Technical Report, CACR, University of Waterloo
URL: http://www.math.uwaterloo.ca/

A Fast Parallel Elliptic Curve Multiplication Resistant against Side Channel Attacks *

Tetsuya Izu** † Tsuyoshi Takagi‡

December 17, 2001

** 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

Abstract

This paper proposes a fast elliptic curve multiplication algorithm applicable for any types of curves over finite fields $\mathbb{F}_p$ (p a prime), based on [Mon87], together with criteria which make our algorithm resistant against the side channel attacks (SCA). The algorithm improves both on an addition chain and an addition formula in the scalar multiplication. Our addition chain requires no table look-up (or a very small number of pre-computed points) and a prominent property is that it 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. By combination of our addition chain and addition formulas, we establish a faster scalar multiplication resistant against the SCA in both single and parallel computation. The improvement of our scalar multiplications over the previous method is about 37% for two processors and 17.1% for a single processor. Our scalar multiplication is suitable for the implementation on smart cards.

Keywords: elliptic curve cryptosystem, scalar multiplication, parallel computation, side channel attack

---

*This paper is revised from [IT02].

†This work was done while the first author was staying at the Centre for Appllied Cryptographic Research (CACR), University of Waterloo and part of this work was done while the first author was visiting Fachbereich Informatik, Technische Universität Darmstadt.

1 Introduction

In recent years, several elliptic-curve based cryptosystems (ECC) have been included in many standards [ANSI, IEEE, NIST, SEC, WAP]. 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, mobile phones and PDAs (Personal Digital Assistants, such as Palm and Pocket PC). Let $E(K)$ be an elliptic curve over a finite field $K = \mathbb{F}_p$ (p a prime). The dominant computation of the encryption/decryption and the signature generation/verification of ECC is the scalar multiplication $d * P$, where $P \in E(K)$ and d is an integer. It is usually computed by combining an adding $P + Q$ (ECADD) and a doubling $2 * P$ (ECDBL), where $P, Q \in E(K)$. Several algorithms have been proposed to enhance the running time of the scalar multiplication [Gor98, 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. Some efficient addition chains use a table look-up method. It is useful for software implementation but not for smart cards because the cost of the memory spaces is expensive and an I/O interface to read the table is relatively slow.

This paper proposes a fast multiplication which is applicable for any type of elliptic curves over finite fields $K = \mathbb{F}_p$ (p a prime). The algorithm improves both the addition chain and the addition formula in the scalar multiplication. Our addition chain requires no table look-up (or a very small table) and a prominent property of our addition chain is that it can be implemented in parallel.[1] The latency of the scalar multiplication is the computation time of one ECDBL + $(n-1)$ ECADDs. The improvement from the method in [IEEE] is the time of $(n-2)$ ECDBLs $-(2n+2)/3$ ECADDs. Moreover, our proposed addition chain computes the scalar multiplication after one ECDBL + $(n-1)$ ECADDs exactly, although the expected time of the binary method is only estimated on average.

The side channel attacks (SCA) allow an adversary to reveal the secret key in the cryptographic device by observing the side channel information such as the computing time and the power consumption [Koc96, KJJ99]. An adversary does not have to break the physical devise to obtain the secret key. It is a serious attack especially against mobile devices like smart cards. 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. There are two approaches to resist the SPA. The first 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 [ANSI, IEEE, SEC]. The second one 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 be a DPA-resistant scheme [Cor99, JT01]. The cost of the conversion is relatively cheap comparing with the scalar multiplication itself.

[1] Recently, Smart proposed a fast implementation over a SIMD type processor, which allows to compute several operations in the definition field in parallel [Sma01].

In this paper, we discuss a criteria, which makes our algorithms to be resistant against the SCA by comparing the Coron's algorithm. Moreover, we also propose addition formulas which only use the x-coordinates of the points. 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 the definition field $\mathbb{F}_p$. By combination of our addition chain and addition formulas, we establish a faster scalar multiplication algorithm 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 17.1% for a single processor.

2 Elliptic Curves and Scalar Multiplications

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

$$E(K) := \{(x, y) \in K \times K | y^2 = x^3 + ax + b \ (a, b \in K,\ 4a^3 + 27b^2 \neq 0)\} \cup \mathcal{O}, \qquad (1)$$

where $\mathcal{O}$ 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 [MO90]. We can also apply the addition chains developed for the ElGamal cryptosystem over finite fields [Gor98]. The third one is to use a special type of curve such as the Montgomery form elliptic curve [OS00], or the Hesse form [JQ01, Sma01].

Coordinate System: There are several ways to represent a point on an elliptic curve. The costs of computing an ECADD and an ECDBL depend on the representation of the coordinate system. The detailed description of the coordinate systems is given in [CMO98]. The major coordinate systems are as follows: the affine coordinate system ($\mathcal{A}$), the projective coordinate system ($\mathcal{P}$), the Jacobian coordinate system ($\mathcal{J}$), the Chudonovsky coordinate system ($\mathcal{J}^C$), and the modified Jacobian coordinate system ($\mathcal{J}^m$). We summarize the costs in Table 1, where M, S, I denotes the computation time of a multiplication, a squaring, and an inverse in the definition field K, respectively. The speed of ECADD or ECDBL can be enhanced when the third coordinate is $Z = 1$ or the coefficient of the definition equation is $a = -3$.

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

Table 1: Computing times of an addition (ECADD) and a doubling (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 of $d = d_{n-1}2^{n-1} + d_{n-2}2^{n-2} + \ldots + d_1 2 + d_0$, 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 methods (or the add-and-double methods). On average they require $(n-1)$ ECDBLs + $(n-1)/2$ ECADDs. Because computing the inverse $-P$ 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]
```

```
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 1 (leftside): Binary method from the most significant bit
Algorithm 2 (rightside): 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. We can refer to the nice survey paper [Gor98]. In this paper we are interested in efficient algorithms without table look-up (or with a very small pre-computed table). 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 and Sakurai proposed to use the Montgomery form [OS00]. The addition formula of the Montgomery form is much simpler than that of the Weierstrass form, and its scalar multiplication is also faster. However every Montgomery form cannot be generally converted to the Weierstrass form, because the order of the Montgomery form curves is always divisible by 4.

ECC has been standardized in several organizations like ANSI, IEEE, SEC, NIST, WAP. In all standards, the curves are defined by the Weierstrass form over $\mathbb{F}_p$ or $\mathbb{F}_{2^m}$, where p is a prime number or m is an integer. The example curves over $\mathbb{F}_p$ cannot be represented by the Montgomery form. Indeed, all curves in [NIST, ANSI] and all curves defined over a prime field with larger than 160-bit prime in [IEEE] are not compatible.

