathcal{J}^m$	$(13n-1)M + (6n+1)S + 1I$	$2079M + 961S + 1I$ $(2877.8M)$
	x (mul)	**$(8n+21)M + (2n+8)S + 1I$**	**$1301M + 328S + 1I$ $(1593.4M)$**
	x (add)	**$(8n+21)M + (2n+8)S + 1I$**	**$1301M + 328S + 1I$ $(1593.4M)$**

Table 3: Computing times of a scalar multiplication (two parallel processors)