

Technische Universität Darmstadt, Germany  
Department of Computer Science  
Cryptography and Computeralgebra

Fast Elliptic Curve Operation Costs and  
Comparison of Sole Inversion  
Precomputation Schemes

Julian Lehmann

Bachelor Thesis  
August 2008



Supervised by: Prof. Johannes Buchmann  
Dipl.-Math. Erik Dahmen

## Abstract

Our information society is faced with a dilemma: On the one hand the amount of digitally exchanged information increases every day. The mobile connectivity of each individual and the availability of data have become critical in many jobs. But also for private use, server-based approaches where data (e.g. media or emails) is stored centrally and used on demand replace local storage.

On the other hand possible abuse grows at the same pace as the amount of available data. Therefore questions like integrity, authenticity and confidentiality are of the utmost importance.

The science of securing communication channels in electronic environments is called cryptography. It bridges the above gap by enabling public information exchange in a secured environment that prohibits unrestricted access. More and more also very small devices contain information that has to be secured (e.g. passports containing biometrical data or health insurance cards containing disease patterns). For these devices, cryptography is performed on smart cards and similar devices that have a very restricted environment (e.g. memory, computational power). Due to the restriction of these devices, it is necessary to perform the cryptography efficiently and adapt to their very special environment. Using the mathematical notion of Elliptic Curves (EC) we can define a cryptosystem with the same level of security like previous cryptosystems but using smaller numbers for the calculations. This results in less memory requirements and less computational cost.

In order to speed up the central operation in Elliptic Curve Cryptosystems (ECC), namely scalar point multiplication, new methods have been proposed for the precomputation of points. This paper provides a complete list of updated operation costs using different point representations. Furthermore, focussing on given memory constraints, it compares the currently most promising algorithms and subjects them to a thorough performance analysis.

---

## Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
<b>2</b>	<b>Preliminaries</b>	<b>6</b>
2.1	Elliptic Curves . . . . .	6
2.2	Scalar Multiplication . . . . .	7
2.2.1	Double-And-Add (DA) Algorithm . . . . .	7
2.2.2	Average Hamming Density . . . . .	7
2.2.3	Coordinate Systems . . . . .	9
2.2.4	Overall Cost . . . . .	10
<b>3</b>	<b>Operation Costs</b>	<b>10</b>
3.1	Jacobian Doublings . . . . .	11
3.2	Jacobian Additions . . . . .	12
3.3	Affine Doublings converted to Jacobian . . . . .	13
3.4	Mixed Additions . . . . .	14
3.5	Projective Doublings . . . . .	15
3.6	Summary . . . . .	15
<b>4</b>	<b>Efficient composite operation <math>dP + Q</math></b>	<b>17</b>
4.1	Double-and-Add Algorithm . . . . .	18
4.2	Precomputation Scheme . . . . .	18
<b>5</b>	<b>Affine Precomputation with Sole Inversion</b>	<b>19</b>
<b>6</b>	<b>Performance Comparison</b>	<b>20</b>
6.1	Sole Inversion Schemes . . . . .	20
6.2	Inversion-free Scheme in $J^c$ . . . . .	24
<b>7</b>	<b>Conclusion</b>	<b>26</b>
	<b>Bibliography</b>	<b>27</b>
	<b>Appendix</b>	<b>29</b>

---

## List of Algorithms

1	Double-and-Add Algorithm . . . . .	7
2	$t(2\mathcal{P})$ . . . . .	30
3	$t(2\mathcal{P})$ if $a = -3$ . . . . .	31
4	$t(2\mathcal{J})$ . . . . .	32
5	$t(2\mathcal{J}^c)$ . . . . .	33
6	$t(2\mathcal{J}^m)$ . . . . .	34
7	$t(2\mathcal{J})$ if $a = -3$ . . . . .	35
8	$t(2\mathcal{J}^c)$ if $a = -3$ . . . . .	36
9	$t(2\mathcal{A} = \mathcal{J})$ . . . . .	37
10	$t(\mathcal{J}^c + \mathcal{J} = \mathcal{J}^m)$ . . . . .	38
11	$t(\mathcal{J} + \mathcal{A} = \mathcal{J}^m)$ . . . . .	39

## List of Tables

1	Computation requirements for doubling and addition in the general case . . . . .	16
2	Computation requirements changes if $a = -3$ . . . . .	16
3	Total cost for a 192-bit scalar multiplication . . . . .	22
4	Total cost for a 224-bit scalar multiplication . . . . .	23
5	Total cost for a 256-bit scalar multiplication . . . . .	23
6	I/M break even points for $n=192$ . . . . .	24
7	I/M break even points for $n=224$ . . . . .	25
8	I/M break even points for $n=256$ . . . . .	25

---

## List of Abbreviations

AHD	Average Hamming density
DLP	Discrete Logarithm Problem
ECC	Elliptic Curve Cryptography
ECADD	Elliptic Curve Point Addition
ECDBL	Elliptic Curve Point Doubling
ECDLP	Elliptic Curve Discrete Logarithm Problem
RSA	Cryptosystem proposed by Rivest, Shamir and Adleman
$w$ NAF	Width- $w$ Non Adjacent Form

## List of Symbols

$n$	Bit length of a scalar
$d$	Scalar, i.e. a positive integer
$d[i]$	The $i$ -th bit of the scalar $d$ , $i = 1, \dots, n$
$\mathcal{D}$	Digit set
$M$	Field multiplication
$S$	Field squaring
$I$	Field inversion
$\mathbb{F}_p$	Prime field
$E(\mathbb{F}_p)$	Additive group of points on an elliptic curve
$\mathcal{A}$	Affine coordinates
$\mathcal{J}$	Jacobian coordinates
$\mathcal{J}^m$	Modified Jacobian coordinates
$\mathcal{J}^c$	Chudnovsky Jacobian coordinates

---

## 1 Introduction

The most important goals of cryptographic schemes (so-called *cryptosystems*) are to achieve confidentiality, integrity, authenticity and non-repudiation of transmitted information [Buc04].