3 Side Channel Attacks to ECC

The side channel attacks (SCA) are serious attacks against mobile devices such as smart cards, mobile phones and PDAs. 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, KJJ99]. 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 we 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 adversary 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                     | 1:  Q[0] = P, Q[1] = 0
 2:  for i=n-2 down to 0          | 2:  for i=0 to n-1
 3:     Q[0] = ECDBL(Q[0])        | 3:     Q[2] = ECADD(Q[0],Q[1])
 4:     Q[1] = ECADD(Q[0],P)      | 4:     Q[0] = ECDBL(Q[0])
 5:     Q[0] = Q[d[i]]            | 5:     Q[1] = Q[1+d[i]]
 6:  return Q[0]                  | 6:  return Q[1]
```

Algorithm 1' (leftside): Add-and-double-always method from the most significant bit (SPA-resistant)
Algorithm 2' (rightside): Add-and-double-always method from the least significant bit (SPA-resistant)

Note that in Algorithm 1' and 2', there are no computational advantage even if we use the NAF because we have to compute both ECADD and ECDBL for each bit.

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 elliptic curve points).

Another countermeasure is to establish the indistinguishability between an ECADD and an ECDBL. Joye, Quisquater and 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, Sma01]. As we discussed above, 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.

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.

Coron also proposed three countermeasures, but Okeya and Sakurai showed the bias in his 1st and 2nd countermeasures. They asserted that Coron's 3rd method is secure against the DPA [OS00]. The key idea of Coron's 3rd countermeasure for the projective coordinate is as follows. Note that in the projective coordinate, we require 1 inversion and 2 multiplications in the definition fields to pull back from the projective point $(X_d : Y_d : Z_d)$ to the affine point (x_d, y_d). Let $P = (X : Y : Z)$ be a base point in a projective coordinate. Then $(X : Y : Z)$ equals to $(rX : rY : rZ)$ for all $r \in K$. If we randomize a base point with r before starting the scalar multiplication, the side information for the statistic analysis will be randomized. This countermeasure requires only three multiplications before the scalar multiplication, and no extra cost after the scalar multiplication.

The other enhancement method 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. Let $(X'_d : Y'_d : Z'_d)$ be the point after computing the scalar multiplication. The point (x_d, y_d) is pulled back to the original curve by computing $r^{-2}X'_d$ and $r^{-3}Y'_s$. This method requires 3 squaring and 5 multiplications for the randomizing the point P, and 1 squaring, 3 multiplications, and 1 inversion for pulling back to the original curve. Joye-Tymen method can choose the Z-coordinate equal to 1 during the computation of the scalar multiplication and it improves the efficiency of the scalar multiplication in some cases.

4 Our Proposed Algorithm

We explain our proposed algorithm for the scalar multiplication 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 SPA compared with Coron's SPA-resistant algorithm (Algorithm 1'). We also establish the addition formulas, which only use the x-coordinate of the points, for the Weierstrass form curves.

4.1 Addition Chain

We describe our proposed addition chain in the following:

```
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: Our 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 $d[i] = 1$. We prove the correctness of our proposed algorithm in the following.

Theorem 1 *Algorithm 3, on input a point P and an integer $d > 2$, outputs the correct value of the scalar multiplication $d * P$.*

Proof: When we write $Q[0], Q[1]$, it means that $Q[0]$ in Step 5 and $Q[1]$ in Step 6 of Algorithm 3 in the following. The loop of Step 2 generates a sequence

$$(Q[0], Q[1])_{n-2}, (Q[0], Q[1])_{n-3}, ..., (Q[0], Q[1])_1, (Q[0], Q[1])_0, \tag{3}$$

from the bit sequence $d[n-2], d[n-3], ..., d[1], d[0]$. At first we prove $Q[1] = Q[0] + P$ for each $(Q[0], Q[1])_i, i = 0, 1, .., n-2$, by the induction for the number of the sequence. For $n = 2$ we have only one loop in Step 3 and we have two cases $d[0] = 0$ or 1. Then we obtain $Q[0] = 2 * P, Q[1] = 3 * P$ for $d[0] = 0$, and $Q[0] = 3 * P, Q[1] = 4 * P$ for $d[0] = 1$. The fact $Q[1] = Q[0] + P$ is correct for $n = 2$. Next, we assume that $Q[1] = Q[0] + P$ up to $n = k$.

In this case we have $R[1] = R[0] + P$, where $(Q[0], Q[1])_1 = (R[0], R[1])$. For $n = k + 1$ we also have two cases $d[0] = 0$ or 1. Then we obtain $Q[0] = 2 * R[0], Q[1] = 2 * R[0] + P$ for $d[0] = 0$, and $Q[0] = 2 * R[0] + P, Q[1] = 2 * R[0] + 2 * P$ for $d[0] = 1$. The fact $Q[1] = Q[0] + P$ is correct for $n = k + 1$. Thus we proved that $Q[1] = Q[0] + P$ for each $(Q[0], Q[1])_i, i = 0, 1, .., n - 2$.

Next, we prove that $Q[0]$ is equivalent to $Q[0]$ in Step 4 of Algorithm 1 ($Q[0]$ in Step 5 of Algorithm 2) for each loop of $d[i], (i = 0, 1, .., n-2)$. In each loop of $d[i]$, for given $Q[0], Q[1]$, the new $Q[0]$ is computed as follows: ECDBL($Q[0]$) for $d[i] = 0$ and ECADD($Q[0], Q[1]$) $=$ $Q[0] + (Q[0] + P) = 2 * Q[0] + P =$ ECADD(ECDBL($Q[0]$)$, P$) for $d[i] = 1$. On the other hand, in each loop of $d[i]$ in Algorithm 1, for given $Q[0]$, the new $Q[0]$ is computed as follows: ECDBL($Q[0]$) for $d[i] = 0$ and ECADD(ECDBL($Q[0]$)$, P$) for $d[i] = 1$. They are completely the same computations. Thus we can conclude that the output $d * P$ is correct. ■

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 2 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: First, note that ECADD and ECDBL of each loop of Algorithm 2' can be computed in parallel. Algorithm 2' then requires only n ECADDs with two processors. However, Algorithm 1' cannot be parallelized in this sense, because its loop is constructed from the most significant bit, and the output of ECADD requires the output of ECDBL in each loop. The addition chain of Algorithm 3 is also constructed from the most significant bit, but we can compute the loop of Algorithm 3 in parallel.

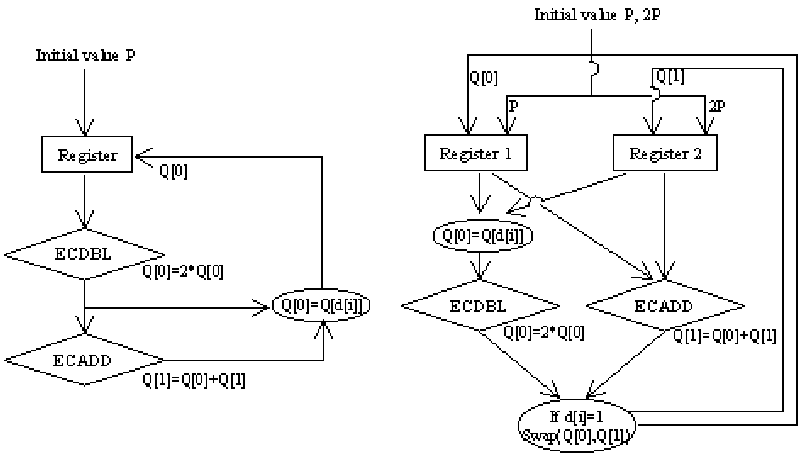

Figure 1: Algorithm 1' (left), the parallel implementation of Algorithm 3 (right)

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. Algorithm 1' always requires $(n-1)$ ECDBLs and $(n-1)$ ECADDs. The improvement of Algorithm 3 from Algorithm 1' is $(n-1)$ ECDBLs.