To ensure confidentiality, this information (*plaintext*) is first encrypted by an *encryption function*  $\mathcal{E}$ . After the encrypted information is transmitted to the recipient, the reverse process, called decryption, is performed by a *decryption function*  $\mathcal{D}$ . Besides the plaintext  $m$ , the encryption function requires the input of an *encryption key*  $e$ . It returns the encrypted message, the *ciphertext*  $c$ . The ciphertext and a *decryption key*  $d$  are the input for the decryption function which returns the original message, the *plaintext*. The respective formulas are given by

$$\mathcal{E}_e(m) = c \qquad \mathcal{D}_d(c) = m$$

Today's cryptosystems make use of so-called public-key systems which have the advantage that it is only necessary to keep the decryption key  $d$  secret, the "private key"; the encryption key  $e$  can be published and therefore anyone can encrypt data using this "public key". It is not possible to retrieve the private key from the public key with reasonable time effort since it is based on some complex mathematical problem, in our case solving the equation  $a^b = c$  for  $b$  when  $a$  and  $c$  are known; it involves the use of logarithms and is therefore called the Discrete Logarithm Problem (DLP).

Due to their tamper resistance and mobility, cryptosystems are often implemented on smart cards. However, smart cards have only the size of a credit card and their computational power and memory is very limited. Therefore, in comparison to standard public-key cryptosystems such as RSA [RSA78], Elliptic Curve Cryptography (ECC), independently introduced by Koblitz [Kob87] and Miller [Mil86] in the 1980's, has attracted increasing attention in recent years due to its shorter key length requirement. The same holds for ECDSA compared to its standard counterpart *Digital Signature Algorithm* (*DSA*) which is used to achieve the remaining three goals of cryptography: integrity, authenticity and non-repudiation.

Elliptic cryptosystems based on the ECDLP with a 160-bit key are considered to have the same security as both ElGamal [ElG85] and RSA cryptosystems with a 1024-bit key. This significantly shorter key length implies reduced power consumption, computing effort and memory requirement; factors that are critical in restricted environments like smart cards or mobile devices like cell phones or PDA's.

---

This thesis is organized as follows: Chapter 2 introduces the basic concept of elliptic curves and their application to cryptography. In Chapter 3, we update operation costs on elliptic curves using standard representations of the points on an elliptic curve. Chapter 4 and 5 summarize two ideas that both result in precomputation schemes. In Chapter 6 we analyze the performance of both schemes for a given number of memory registers. Finally, Chapter 7 states the conclusion.

## 2 Preliminaries

### 2.1 Elliptic Curves

**Definition 1.** Let  $\mathbb{F}_p$  denote a prime field, where  $p$  is a prime number. A prime field consists of the integers

$$\mathbb{F}_p = \{0, 1, \dots, p-1\}$$

and all arithmetic operations are computed modulo  $p$ . Those operations are field multiplications ( $M$ ), field squarings ( $S$ ) and field inversions ( $I$ ).

For a comparison of the costs of these three operations, see section 3.

**Definition 2.** An elliptic curve  $E$  over a prime field  $\mathbb{F}_p$  (denoted by  $E(\mathbb{F}_p)$ ) is defined by the equation

$$E : y^2 + a_1xy + a_2y = x^3 + a_3x^2 + a_4x + a_5 \quad (1)$$

where  $a_1, a_2, a_3, a_4, a_5 \in \mathbb{F}_p$  are such that, for each point  $(x, y)$  on  $E$ , the partial derivatives do not vanish simultaneously.

In practice, the equation can be simplified to the reduced Weierstrass equation (transformation see [HMOV04] p.78/79):

$$E : y^2 = x^3 + ax + b \quad (2)$$

where  $a, b \in \mathbb{F}_p$ ,  $\Delta = 4a^3 + 27b^2 \neq 0$  and  $\text{char}(\mathbb{F}_p) > 3$ .

The set of pairs  $(x, y)$  that solves (2) where  $x, y \in \mathbb{F}_p$  and the point at infinity  $O$  as neutral element form an abelian group  $(E(\mathbb{F}_p), +)$  which is used for the ECC computations. The elements of  $E(\mathbb{F}_p)$ , i.e. the points on the elliptic curve can be represented in different coordinate systems.

---

## 2.2 Scalar Multiplication

Scalar multiplication  $[d]P$ , where  $d$  is the secret key (scalar) and  $P$  is a point on the elliptic curve, is the central operation of elliptic cryptosystems. Traditionally, methods to efficiently compute a scalar multiplication use a binary expansion of  $d$ , which directly translates to computations using the simplest ECC operations, namely doubling and addition. Scalar multiplication occurs for example in the Diffie-Hellmann key exchange [DH76] and the ElGamal encryption scheme [ElG85].

### 2.2.1 Double-And-Add (DA) Algorithm

An efficient method to compute scalar multiplications is the *Double-and-Add (DA)* Algorithm.

---

**Algorithm 1** Double-and-Add Algorithm

---

**Require:** Point  $P \in E(\mathbb{F}_p)$ ,  $n$ -bit scalar  $d$ .

**Ensure:** Scalar multiplication  $[d]P$

- 1:  $X \leftarrow \mathcal{O}$
  - 2: **for**  $i = n - 1$  down to 0 **do**
  - 3:    $X \leftarrow \text{ECDBL}(X)$
  - 4:   **if**  $d[i] \neq 0$  **then**  $X \leftarrow \text{ECADD}(X, d[i]P)$
  - 5: **end for**
  - 6: **return**  $X$
- 

Looking at Algorithm 1, we see that the required number of ECADD operations depends on the number of non-zero elements of the representation used for the scalar. Hence we have two dimensions of improvement:

1. Reducing the number of non-zero elements of the scalar representation
2. Optimizing the operation costs for ECADD and ECDBL

### 2.2.2 Average Hamming Density

**Definition 3.** The vector  $(d[n-1], \dots, d[0])$  is called a  $\mathcal{D}$ -representation of the integer  $d$ , if

$$d = \sum_{i=0}^{n-1} d[i] \cdot 2^i$$

and  $d[i] \in \mathcal{D}, \forall i = 0, \dots, n - 1$ . Furthermore,  $\mathcal{D}$  is called the **digit set**.

Let  $\mathcal{X}$  be a class of  $\mathcal{D}$ -representations generated by a certain algorithm.

---

**Definition 4.** Let  $d = (d[n-1], \dots, d[0])$  be a  $\mathcal{D}$ -representation with bit length  $n$ . The Hamming weight of  $d$  is the number of non-zero digits in  $d$  and denoted by  $HW(d)$ . The Hamming density of  $d$  is given as  $HD(d) := HW(d)/n$ . The **Average Hamming Density** of a class of  $\mathcal{D}$ -representations  $\mathcal{X}$  is the expected Hamming density of a randomly chosen  $\mathcal{D}$ -representation in  $\mathcal{X}$  with bit length  $n \rightarrow \infty$  and denoted by  $AHD(\mathcal{X})$ .

**Example 5.** In the case of the binary representation, the digits 0 and 1 appear each with a probability of  $1/2$ . Therefore we see that

$$AHD(\text{binary}) = \frac{1}{2}$$

Now, it is possible to save ECADD operations by deploying the scalars in  $\mathcal{D}$ -representations with a low AHD. Instead of the well-known bitwise processing of the binary representation, **window methods** [Möl02, Möl04, SST04] process  $w$  digits of  $d$  at a time. As a trade-off, window methods have to increase the size of the digit set  $\mathcal{D} = \{0, 1\}$  and we have to precompute some points before the actual scalar multiplication. In this thesis, we consider computation from a **memory** point of view, where a certain number of registers  $r$  is given. Therefore our window size is dictated by memory limitations and depending on how much memory the respective precomputation scheme needs, we can precompute  $L$  points  $[3]P, [5]P, \dots, [2L+1]P$ . One can find different notations in literature for the digit set and precomputation points, we will use the notation based on the number  $L$  of points to precompute. The precomputation will be described in sections 4 and 5.

The  $\mathcal{D}$ -representation  $w$ NAF was independently proposed by Blake, Seroussi and Smart [BSS99] and Solinas [Sol00], although the basic idea goes back to Miyaji, Ono and Cohen [MOC97]. For an explicit algorithm to compute  $w$ NAFs and examples see [HMV04] p.99-100. A drawback is that only a small portion of the numbers are possible table sizes.

Therefore the concept has later been generalized allowing arbitrary table sizes to the so-called *Frac- $w$ NAF* [Möl02]. In [MS04], Muir and Stinson proved that the HW of a scalar given in its  $w$ NAF is **minimal** for any choice of  $w$ . This implies, that the AHD of the *Frac- $w$ NAF*

$$AHD_L = \left( \frac{L+1}{2^{\lfloor \log_2(L+1) \rfloor}} + \lfloor \log_2(L+1) \rfloor + 2 \right)^{-1}$$

is minimal [Möl04] amongst all  $\mathcal{D}$ -representations which use the digit set  $\mathcal{D}_L = \{0, \pm 1, \pm 3, \pm 5, \dots, \pm(2L+1)\}$ .

---

Note that  $[-3]P$  can be determined from  $[3]P$  by an “on-the-fly” point inversion [BSS99] and therefore only the positive points have to be precomputed.

Using this AHD, we see that performing Algorithm 1 costs

$$cs_{DA} = n \cdot \text{AHD}_L \cdot \text{ECADD} + n \cdot \text{ECDBL} \quad (3)$$

### 2.2.3 Coordinate Systems

The traditional, point-arithmetically given  $(x, y)$ -based point representation is called affine coordinate system ( $\mathcal{A}$ ). Since inversions are the most expensive field operation, projective coordinates have been effectively used to speed up operations and eliminating the inversion by introducing a third coordinate using the notion of equivalence classes ([HMV04] p.86). Given a prime field  $\mathbb{F}_p$ , there is an equivalence relation  $\equiv$  among non-zero triplets over  $\mathbb{F}_p$ , such that

$$\begin{aligned} (X_1, Y_1, Z_1) &\equiv (X_2, Y_2, Z_2) \\ \Leftrightarrow X_1 &= \lambda^c X_2, \quad Y_1 = \lambda^d Y_2, \quad Z_1 = \lambda Z_2 \end{aligned}$$

for some  $\lambda \in \mathbb{F}_p^*$ . It is important to note that any  $(X, Y, Z)$  in the equivalence class  $(X : Y : Z) = \{(\lambda^2 X, \lambda^3 Y, \lambda Z) : \lambda \in \mathbb{F}_p^*\}$  can be used as representative of a given point. If  $c = 1$  and  $d = 1$  we get standard projective coordinates ( $\mathcal{P}$ ); if  $c = 2$  and  $d = 3$  we get Jacobian coordinates ( $\mathcal{J}$ ) that have yielded efficient formulae. Furthermore its two derivatives Chudnovsky-Jacobian ( $\mathcal{J}^c$ ) and modified Jacobian ( $\mathcal{J}^m$ ) emerged, the latter of which was proposed in order to achieve the fastest possible doublings.