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.

Theorem 3 *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.*

Proof: The differences between Algorithm 1' and Algorithm 3 are Step 5 and Step 6 in Algorithm 3. In the steps, if $d[i] = 0$, we assign the $Q[0] = Q[2]$ and $Q[1] = Q[1]$, otherwise, we assign $Q[0] = Q[1]$ and $Q[1] = Q[2]$. We can modify the steps as follows:

```
S1: If d[i] = 1 then SWAP(Q[2],Q[1])
S2: Q[0] = Q[2]
S3: Q[1] = Q[1]
```

SWAP is a function to swap two variables. Only Step S1 depends on $d[i]$. If the power to execute SWAP is negligible, Algorithm 3 is as secure as Algorithm 1' against the SPA. ■

Next, an SPA-resistant scheme can be converted to a DPA-resistant scheme using Coron's 3rd or the Joye-Tymen's countermeasure as we discussed in the previous section. Thus, we have the following corollary.

Corollary 4 *Algorithm 3 with Coron's 3rd or Joye-Tymen's countermeasure is as secure as Algorithm 1' against the DPA, if we use a computing architecture whose swapping power of two variables is negligible.*

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.

4-Parallel Computation: When 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. There are several scalar multiplications using a pre-computed table [Gor98]. In this paper we are interested in a scalar multiplication with a very small table. The method that uses a very small table is proposed by Lim-Lee [LL94]. The simplest case of the Lim-Lee algorithm requires only one pre-computed value and its improvement over the binary method is 7/12 on average. We review it in the following. Let $d = d_{n-1}2^{n-1} + d_{n-2}2^{n-2} + \ldots + d_1 2 + d_0$ be the binary representation of d with $d_{n-1} = 1$. Let $k = \lfloor n/2 \rfloor$. The exponent d is represented as

$$d = 2^k(f[k]2^k + f[k-1]2^{k-1} + ... + f[0]2^0) + (e[k]2^k + e[k-1]2^{k-1} + ... + e[0]2^0), \quad (4)$$

where $e[i] = d[i], f[i] = d[i+k]$ for $i = 0, 1, ..., k-1$, $f[k] = e[k] = 0$ for even n, and $f[k] = 1, e[k] = 0$ for odd n. Then we obtain $d * P = \sum_{i=0}^{k} (2^i) * (e[i] * P + f[i] * ((2^k) * P))$ and $d * P$ can be computed like in the binary method. This method pre-computes the point $(2^k) * P$ and it is applied for only scalar exponentiations of the fixed based P. We modify the Lim-Lee method to be able to compute in parallel and to be secure against the SPA. The proposed algorithm carries 4 auxiliary variables $Q[0][0], Q[0][1], Q[1][0], Q[0][1]$, which are related with

$$Q[0][1] = Q[0][0] + P, Q[1][0] = Q[0][0] + 2^k * P, Q[1][1] = Q[0][0] + P + 2^k * P, \quad (5)$$

The proposed algorithm is as follows:

```
INPUT d, P, (2^k)*P, (k, b (= n+1 mod 2))
OUTPUT d*P
Q[0] = (2^k)*P, Q[1] = (2^k)*P + P
Q[0][0] = Q[e[k-b]]
Q[0][1] = Q[0][0] + P
Q[1][0] = Q[0][0] + (2^k)*P
Q[1][1] = Q[0][0] + P + (2^k)*P
for i = k-1-b down to 0
  Q[f[i]+0][e[i]+0] = ECDBL(Q[f[i]][e[i]])
  Q[f[i]+0][e[i]+1] = ECADD(Q[f[i]][e[i]],Q[f[i]+0][e[i]+1])
  Q[f[i]+1][e[i]+0] = ECADD(Q[f[i]][e[i]],Q[f[i]+1][e[i]+0])
  Q[f[i]+1][e[i]+1] = ECADD(Q[f[i]][e[i]],Q[f[i]+1][e[i]+1])
return Q[0][0]
```

Algorithm 4: Our proposed addition chain II (SPA resistant)

Theorem 5 *Algorithm 4, on input a point P and an integer $d > 2$, outputs the correct value of the scalar multiplication $d * P$.*

Proof: We prove the theorem for even n. The theorem for odd n can be proven similarly. In this case we have $f[k] = e[k] = 0$, $f[k-1] = 1$, and $b = n + 1 \bmod 2 = 1$. In general $Q[e[k-1]]$ is not one, and we assign $Q[0][0] = Q[k-1]$ in step 2 as the initial value for the addition chain. Then $Q[0][1], Q[1][0], Q[1][1]$ are assigned in step 3,4,5 so that equation (5) should hold. In step 6 we carry the 4 registers $Q[0][0], Q[0][1], Q[1][0], Q[1][1]$, and compute a loop with length $k-2$. Let $Q_i[0][0], Q_i[0][1], Q_i[1][0], Q_i[1][1]$ be the four registers of i-th loop. For each i we have the following relationship:

$$\begin{aligned}
Q_i[0][0] &= (Q_{i-1}[0][0] + Q_{i-1}[e[i-1]][f[i-1]]) \\
Q_i[0][1] &= (Q_{i-1}[0][0] + Q_{i-1}[e[i-1]][f[i-1]]) + P \\
Q_i[1][0] &= (Q_{i-1}[0][0] + Q_{i-1}[e[i-1]][f[i-1]]) + (2^k) * P \\
Q_i[1][1] &= (Q_{i-1}[0][0] + Q_{i-1}[e[i-1]][f[i-1]]) + P + (2^k) * P.
\end{aligned}$$

Therefore the four registers in the loop inherit the relation of equation (5). Moreover the first register $Q_i[0][0]$ is equal to $2 * Q_{i-1}[0][0] + e[i-1] * P + f[i-1] * ((2^k) * P)$, which corresponds to the intermediate value of the Lim-Lee algorithm. Therefore, the output of Algorithm 4 is a correct value of $d * P$. ■

We can compute from Step 7 to Step 10 in parallel, then the latency of each loop is the time for computing ECADDs. The total number of loops is at most $n/2 - 1$, where n is the bit-length of d. Therefore Algorithm 4 can be computed at most $(n/2 + 3)$ ECADDs with 4 processors. It is about two times faster than Algorithm 3. Moreover, the security against the SPA can be discussed in the same way like Theorem 1 for Algorithm 3. If we use a computing architecture whose swapping powers of four variables are negligible, then Algorithm 4 is secure against the SPA. It is possible to apply Coron's 3rd or Joye-Tymen's countermeasure to make Algorithm 4 secure against the DPA.