For a detailed overview of the coordinate systems and the explicit operation formulas see [HMV04] and [CMO98]. The rallye to find new coordinate systems that have even faster doublings is ongoing; we will focus on the analysis of these classical coordinate systems:

system	points	correspondence
affine $\mathcal{A}$	$(x, y)$	
Projective $\mathcal{P}$	$(X, Y, Z)$	$(X/Z, Y/Z)$
Jacobian $\mathcal{J}$	$(X, Y, Z)$	$(X/Z^2, Y/Z^3)$
Chudnovsky Jacobian $\mathcal{J}^c$	$(X, Y, Z, Z^2, Z^3)$	$(X/Z^2, Y/Z^3)$
modified Jacobian $\mathcal{J}^m$	$(X, Y, Z, aZ^4)$	$(X/Z^2, Y/Z^3)$

For each of these systems and for combinations of them, the speed of additions and doublings is different and hence the question which operation is performed in which coordinate system is an important factor for scalar multiplication, the main operation on elliptic curves.

---

### 2.2.4 Overall Cost

To perform the scalar multiplication, the process is split into three operations and every operation is performed in the optimal coordinate system [CMO98]:

1. ECDBL: If  $d_i = 0$  we have a doubling followed by a doubling (so-called "main doubling") and we use  $J^m$  coordinates, i.e. we perform  $t(2J^m)$
2. ECDBL: If  $d_i \neq 0$  we have a doubling followed by an addition and hence we store the result of the doubling in  $\mathcal{J}$  coordinates, i.e. we perform  $t(2J^m = \mathcal{J})$
3. ECADD: For the addition of precomputed points  $[u_i]P$  we use either  $\mathcal{A}$  or  $\mathcal{J}^c$  coordinates, i.e. we perform either  $t(\mathcal{A} + \mathcal{J} = J^m)$  or  $t(\mathcal{J}^c + \mathcal{J} = J^m)$

The total cost of a scalar multiplication in dependency of the precomputation scheme is given by

$$\text{Cost} = cs_{DA} + cp_{scheme} \quad (4)$$

Depending on the coordinate system of the precomputed points ( $\mathcal{A}$  or  $\mathcal{J}^c$ ), we not only obtain different precomputation costs  $cp$  but also need a DA Algorithm with different costs  $cs$  to add the precomputed points.

## 3 Operation Costs

Operation costs are usually measured in field multiplications. It is widely accepted that one field squaring is less computationally expensive than one field multiplication. In this thesis, we use the commonly used ratio of squarings to multiplications of  $S/M = 0.8$ ; some environments result in even lower ratios up to  $S/M = 0.6$  [BHLM01, GAST05, LH00, GG03, Ava04].

Inversions are by far the most costly operation and should therefore be avoided although their relative cost depends on the characteristics of the particular implementation. For instance, benchmarks presented by [LH00] and [IEEE, BHLM01] show  $I/M$  ratios of 30-40 and 50-100. The ratio is particularly large on smart cards equipped with cryptographic coprocessor, which is usually the case [Infineon, Renesas]. Seysen [Sey05] states that on such smart cards an  $I/M$  ratio of  $I > 100M$  is realistic since the inversion is best computed using Fermat's little theorem [CF05, ELM03, JP03]. This approach requires about  $\log_2 p$  field multiplications, where  $p$  is the prime that defines the field.

---

For efficiency purposes, most curves recommended by public-key standards [IEEE] use  $a = -3$ , which does not impose significant restrictions to the cryptosystem [BJ03] and enhances some ECC operations (see below).

Longa and Miri [LM07] proposed to use Pythagoras Theorem as algebraic transformation to rewrite an even multiplication  $2XY$  as sum of three squares  $(X + Y)^2 - X^2 - Y^2$  if two of them have to be calculated in the further process of the operation anyway. Therefore we see that for each replaceable multiplication we trade *one* multiplication for *one* squaring; this difference saves up to 4.5% (for  $t(2\mathcal{J})$ ) of the operation costs. We apply this approach not only to  $\mathcal{J}$  coordinates (as done in [LM07]), but to a wider range of operations.

### 3.1 Jacobian Doublings

Let  $P$  be a point on the elliptic curve  $E$  given in  $\mathcal{J}$ ,  $\mathcal{J}^c$  or  $\mathcal{J}^m$  (see section 2.2.3). Then  $2P = (X_3, Y_3, Z_3)$  can be computed using the traditional formulas [CMO98]:

$$\begin{aligned} X_3 &= \alpha^2 - 2\beta \\ Y_3 &= \alpha(\beta - X_3) - 8Y_1^4 \\ Z_3 &= 2Y_1Z_1 \\ (aZ_3^4 &= 16Y_1^4 \cdot aZ_1^4) \end{aligned} \tag{5}$$

where  $\alpha = 3X_1^2 + aZ_1^4$  and  $\beta = 4X_1Y_1^2$ .

#### General case:

Applying the method of [LM07], we can replace

$$\begin{aligned} Z_3 &= 2Y_1Z_1 &= (Y_1 + Z_1)^2 - Y_1^2 - Z_1^2 \\ \beta &= 4X_1Y_1^2 &= 2((X_1 + Y_1^2)^2 - X_1^2 - Y_1^4) \end{aligned} \tag{6}$$

The resulting cost for  $t(2\mathcal{J})$  is 2M+8S instead of the normal 4M+6S; furthermore 6 temporary registers are required. For explicit pseudocode see Algorithm 4.

The resulting cost for  $t(2\mathcal{J}^c)$  is 3M+8S instead of the normal 5M+6S; furthermore 6 temporary registers are required. For explicit pseudocode see Algorithm 5.

The resulting cost for  $t(2\mathcal{J}^m)$  is 3M+5S instead of the normal 4M+4S; furthermore 5 temporary registers are required. For explicit pseudocode see Algorithm 6. If the result is saved in  $\mathcal{J}$  coordinates, i.e. for  $t(2\mathcal{J}^m = \mathcal{J})$ ,

---

we simply omit the last line of the pseudocode. If the result is saved in  $\mathcal{J}^c$  coordinates, the last line is replaced by the calculation of  $Z_3^2$  and  $Z_3^3$ .

**Special case  $a = -3$ :**

For Jacobian and Chudnovsky doublings, we can accelerate the above computation of  $\alpha$  by trading one multiplication and a squaring for three squarings  $\alpha = 3X_1^2 - 3Z_1^4 = 3(X_1 - Z_1^2)(X_1 + Z_1^2)$ . Afterwards, since we no longer determine  $X_1^2$ , computing  $\beta$  as above is no improvement as one multiplication is replaced by two squarings; we only change the calculation of  $Z_3$ .

The resulting cost for  $t(2\mathcal{J})$  is 3M+5S instead of the normal 4M+6S; furthermore 6 temporary registers are required. For explicit pseudocode see Algorithm 7.

The resulting cost for  $t(2\mathcal{J}^c)$  is 4M+5S instead of the normal 5M+6S; furthermore 6 temporary registers are required. For explicit pseudocode see Algorithm 8.

For the calculation of  $t(2\mathcal{J}^m)$  there are no changes.

### 3.2 Jacobian Additions

Let  $P$  and  $Q$  be points on the elliptic curve  $E$  given in  $\mathcal{J}$ ,  $\mathcal{J}^c$  or  $\mathcal{J}^m$  coordinates (see section 2.2.3). Then  $P + Q = (X_4, Y_4, Z_4)$  can be computed using the traditional formulas [CMO98]:

$$\begin{aligned} X_4 &= \alpha^2 - \beta^3 - 2Z_2^2 X_1 \beta^2 \\ Y_4 &= \alpha(Z_2^2 X_1 \beta^2 - X_4) - Z_2^3 Y_1 \beta^3 \\ Z_4 &= Z_1 Z_2 \beta \end{aligned} \tag{7}$$

where  $\alpha = Z_1^3 Y_2 - Z_2^3 Y_1$  and  $\beta = Z_1^2 X_2 - Z_2^2 X_1$ .

To be able to apply the method of [LM07], we first have to introduce a factor 2 in  $Z_4$  by finding an equivalent point to  $P + Q$  on the curve. Using  $\lambda = 2$  (see section 2.2.3) we get  $P + Q = (X_5, Y_5, Z_5) = (2^2 X_4, 2^3 Y_4, 2Z_4) \equiv (X_4, Y_4, Z_4)$ :

$$\begin{aligned} X_5 &= \alpha^2 - 2(2\beta^3 - 4Z_2^2 X_1 \beta^2) \\ Y_5 &= \alpha(4Z_2^2 X_1 \beta^2 - X_5) - 8Z_2^3 Y_1 \beta^3 \\ Z_5 &= 2Z_1 Z_2 \beta \end{aligned} \tag{8}$$

where  $\alpha = 2(Z_1^3 Y_2 - Z_2^3 Y_1)$  and  $\beta = Z_1^2 X_2 - Z_2^2 X_1$ .

---

Now, we can replace

$$Z_5 = 2Z_1Z_2\beta = ((Z_1 + Z_2)^2 - Z_1^2 - Z_2^2)\beta \quad (9)$$

We do not list all 8 pseudocodes since they do not differ much, it is more a question of juggling with  $Z$ -multiples: if points are given in  $\mathcal{J}^c$  we save operation costs at the beginning but also need some more to determine the 4<sup>th</sup> resp. 5<sup>th</sup> coordinate. All pseudocodes can be derived from Algorithm 10, which is needed for section 6.2.

**General case:**

The resulting cost for  $t(\mathcal{J}^c + \mathcal{J} = \mathcal{J}^m)$  is 11M+6S instead of the normal 12M+5S; furthermore 6 temporary registers are required.

**Special case  $a = -3$ :**

For all operations with target coordinate system  $\mathcal{J}^m$ , the last multiplication with  $a$  becomes trivial and hence the total cost reduces by one multiplication. There is no effect on the other operations.

### 3.3 Affine Doublings converted to Jacobian

Let  $P = (x_1, y_1)$  be a point on the elliptic curve  $E$ . Then  $2P = (x_3, y_3)$  can be computed using the traditional formulas [CMO98] with slightly different notation:

$$\begin{aligned} x_3 &= w^2 - 2x_1 \\ y_3 &= w(x_1 - x_3) - y_1 \end{aligned} \quad (10)$$

where  $w = (3x_1^2 + a)/(2y_1)$ .