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:

$$x_3 \cdot x_3' = \frac{(x_1x_2 - a)^2 - 4b(x_1 + x_2)}{(x_1 - x_2)^2}, \tag{6}$$

$$x_3 + x_3' = \frac{2(x_1 + x_2)(x_1x_2 + a) + 4b}{(x_1 - x_2)^2}. \tag{7}$$

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

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

Thus the x-coordinates of both P_3 and P_4 can be computed just form the x-coordinates of the points P_1, P_2, P_3'. We call this method the multiplicative (additive) x-coordinate-only method. The x-coordinate-only methods for a scalar multiplication 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 of two points $P_3' = P_1 - P_2$. 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. Similarly, in each loop of Algorithm 4, the differences of the points for computing an ECADD are known by equation (5). Therefore, we can assume that the difference $P_2 - P_1$ for input values of ECADD(P_1, P_2) of Algorithm 3 (or Algorithm 4) are always known. On the contrary, in order to know that of Algorithm 2' we need extra computation. The x-coordinate-only methods for Algorithm 2' have no computational advantage.

When we apply the x-coordinate-only methods to Algorithm 3 (or Algorithm 4), the output is only the x-coordinate of $d*P$. This is enough for some cryptographic applications such as a key exchange scheme and an encryption/decryption scheme [SEC]. But other applications also require the y-coordinate of $d*P$ in the verification of a signature scheme [SEC]. However, the y-coordinate of $d*P$ is easily obtained in the following way: Let $P = (x, y)$, $d*P = (x_d, y_d)$, $(d+1)*P = (x_{d+1}, y_{d+1})$. In the final step of Algorithm 3 (or Algorithm 4), we know the values of x,y,x_d which is assigned to $Q[0]$ and x_{d+1} which is assgned to $Q[1]$. Using a standard addition formula (2), we obtain the relation

$$y_d = \frac{y^2 + x_d^3 + ax_d + b - (x - x_d)^2(x + x_d + x_{d+1})}{2y}.$$

and y_d is recoverd from this relation. This y-recovering technique was originally introduced by Agnew et al. for curves over $\mathbb{F}_{2^m}$ [AMV93].

In the implementation of the x-coordinate-only methods, the projective coordinate system offers a faster computation. In this system, equations (6), (7) and (8) turn to

$$\frac{X_3}{Z_3} = \frac{Z_3'}{X_3'}\frac{(X_1X_2 - aZ_1Z_2)^2 - 4bZ_1Z_2(X_1Z_2 + X_2Z_1)}{(X_1Z_2 - X_2Z_1)^2}, \tag{9}$$

$$\frac{X_3}{Z_3} = \frac{2(X_1Z_2 + X_2Z_1)(X_1X_2 + aZ_1Z_2) + 4bZ_1^2Z_2^2}{(X_1Z_2 - X_2Z_1)^2} - \frac{X_3'}{Z_3'}, \tag{10}$$

$$\frac{X_4}{Z_4} = \frac{(X_1^2 - aZ_1^2)^2 - 8bX_1Z_1^3}{4(X_1Z_1(X_1^2 + aZ_1^2) + bZ_1^4)}. \tag{11}$$

The computing times for (9),(10),(11) 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 to compute (9), (10), (11) are listed in the appendix. After the final step of Algorithm 3 (or Algorithm 4), $d*P = (X_d' : Y_d' : Z_d')$ is computed from $x, y, X_d, Z_d, X_{d+1}, Z_{d+1}$ by

$$\begin{aligned}
X_d' &= 2yZ_d^2Z_{d+1}X_d, \\
Y_d' &= Z_{d+1}(y^2Z_d^3 + X_d^3 + aX_dZ_d^2 + bZ_d^3) \\
&\quad - (xZ_d - X_d)^2(xZ_dZ_{d+1} + X_dZ_{d+1} + X_{d+1}Z_d), \\
Z_d' &= 2yZ_d^2Z_{d+1}Z_d.
\end{aligned}$$

The computing time is $14M + 3S$ for $(X'_d : Y'_d : Z'_d)$ and $2M + 1I$ for y-recovering. The concrete algorithm is in the appendix. [2]

4.3 Capsulation

In Algorithm 3, both ECADD and ECDBL are always computed for each bit $d[i]$. When we use one processor, we can encapsulate ECADD and ECDBL as one formula so that their intermediate variables may be shared and the efficiency may be reduced.

In fact, with our additive addition formula (10) and (11), we can construct a faster scalar computation. By multiplying $Z_2^4(\neq 0)$ with both its denominator and numerator in (11), we obtain

$$\begin{aligned}\frac{X_4}{Z_4} &= \frac{(X_1^2 - aZ_1^2)^2 - 8bX_1Z_1^3}{4(X_1Z_1(X_1^2 + aZ_1^2) + bZ_1^4)} \\ &= \frac{(X_1^2Z_2^2 - aZ_1^2Z_2^2)^2 - 8bX_1Z_1^3Z_2^4}{4Z_1Z_2(X_1Z_2(X_1^2Z_2^2 + aZ_1^2Z_2^2) + bZ_1^3Z_2^3)}.\end{aligned}$$

Because X_1Z_2, Z_1Z_2, aZ_1Z_2, $Z_1^2Z_2^2$, $bZ_1^2Z_2^2$ are computed in $\mathtt{ECADD}_a^{(x)}$ (Formula 2), we can share the intermediate variables in $\mathtt{ECADD}_a^{(x)}$ (Formula 2) and in $\mathtt{ECDBL}^{(x)}$ (Formula 3), if we encapsulate them into a new function $\mathtt{ECADDDBL}_a^{(x)}$. This function computes $P_1 + P_2$ and $2 * P_2$ on input P_1, P_2. Here, we can reduce $1M + 1S$ and the total efficiency is $\mathtt{ECADDDBL}_a^{(x)} = 15M + 4S$. If we choose $Z'_3 = 1$ then $\mathtt{ECADDDBL}_{a(Z'_3=1)}^{(x)} = 13M + 4S$. In appendix we show the concrete algorithm for $\mathtt{ECADDDBL}_a^{(x)}$ (Formula 5). For ordered input (P_1, P_2), Formula $\mathtt{ECADDDBL}_a^{(x)}$ outputs the ordered points $(P_1 + P_2, 2 * P_2)$.

A similar function, which outputs $P_1 + P_2$ and $2 * P_1$ is easily constructed by swapping the input values P_1 and P_2 in Formula 5 based on $d[i]$. Indeed, the x-coordinate-only method requires $P'_3 = P_1 - P_2$ in order to compute $P_1 + P_2$. We choose the fixed value $P'_3 = P$ for the $(m, m+1)$-method. If we compute $swap(P_1, P_2)$ for computing $2 * P_2$, we have to set $P'_3 = -P$. However, the projective coordinate is used for x-coordinate-only method, both $X(-P) = X(P)$ and $Z(-P) = Z(P)$ hold, and we do not have to choose $P_3 = -P$. Therefore, for ordered input (P_2, P_1), Formula $\mathtt{ECADDDBL}_a$ outputs the ordered points $(P_1 + P_2, 2 * P_1)$. Thus we can establish the following scalar multiplication:

```
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[d[i]+1],Q[d[i]]) = ECADDDBL(Q[d[i]+1],Q[d[i]])
 4: return Q[0]
```

Algorithm 3': Improved addition chain (SPA resistant)

[2] A similar discussion for y-recovering on Montgomery form is found in [OS01]. However, Algorithm 2 and Algorithm 4 in [OS01] doesn't output the expected values. The formulas for them must be $X_d^{rec} = 4ByX_{d+1}Z_{d+1}Z_dX_d$, $Y_d^{rec} = Z_{d+1}^2U^2 - X_{d+1}^2V^2$, $Z_d^{rec} = 4ByX_{d+1}Z_{d+1}Z_d^2$ and y-recovering needs only $13M + 1I$, which is faster than Algorithm 3 in [OS01].

Here $d[i]+1$ is computed as modulo 2. We denote by ECADDDBL Formula $\mathtt{ECADDDBL}_a^{(x)}$. Both the input and the output for $\mathtt{ECADDBL}_a^{(x)}$ are ordered pairs, which are $(Q[0], Q[1])$ for $d[i]=1$ and $(Q[1], Q[0])$ for $d[i]=0$. Formula $\mathtt{ECADDDBL}_a^{(x)}$ outputs the pair $(Q[0]+Q[1], 2*Q(d[i]))$, which is equivalent to the outputs of $\mathtt{ECDBL}^{(x)}$ and $\mathtt{ECADD}_a^{(x)}$ for Algorithm 3, respectively. Then the output values are assigned as $(Q[1], Q[0])$ for $d[i]=0$ and $(Q[0], Q[1])$ for $d[i]=1$, which is equivalent to the outputs of i-th loop of Algorithm 3.

We use the same swapping function in the proof of Theorem 2, and the scalar multiplication Algorithm 3' using Formula 5 is secure against the SPA in the sense of Theorem 2. It is possible to apply Coron's 3rd or Joye-Tymen's countermeasure to make Algorithm 4 secure against the DPA. Thus, we have the following theorem.

Theorem 6 *Algorithm 3' with Coron's 3rd or Joye-Tymen's countermeasure is as secure as Algorithm 1' against the DPA, if we use a computing architecture whose swapping power of two variables is negligible.*

5 Comparison

In this section, we compare the computing times of a scalar multiplication resistant against the SCA. As a result, we show that our proposed algorithm establishes a faster scalar multiplication. The improvement of our scalar multiplication over the previously fastest method is about 37% for two processors and 17.1% 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 assumed to be DPA-resistant using Coron's 3rd countermeasure or Joye-Tymen's countermeasure, which are described in Section 3. 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 information (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. Note that one inversion is always required in order to convert a point from the projective coordinates to the affine coordinates. In the estimation, we include the times for randomization by Coron's 3rd countermeasure or Joye-Tymen's countermeasure, and the times for recovering the y-coordinate in the x-coordinate-only method are also included. [3] 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: In Table 2, we summarize the estimated running time using a single processor. Algorithm 3/Joye-Tymen/Capsulation with the additive x-coordinate-only method is the fastest of all scalar multiplications ($2641.8M$). The previously fastest algorithm was Algorithm 1'/Joye-Tymen with the Jacobian coordinate system $\mathcal{J}$ ($3093.2M$). The improvement of the proposed algorithm over it is about 17.1%.

[3] These algorithms may contain several inversions, but we can compute them by only one inversion and several multiplications instead. For example, we estimate two inversions $x^{-1}, y^{-1} \in \mathbb{F}_p$ as the cost for computing $z=(xy)^{-1}$, $x^{-1}=zy$ and $y^{-1}=zx$, that is, one inversion and three multiplications.

	Addition formula	Computing Time		
		Total	$n = 160$	
Algorithm 1' /Coron 3rd	$\mathcal{P}$	$(19n-15)M+(7n-7)S+1I$	$3025M+1113S+1I$	$(3945.4M)$
	$\mathcal{J}$	$(16n-10)M+(10n-8)S+1I$	$2550M+1592S+1I$	$(3853.6M)$
	$\mathcal{J}^C$	$(16n-10)M+(9n-8)S+1I$	$2550M+1432S+1I$	$(3725.6M)$
	$\mathcal{J}^m$	$(17n-10)M+(10n-7)S+1I$	$2710M+1593S+1I$	$(4014.4M)$
Algorithm 1' /Joye-Tymen	$\mathcal{P}$	$(16n-8)M+(7n-5)S+1I$	$2552M+1115S+1I$	$(3474.0M)$
	$\mathcal{J}$	$(12n-4)M+(9n-6)S+1I$	$1916M+1434S+1I$	$(3093.2M)$
	$\mathcal{J}^C$	$(13n-5)M+(9n-6)S+1I$	$2075M+1434S+1I$	$(3252.2M)$
	$\mathcal{J}^m$	$(13n-5)M+(9n-6)S+1I$	$2075M+1434S+1I$	$(3252.2M)$
Algorithm 2' /Coron 3rd	$\mathcal{P}$	$(19n+4)M+7nS+1I$	$3044M+1120S+1I$	$(3970.0M)$
	$\mathcal{J}$	$(16n+6)M+(10n+2)S+1I$	$2566M+1602S+1I$	$(3877.6M)$
	$\mathcal{J}^C$	$(16n+6)M+(9n+1)S+1I$	$2566M+1441S+1I$	$(3748.4M)$
	$\mathcal{J}^m$	$(17n+7)M+(10n+3)S+1I$	$2727M+1603S+1I$	$(4039.4M)$
Algorithm 2' /Joye-Tymen	$\mathcal{P}$	$(19n+8)M+(7n+2)S+1I$	$3048M+1122S+1I$	$(3975.6M)$
	$\mathcal{J}$	$(16n+8)M+(10n+3)S+1I$	$2568M+1603S+1I$	$(3880.4M)$
	$\mathcal{J}^C$	$(16n+8)M+(9n+3)S+1I$	$2568M+1443S+1I$	$(3752.4M)$
	$\mathcal{J}^m$	$(17n+8)M+(10n+3)S+1I$	$2728M+1603S+1I$	$(4040.4M)$
Algorithm 3 /Coron 3rd	$\mathcal{P}$	$(19n-8)M+(7n-2)S+1I$	$3032M+1118S+1I$	$(3956.4M)$
	$\mathcal{J}$	$(16n-6)M+(10n-2)S+1I$	$2554M+1598S+1I$	$(3862.4M)$
	$\mathcal{J}^C$	$(16n-5)M+(9n-2)S+1I$	$2555M+1438S+1I$	$(3735.4M)$
	$\mathcal{J}^m$	$(17n-6)M+(10n-3)S+1I$	$2714M+1597S+1I$	$(4021.6M)$
	$\boldsymbol{x}$ **(mul)**	$\boldsymbol{(15n+8)M+(5n+1)S+1I}$	$\boldsymbol{2408M+801S+1I}$	$\boldsymbol{(3078.6M)}$
	$\boldsymbol{x}$ **(add)**	$\boldsymbol{(16n+7)M+(5n+1)S+1I}$	$\boldsymbol{2567M+801S+1I}$	$\boldsymbol{(3237.6M)}$
(Algorithm 3')	$\boldsymbol{x^c}$ **(add)**	$\boldsymbol{(15n+8)M+(4n+2)S+1I}$	$\boldsymbol{2408M+642S+1I}$	$\boldsymbol{(2951.6M)}$
Algorithm 3 /Joye-Tymen	$\mathcal{P}$	$(19n-4)M+7nS+1I$	$3036M+1120S+1I$	$(3962.0M)$
	$\mathcal{J}$	$(16n-4)M+(10n-1)S+1I$	$2556M+1599S+1I$	$(3865.2M)$
	$\mathcal{J}^C$	$(16n-3)M+9nS+1I$	$2557M+1440S+1I$	$(3739.0M)$
	$\mathcal{J}^m$	$(17n-5)M+(10n-3)S+1I$	$2715M+1597S+1I$	$(4022.6M)$
	$\boldsymbol{x}$ **(mul)**	$\boldsymbol{(14n+14)M+(5n+5)S+1I}$	$\boldsymbol{2254M+805S+1I}$	$\boldsymbol{(2928.0M)}$
	$\boldsymbol{x}$ **(add)**	$\boldsymbol{(14n+14)M+(5n+5)S+1I}$	$\boldsymbol{2254M+805S+1I}$	$\boldsymbol{(2928.0M)}$
(Algorithm 3')	$\boldsymbol{x^c}$ **(add)**	$\boldsymbol{(13n+15)M+(4n+6)S+1I}$	$\boldsymbol{2095M+646S+1I}$	$\boldsymbol{(2641.8M)}$