This point would be equivalent to the Jacobian point  $(x_3, y_3, 1)$  but still contains inverted elements. Transforming this to  $\mathcal{J}$ ,  $\mathcal{J}^c$  or  $\mathcal{J}^m$  requires clearing denominators which can be done by using the equivalent point  $(\lambda^2x_3, \lambda^3y_3, \lambda)$  where  $\lambda = 2y_1$  (see section 2.2.3). In the first two coordinates we now need to calculate  $2x_1y_1^2$  at some point. We can trade this multiplication for one squaring using the method of [LM07]:

$$2x_1y_1^2 = (x_1 + y_1^2)^2 - x_1^2 - y_1^4 \quad (11)$$

The pseudocodes can be easily derived from [LM08] Appendix B, Algorithm 1 (p.20). For reasons of completeness, we repeat it here (see Algorithm 9). All methods need 6 temporary registers.

---

**General case:**

The resulting cost for  $t(2\mathcal{A} = \mathcal{J})$  is 1M+5S instead of the normal 2M+4S.

The resulting cost for  $t(2\mathcal{A} = \mathcal{J}^c)$  is 2M+5S instead of the normal 3M+4S. We get  $Z_3^2 = \lambda^2 = 4y_1^2$  without significant extra calculation, but  $Z_3^3 = \lambda^2 \cdot \lambda = 8y_1^3$  requires an extra multiplication.

The resulting cost for  $t(2\mathcal{A} = \mathcal{J}^m)$  is 2M+5S instead of the normal 3M+5S since  $aZ_3^4 = a \cdot 16Y_1^4$  requires a multiplication with  $a$ .

**Special case  $a = -3$ :**

For  $t(2\mathcal{A} = \mathcal{J}^m)$ , the last multiplication with  $a$  becomes trivial and hence the total cost is 1M+5S. There is no effect on the other operations.

### 3.4 Mixed Additions

Let  $P = (X_1, Y_1, Z_1)$  be a point on the elliptic curve  $E$  given in  $\mathcal{J}$  (or with additional coordinates in  $\mathcal{J}^c$  or  $\mathcal{J}^m$ ) and  $Q = (x_2, y_2)$  in  $\mathcal{A}$  coordinates (see section 2.2.3). Then  $P + Q$  can be computed in  $\mathcal{J}$ ,  $\mathcal{J}^c$  or  $\mathcal{J}^m$  using the traditional formulas from the previous section 3.2 (Jacobian addition), where all  $Z_2$  can be replaced by 1, i.e. omitted. To apply the method of [LM07], we again have to introduce a factor 2 in  $Z_4$  by finding an equivalent point to  $P+Q$  on the curve. Using  $\lambda = 2$  (see section 2.2.3) we get  $P + Q = (X_5, Y_5, Z_5) = (2^2X_4, 2^3Y_4, 2Z_4) \equiv (X_4, Y_4, Z_4)$ :

$$\begin{aligned} X_5 &= \alpha^2 - 2(2\beta^3 - 4X_1\beta^2) \\ Y_5 &= \alpha(4X_1\beta^2 - X_5) - 8Y_1\beta^3 \\ Z_5 &= 2Z_1\beta \end{aligned} \tag{12}$$

where  $\alpha = 2(Z_1^3y_2 - Y_1)$  and  $\beta = Z_1^2x_2 - X_1$ .

Now, we can replace

$$Z_5 = 2Z_1\beta = (Z_1 + \beta)^2 - Z_1^2 - \beta^2 \tag{13}$$

We do not list all 6 pseudocodes since they do not differ much, it is more a question of juggling with  $Z$ -multiples: if points are given in  $\mathcal{J}^c$  we save operation costs at the beginning but also need some more to determine the 4<sup>th</sup> resp. 5<sup>th</sup> coordinate. All pseudocodes can be derived from Algorithm 11.

**General case:**

The resulting cost for  $t(\mathcal{J} + \mathcal{A} = \mathcal{J}^m)$  is 8M+6S instead of the normal 9M+5S; furthermore 6 temporary registers are required.

---

**Special case  $a = -3$ :**

For all operations with target coordinate system  $\mathcal{J}^m$ , the last multiplication with  $a$  becomes trivial and hence the total cost reduces by one multiplication. There is no effect on the other operations.

**3.5 Projective Doublings**

Let  $P = (X_1, Y_1, Z_1)$  be a point on the elliptic curve  $E$ . Then  $2P = (X_3, Y_3, Z_3)$  can be computed using the traditional formulas stated in [CMO98] with slightly different notation:

$$\begin{aligned} X_3 &= hs \\ Y_3 &= w(B - h) - 2R^2 \\ Z_3 &= s^3 \end{aligned} \tag{14}$$

where  $w = 3X_1^2 + aZ_1^2$ ,  $s = 2Y_1Z_1$ ,  $R = Y_1s$ ,  $B = 2X_1R$  and  $h = w^2 - 2B$ .

**General case:**

Bernstein and Lange [BL07] applied the method of [LM07] to replace

$$B = 2X_1R = (X_1 + R)^2 - X_1^2 - R^2 \tag{15}$$

The resulting cost for  $t(2\mathcal{P})$  is 6M+6S instead of the normal 8M+5S; furthermore 6 temporary registers are required. For explicit pseudocode see Algorithm 2.

**Special case  $a = -3$** 

We can accelerate the above computation of  $w$  by trading one multiplication for two squarings  $w = 3X_1^2 - 3Z_1^2 = 3(X_1 - Z_1)(X_1 + Z_1)$ . Afterwards, since we no longer determine  $X_1^2$ , computing  $B$  as above is no improvement as one multiplication would be replaced by two squarings.

The resulting cost for  $t(2\mathcal{P})$  if  $a = -3$  is 7M+3S instead of the normal 8M+5S; furthermore 6 temporary registers are required. For explicit pseudocode see Algorithm 3.