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

In order to demonstrate the efficiency of our algorithm, we implemented our proposed algorithm and the previously fastest algorithm on a Celeron 500 MHz using the LiDIA library [LiDIA]. It should be emphasized here that our implementation was not optimized for cryptographic purposes — it is only intended to provide a comparison. The improvement is about 20%. The results are showed in Table 3.

Previously fastest scheme	Algorithm 1'/Joye-Tymen ($\mathcal{J}$)	25.5 ms
Proposed scheme	Algorithm 3 /Joye-Tymen (x)/Capsulation	21.5 ms

Table 3: Computing times on a Celeron 500 MHz using LiDIA (a single processor)

Parallel Case: In Table 4, we summarize the estimated running time using two parallel processors. 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 system $\mathcal{J}^C$ $(2180.6M)$. Algorithm 3/Joye-Tymen with x-coordinate-only methods provides the fastest multiplication $(1592.4M)$. The improvement of the proposed algorithm over it is about 37%.

	Addition formula	Computing Time		
		Total	$n = 160$	
Algorithm 2' /Coron 3rd	$\mathcal{P}$	$(12n+4)M + 2nS + 1I$	$1924M + 320S + 1I$	$(2210.0M)$
	$\mathcal{J}$	$(12n+6)M + (4n+2)S + 1I$	$1926M + 642S + 1I$	$(2469.6M)$
	$\mathcal{J}^{\mathcal{C}}$	$(11n+6)M + (3n+1)S + 1I$	$1766M + 481S + 1I$	$(2180.6M)$
	$\mathcal{J}^{m}$	$(13n+7)M + (6n+3)S + 1I$	$2087M + 963S + 1I$	$(2887.4M)$
Algorithm 2' /Joye-Tymen	$\mathcal{P}$	$(12n+8)M + (2n+2)S + 1I$	$1928M + 322S + 1I$	$(2215.6M)$
	$\mathcal{J}$	$(12n+8)M + (4n+3)S + 1I$	$1928M + 643S + 1I$	$(2472.4M)$
	$\mathcal{J}^{\mathcal{C}}$	$(11n+8)M + (3n+3)S + 1I$	$1768M + 483S + 1I$	$(2184.4M)$
	$\mathcal{J}^{m}$	$(13n+8)M + (6n+3)S + 1I$	$2088M + 963S + 1I$	$(2898.4M)$
Algorithm 3 /Coron 3rd	$\mathcal{P}$	$(12n-1)M + (2n+3)S + 1I$	$1919M + 323S + 1I$	$(2207.4M)$
	$\mathcal{J}$	$(12n-2)M + (4n+4)S + 1I$	$1918M + 644S + 1I$	$(2463.2M)$
	$\mathcal{J}^{\mathcal{C}}$	$11nM + (3n+4)S + 1I$	$1760M + 484S + 1I$	$(2177.2M)$
	$\mathcal{J}^{m}$	$(13n-2)M + (6n+1)S + 1I$	$2078M + 961S + 1I$	$(2876.8M)$
	x (mul)	$\boldsymbol{(9n+14)M + (2n+4)S + 1I}$	$\boldsymbol{1454M + 324S + 1I}$	$\boldsymbol{(1743.2M)}$
	x (add)	$\boldsymbol{(10n+13)M + (2n+4)S + 1I}$	$\boldsymbol{1613M + 324S + 1I}$	$\boldsymbol{(1902.2M)}$
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)	$\boldsymbol{(8n+20)M + (2n+8)S + 1I}$	$\boldsymbol{1300M + 328S + 1I}$	$\boldsymbol{(1592.4M)}$
	x (add)	$\boldsymbol{(8n+20)M + (2n+8)S + 1I}$	$\boldsymbol{1300M + 328S + 1I}$	$\boldsymbol{(1592.4M)}$

Table 4: Computing times of a scalar multiplication (two parallel processors)

6 Concluding Remarks

In this paper, we proposed a fast scalar multiplication algorithm which improves both the addition chain and the addition formula. Our addition chain requires no table look-up (Algorithm 3) or a very small table (Algorithm 4). The prominent property of our proposed chain is the capability of parallel computations. We showed our chain is secure against the SCA (Theorem 2). We also proposed addition formulas, the x-coordinate-only method (Section 4.2). By combination of these methods, we established the fastest scalar multiplication among the previously reported algorithms. The improvement of our scalar multiplication over the previous fastest method is about 37.0% with two processors and 17.1% for a single processor, as reported in Section 5. Our proposed methods are suitable for the implementation over smart cards.

Acknowledgments. We would like to thank Edlyn Teske, Bodo Möller, and Evangelos Karatsiolis for their valuable comments.

References

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

[ANSI] ANSI X9.62, Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA), draft, 1998.

[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, Springer-Verlag, 1999.

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

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

[Gor98] D.Gordon, "A survey of fast exponentiation methods", J. Algorithms, vol.27, pp.129-146, 1998.

[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, Springer-Verlag, 2001.

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

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

[KJJ99] C.Kocher, J.Jaffe and B.Jun, "Differential power analysis", *Crypto'99*, LNCS 1666, pp.388-397, Springer-Verlag, 1999.

[LiDIA] LiDIA, A C++ Library For Computational Number Theory, Technische Universität Darmstadt, `http://www.informatik.tu-darmstadt.de/TI/LiDIA/`

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

[LL94] C. Lim and P. Lee, "More flexible exponentiation with precomputation", *Crypto'94*, LNCS 839, p.95-107, Springer-Verlag, 1994.

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

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

[MO90] F. Morain and J. Olivos, "Speeding up the computation on an elliptic curve using addition-subtraction chains", Inform. Theory Appl. 24, pp.531-543, 2000.

[NIST] National Institute of Standards and Technology, Recommended Elliptic Curves for Federal Government Use, in the appendix of FIPS 186-2, Available from `http://csrc.nist.gov/publication/fips/fips186-2/fips186-2.pdf`

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

[OMS01] K.Okeya, K.Miyazaki and K.Sakurai, "A fast scalar multiplication method with randomized projective coordinates on a Montgomery-form elliptic curve secure against side channel attacks", *ICISC 2001*, LNCS, to appear.

[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, Springer-Verlag, 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, Springer-Verlag, 2001.

[Sma01] N.Smart, "The Hessian form of an elliptic curve", *CHES2001*, LNCS 2162, pp.118-125, Springer-Verlag, 2001.
[SEC] Standards for Efficient Cryptography Group (SECG), Specification of Standards for Efficient Cryptography. Available from `http://www.secg.org`
[WAP] Wireless Application Protocol (WAP) Forum, Wireless Transport Layer Security (WTLS) Specification. Available from `http://www.wapforum.org`

Appendix

We show the concrete algorithms for computing $\mathtt{ECDBL}^{(x)}$, $\mathtt{ECADD}_m^{(x)}$, $\mathtt{ECADD}_a^{(x)}$, $\mathtt{ECADDDBL}_a^{(x)}$, and $d * P = (X'_d : Y'_d : Z'_d)$ which are described in Section 4. In order to estimate the efficiency, we use three notations $\times, \cdot, *$ for the multiplication of the definition field K. The notation $\times$ is a standard multiplication in K. The notation $\cdot$ is a multiplication executed in negligible time becaus one of the input is a small constant. The notation $*$ is also calculated in negligible time if we choose $Z'_3 = 1$.

$$
\begin{aligned}
&T_1 \leftarrow X_1 \times X_2\\
&T_2 \leftarrow Z_1 \times Z_2\\
&T_3 \leftarrow X_1 \times Z_2\\
&T_4 \leftarrow X_2 \times Z_1\\
&T_5 \leftarrow a \times T_2 \ (= aZ_1Z_2)\\
&T_6 \leftarrow T_1 - T_5 \ (= X_1X_2 - aZ_1Z_2)\\
&T_7 \leftarrow T_6^2 \ (= (X_1X_2 - aZ_1Z_2)^2)\\
&T_8 \leftarrow b \times T_2 \ (= bZ_1Z_2)\\
&T_9 \leftarrow 4 \cdot T_8 \ (= 4bZ_1Z_2)\\
&T_{10} \leftarrow T_3 + T_4 \ (= X_1Z_2 + X_2Z_1)\\
&T_{11} \leftarrow T_9 \times T_{10} \ (= 4bZ_1Z_2(X_1Z_2 + X_2Z_1))\\
&T_{12} \leftarrow T_7 - T_{11} \ (= (X_1X_2 - aZ_1Z_2)^2 - 4bZ_1Z_2(X_1Z_2 + X_2Z_1))\\
&X_3 \leftarrow Z'_3 * T_{12}\\
&T_{13} \leftarrow T_3 - T_4 \ (= X_1Z_2 - X_2Z_1)\\
&T_{14} \leftarrow T_{13}^2 \ (= (X_1Z_2 - X_2Z_1)^2)\\
&Z_3 \leftarrow X'_3 \times T_{14}
\end{aligned}
$$

Formula 1: Computing $\mathtt{ECADD}_m^{(x)}$ ($\cdot$ is negligible, $*$ is negligible if $Z'_3 = 1$)

$$
\begin{aligned}
&T_1 \leftarrow X_1 \times X_2\\
&T_2 \leftarrow Z_1 \times Z_2\\
&T_3 \leftarrow X_1 \times Z_2\\
&T_4 \leftarrow X_2 \times Z_1\\
&T_5 \leftarrow T_3 + T_4 \ (= X_1Z_2 + X_2Z_1)\\
&T_6 \leftarrow a \times T_2 \ (= aZ_1Z_2)\\
&T_7 \leftarrow T_1 + T_6 \ (= X_1X_2 + aZ_1Z_2)\\
&T_8 \leftarrow T_5 \times T_7 \ (= (X_1Z_2 + X_2Z_1)(X_1X_2 + aZ_1Z_2))\\
&T_9 \leftarrow 2 \cdot T_8 \ (= 2(X_1Z_2 + X_2Z_1)(X_1X_2 + aZ_1Z_2))\\
&T_{10} \leftarrow T_2^2 \ (= Z_1^2Z_2^2)\\
&T_{11} \leftarrow b \times T_{10} \ (bZ_1^2Z_2^2)\\
&T_{12} \leftarrow 4 \cdot T_{11} \ (= 4bZ_1^2Z_2^2)\\
&T_{13} \leftarrow T_9 + T_{12} \ (= 2(X_1Z_2 + X_2Z_1)(X_1X_2 + aZ_1Z_2) + 4bZ_1^2Z_2^2)\\
&T_{14} \leftarrow T_3 - T_4 \ (= X_1Z_2 - X_2Z_1)\\
&T_{15} \leftarrow T_{14}^2 \ (= (X_1Z_2 - X_2Z_1)^2)\\
&T_{16} \leftarrow Z_3' * T_{13}\\
&T_{17} \leftarrow X_3' \times T_{15}\\
&X_3 \leftarrow T_{16} - T_{17}\\
&Z_3 \leftarrow Z_3' * T_{15}
\end{aligned}
$$