**3.6 Summary**

By consequently applying the method of [LM07] to all most useful operations, we can summarize all new operation costs in table 1, an updated version of [CMO98]. Furthermore, we provide explicit pseudocodes (see Appendix A) and review all costs in case  $a = -3$  (table 2). Affine additions converted to Jacobian such as  $t(\mathcal{A} + \mathcal{A} = \mathcal{J})$  as well as Projective additions  $t(\mathcal{P} + \mathcal{P})$  cannot be improved using our approach.

Table 1: Computation requirements for doubling and addition in the general case

<i>doubling</i>		<i>addition</i>	
<i>operation</i>	<i>computation</i>	<i>operation</i>	<i>computation</i>
$t(2\mathcal{P})$	$6M + 6S$	$t(\mathcal{J}^m + \mathcal{J}^m)$	$12M + 7S$
$t(2\mathcal{J}^c)$	$3M + 8S$	$t(\mathcal{J}^m + \mathcal{J}^c = \mathcal{J}^m)$	$11M + 6S$
$t(2\mathcal{J})$	$2M + 8S$	$t(\mathcal{J}^c + \mathcal{J} = \mathcal{J}^m)$	$11M + 6S$
$t(2\mathcal{J}^m = \mathcal{J}^c)$	$3M + 6S$	$t(\mathcal{J} + \mathcal{J})$	$11M + 5S$
$t(2\mathcal{J}^m)$	$3M + 5S$	$t(\mathcal{P} + \mathcal{P})$	$12M + 2S$
$t(2\mathcal{A} = \mathcal{J}^c)$	$2M + 5S$	$t(\mathcal{J}^c + \mathcal{J}^c = \mathcal{J}^m)$	$10M + 5S$
$t(2\mathcal{J}^m = \mathcal{J})$	$2M + 5S$	$t(\mathcal{J}^c + \mathcal{J}^c)$	$10M + 4S$
$t(2\mathcal{A} = \mathcal{J}^m)$	$2M + 5S$	$t(\mathcal{J}^c + \mathcal{J} = \mathcal{J})$	$10M + 4S$
$t(2\mathcal{A} = \mathcal{J})$	$1M + 5S$	$t(\mathcal{J}^c + \mathcal{J}^c = \mathcal{J})$	$9M + 3S$
–	–	$t(\mathcal{J} + \mathcal{A} = \mathcal{J}^m)$	$8M + 6S$
–	–	$t(\mathcal{J}^m + \mathcal{A} = \mathcal{J}^m)$	$8M + 5S$
–	–	$t(\mathcal{J}^c + \mathcal{A} = \mathcal{J}^m)$	$7M + 5S$
–	–	$t(\mathcal{J}^c + \mathcal{A} = \mathcal{J}^c)$	$7M + 4S$
–	–	$t(\mathcal{J} + \mathcal{A} = \mathcal{J})$	$7M + 4S$
–	–	$t(\mathcal{J}^m + \mathcal{A} = \mathcal{J})$	$7M + 4S$
–	–	$t(\mathcal{A} + \mathcal{A} = \mathcal{J}^m)$	$5M + 4S$
–	–	$t(\mathcal{A} + \mathcal{A} = \mathcal{J}^c)$	$5M + 3S$
$t(2\mathcal{A})$	$2M + 2S + I$	$t(\mathcal{A} + \mathcal{A})$	$2M + S + I$

As mentioned above, the case  $a = -3$  yields some improvement. The only effected operations from the table above are (see Appendix A for details):

Table 2: Computation requirements changes if  $a = -3$

<i>doubling</i>		<i>addition</i>	
<i>operation</i>	<i>computation</i>	<i>operation</i>	<i>computation</i>
$t(2\mathcal{P})$	$7M + 3S$	$t(\mathcal{J}^m + \mathcal{J}^m)$	$11M + 7S$
$t(2\mathcal{J}^c)$	$4M + 5S$	$t(\mathcal{J}^m + \mathcal{J}^c = \mathcal{J}^m)$	$10M + 6S$
$t(2\mathcal{J})$	$3M + 5S$	$t(\mathcal{J}^c + \mathcal{J} = \mathcal{J}^m)$	$10M + 6S$
$t(2\mathcal{A} = \mathcal{J}^c)$	$2M + 5S$	$t(\mathcal{J}^c + \mathcal{J}^c = \mathcal{J}^m)$	$11M + 4S$
$t(2\mathcal{A} = \mathcal{J}^m)$	$2M + 5S$	$t(\mathcal{J} + \mathcal{A} = \mathcal{J}^m)$	$7M + 6S$
$t(2\mathcal{A} = \mathcal{J})$	$1M + 5S$	$t(\mathcal{J}^m + \mathcal{A} = \mathcal{J}^m)$	$7M + 5S$
–	–	$t(\mathcal{J}^c + \mathcal{A} = \mathcal{J}^m)$	$6M + 5S$

---

## 4 Efficient composite operation $dP + Q$

Longa and Miri [LM08] proposed to compute the in many applications recurring composite operation  $[d]P + Q$  as

$$[d]P + Q = P + \dots + P + (P + Q)$$

based on Meloni's idea [Mel06] of a cheaper "addition of points in Jacobian coordinates having identical z-coordinates", from now on called "special addition". The scheme has the following outline:

1. Perform the first addition  $(P+Q) = (X_1, Y_1, Z_1) + (X_2, Y_2) = (X_3, Y_3, Z_3)$  as Affine-Jacobian mixed addition according to the formulas above with cost  $t(\mathcal{J} + \mathcal{A}) = 7M + 4S$ . Then

$$\begin{aligned} X_3 &= 4(Z_1^3 Y_2 - Y_1)^2 - 4(Z_1^2 X_2 - X_1)^3 - 8X_1(Z_1^2 X_2 - X_1)^2 \\ Y_3 &= 2(Z_1^3 Y_2 - Y_1)(4X_1(Z_1^2 X_2 - X_1)^2 - X_3) - 8Y_1(Z_1^2 X_2 - X_1)^3 \\ Z_3 &= 2Z_1(Z_1^2 X_2 - X_1) \end{aligned} \tag{16}$$