Formula 2: Computing $\texttt{ECADD}_a^{(x)}$ ($\cdot$ is negligible, $*$ is negligible if $Z_3' = 1$)

$$
\begin{aligned}
&T_1 \leftarrow X_1^2\\
&T_2 \leftarrow Z_1^2\\
&T_3 \leftarrow a \times T_2 \ (= aZ_1^2)\\
&T_4 \leftarrow T_1 - T_3 \ (= X_1^2 - aZ_1^2)\\
&T_5 \leftarrow T_4^2 \ (= (X_1^2 - aZ_1^2)^2)\\
&T_6 \leftarrow b \times T_2 \ (= bZ_1^2)\\
&T_7 \leftarrow X_1 \times Z_1 \ (= X_1Z_1)\\
&T_8 \leftarrow T_6 \times T_7 \ (= bX_1Z_1^3)\\
&T_9 \leftarrow 8 \cdot T_8 \ (= 8bX_1Z_1^3)\\
&X_4 \leftarrow T_5 - T_9\\
&T_{10} \leftarrow T_1 + T_3 \ (= X_1^2 + aZ_1^2)\\
&T_{11} \leftarrow T_7 \times T_{10} \ (= X_1Z_1(X_1^2 + aZ_1^2))\\
&T_{12} \leftarrow T_6 \times T_2 \ (= bZ_1^4)\\
&T_{13} \leftarrow T_{11} + T_{12} \ (= X_1Z_1(X_1^2 + aZ_1^2) + bZ_1^4))\\
&Z_4 \leftarrow 4 \cdot T_{13}
\end{aligned}
$$

Formula 3: Computing $\texttt{ECDBL}^{(x)}$ ($\cdot$ is negligible)

$$
\begin{aligned}
&T_1 \leftarrow Z_d^2 \\
&T_2 \leftarrow T_1 \times Z_{d+1} \\
&T_3 \leftarrow y \times T_2 \ (= Z_d^2 Z_{d+1}) \\
&T_4 \leftarrow X_d \times T_3 \ (= y Z_d^2 Z_{d+1}) \\
&X'_d \leftarrow 2 \cdot T_4 \\
&T_5 \leftarrow Z_d \times T_3 \ (= y Z_d^2 Z_{d+1} Z_d) \\
&Z'_d \leftarrow 2 \cdot T_5 \\
&T_6 \leftarrow X_d^2 \\
&T_7 \leftarrow a \times T_1 \ (= a Z_d^2) \\
&T_8 \leftarrow T_6 + T_7 \ (= X_d^2 + a Z_d^2) \\
&T_9 \leftarrow X_d \times Z_{d+1} \\
&T_{10} \leftarrow T_9 \times T_8 \ (= X_d Z_{d+1}(X_d^2 + a Z_d^2)) \\
&T_{11} \leftarrow y \times T_5 \ (= y^2 Z_d^3 Z_{d+1}) \\
&T_{12} \leftarrow b \times Z_d \\
&T_{13} \leftarrow T_{12} \times T_2 \ (= b Z_d^3 Z_{d+1}) \\
&T_{14} \leftarrow x \times Z_d \\
&T_{15} \leftarrow T_{14} - X_d \ (= x Z_d - X_d) \\
&T_{16} \leftarrow T_{15}^2 \ (= (x Z_d - X_d)^2) \\
&T_{17} \leftarrow T_{14} \times Z_{d+1} \ (= x Z_d Z_{d+1}) \\
&T_{18} \leftarrow X_{d+1} \times Z_d \\
&T_{19} \leftarrow T_{17} + T_9 + T_{18} \\
&T_{20} \leftarrow T_{16} \times T_{19} \\
&Y'_d \leftarrow T_{10} + T_{11} + T_{13} - T_{20}
\end{aligned}
$$

Formula 4: Computing $d * P = (X'_d : Y'_d : Z'_d)$ ($\cdot$ is negligible)

$$
\begin{aligned}
&T_1 \leftarrow X_1 \times X_2 \\
&T_2 \leftarrow Z_1 \times Z_2 \\
&T_3 \leftarrow X_1 \times Z_2 \\
&T_4 \leftarrow X_2 \times Z_1 \\
&T_5 \leftarrow T_3 + T_4 \ (= X_1 Z_2 + X_2 Z_1) \\
&T_6 \leftarrow a \times T_2 \ (= a Z_1 Z_2) \\
&T_7 \leftarrow T_1 + T_6 \ (= X_1 X_2 + a Z_1 Z_2) \\
&T_8 \leftarrow T_5 \times T_7 \ (= (X_1 Z_2 + X_2 Z_1)(X_1 X_2 + a Z_1 Z_2)) \\
&T_9 \leftarrow 2 \cdot T_8 \ (= 2(X_1 Z_2 + X_2 Z_1)(X_1 X_2 + a Z_1 Z_2)) \\
&T_{10} \leftarrow T_2^2 \ (= Z_1^2 Z_2^2) \\
&T_{11} \leftarrow b \times T_{10} \ (b Z_1^2 Z_2^2) \\
&T_{12} \leftarrow 4 \cdot T_{11} \ (= 4b Z_1^2 Z_2^2) \\
&T_{13} \leftarrow T_9 + T_{12} \ (= 2(X_1 Z_2 + X_2 Z_1)(X_1 X_2 + a Z_1 Z_2) + 4b Z_1^2 Z_2^2) \\
&T_{14} \leftarrow T_3 - T_4 \ (= X_1 Z_2 - X_2 Z_1) \\
&T_{15} \leftarrow T_{14}^2 \ (= (X_1 Z_2 - X_2 Z_1)^2) \\
&T_{16} \leftarrow Z_3' * T_{13} \\
&T_{17} \leftarrow X_3' \times T_{15} \\
&X_3 \leftarrow T_{16} - T_{17} \\
&Z_3 \leftarrow Z_3' * T_{15} \\
&T_{21} \leftarrow T_3^2 \ (= X_1^2 Z_2^2) \\
&T_{22} \leftarrow T_6 \times T_2 \ (= a Z_1^2 Z_2^2) \\
&T_{23} \leftarrow T_{21} - T_{22} \ (= X_1^2 Z_2^2 - a Z_1^2 Z_2^2) \\
&T_{24} \leftarrow T_{23}^2 \ (= (X_1^2 Z_2^2 - a Z_1^2 Z_2^2)^2) \\
&T_{25} \leftarrow T_{11} \times T_2 \ (= b Z_1^3 Z_2^3) \\
&T_{26} \leftarrow T_{25} \times T_3 \ (= b X_1 Z_1^3 Z_2^4) \\
&T_{27} \leftarrow 8 \cdot T_{26} \ (= 8b X_1 Z_1^3 Z_2^4) \\
&X_4 \leftarrow T_{24} - T_{27} \\
&T_{28} \leftarrow T_{21} + T_{22} \ (= X_1^2 Z_2^2 + a Z_1^2 Z_2^2) \\
&T_{29} \leftarrow T_3 \times T_{28} \ (= X_1 Z_2 (X_1^2 Z_2^2 + a Z_1^2 Z_2^2)) \\
&T_{30} \leftarrow T_{29} + T_{25} \ (= X_1 Z_2 (X_1^2 Z_2^2 + a Z_1^2 Z_2^2) + b Z_1^3 Z_2^3) \\
&T_{31} \leftarrow T_2 \times T_{30} \ (= Z_1 Z_2 (X_1 Z_2 (X_1^2 Z_2^2 + a Z_1^2 Z_2^2) + b Z_1^3 Z_2^3)) \\
&Z_4 \leftarrow 4 \cdot T_{31}
\end{aligned}
$$

Formula 5: Computing $\texttt{ECADDDBL}_a^{(x)}$ ($\cdot$ is negligible)