2. By fixing  $\lambda_1 = 2(Z_1^2 X_2 - X_1)$  (see section 2.2.3) we can assume the new representation  $P^1 \equiv P$  in the same equivalence class as  $P$  and having the same Z-coordinate as  $(P + Q)$ .

$$P^1 := (X_1^{(1)}, Y_1^{(1)}, Z_1^{(1)}) = (\lambda_1^2 X_1, \lambda_1^3 Y_1, \lambda_1 Z_1) \equiv (X_1, Y_1, Z_1) \tag{17}$$

This can be done using previously done calculation, so there is no extra cost involved.

3. Use this point to perform the next Jacobian addition  $P^1 + (P + Q)$  using a special addition with cost  $t_s(\mathcal{J} + \mathcal{J}) = 5M + 2S$ .
4. Iterate Steps 2 and 3 until the addition is fully performed (i.e.  $P$  can be generically rewritten as some  $P^j$  with the same z-coordinate as  $[j]P + Q$  and then be added using a special addition)

We omit the technical formulas of the scheme (for details see [LM08] p.5-7) and look at how this methodology was applied to accelerate scalar multiplication:

---

## 4.1 Double-and-Add Algorithm

If we choose  $d = 2$  we obtain  $2P + Q$ , the main operation of the DA Algorithm. The authors follow the above scheme: perform the first addition  $(P + Q)$  as Affine-Jacobian mixed addition with cost  $t(\mathcal{J} + \mathcal{A} = \mathcal{J}) = 7M + 4S$  and the second as special addition  $P^1 + (P + Q)$  with cost  $t_s(\mathcal{J} + \mathcal{J}) = 5M + 2S$ . By merging the two operations, it is possible to trade one multiplication for one squaring and we obtain a cost of  $11M + 7S$ .

**Theorem 6.** *For precomputations in  $\mathcal{A}$  we use  $t(2\mathcal{J}^m) = 3M + 5S$  and the above composite operation. The total cost of a scalar multiplication is then given by*

$$\text{Cost}_{\mathcal{A}} = [n \cdot \text{AHD}(11M + 7S) + n(1 - \text{AHD})(3M + 5S)] + cp_{\mathcal{A}} \quad (18)$$

## 4.2 Precomputation Scheme

If we choose  $Q = P$  and  $d = 2j$  for arbitrary  $j > 0$ , we obtain a precomputation scheme for  $[u_i]P = dP + P = 2P + \dots + 2P + P$ . The authors propose **two sole inversion precomputation schemes** using a "Modified Montgomery's method" [LM08] p.24 to simultaneously convert the precomputed points back to  $\mathcal{A}$  and reduce the number of inversion to one.

**Theorem 7.** *The resulting cost of the two LM precomputation schemes to compute the  $L$  points  $[3]P, [5]P, \dots, [2L + 1]P$  is given by*

$$cp_{[1]} = (9L)M + (3L + 5)S + I \quad (19)$$

$$cp_{[2]} = (9L)M + (2L + 6)S + I \quad (20)$$

and they need  $3L + 3$  or  $4L + 1$  registers respectively.

*Proof.* see [LM08] □

---

## 5 Affine Precomputation with Sole Inversion

To enable an efficient precomputation for affine coordinates in environments where inversion is expensive, all inversions have to be performed using Montgomery's method [HMOV04]. This can only be done if all values that have to be inverted are known. Dahmen et al. [DOS07] proposed a recursive strategy to compute all these values using only known variables  $(a, b, P)$ .

The main idea of the scheme can be summarized in four steps:

**Step 1.** In the formulas to compute points in affine coordinates

$$\begin{aligned}
[2]P &= (x_2, y_2) : \\
\lambda_1 &= \frac{(3x_1^2+a)}{(2y_1)} & x_2 &= \lambda_1^2 - 2x_1 \\
& & y_2 &= \lambda_1(x_1 - x_2) - y_1 \\
[3]P &= (x_3, y_3) : \\
\lambda_2 &= \frac{(y_2-y_1)}{(x_2-x_1)} & x_3 &= \lambda_2^2 - x_2 - x_1 \\
& & y_3 &= \lambda_2(x_2 - x_3) - y_2 \\
[2i-1]P &= (x_{i+1}, y_{i+1}) : \\
\lambda_i &= \frac{(y_i-y_2)}{(x_i-x_2)} & x_{i+1} &= \lambda_i^2 - x_2 - x_i \\
& & y_{i+1} &= \lambda_i(x_2 - x_{i+1}) - y_2
\end{aligned} \tag{21}$$

call the respective denominators  $\delta_i$  and compute values  $d_i$ , such that  $d_i = d_1^2 \cdot \dots \cdot d_{i-1} \cdot \delta_i$  holds for  $i = 1, \dots, k$ . Use a recursive strategy which successively substitutes the formulas for  $x_i, y_i$  in the formulas for  $x_{i+1}, y_{i+1}$ .

**Step 2.** Determine the inverses of  $d_i$  using Montgomery's method [HMOV04].

**Step 3.** Recover from the previous step the inverses of the denominators  $\delta_i$ .

**Step 4.** Use the  $\delta_i$  to compute the points  $[3]P, [5]P, \dots, [2L+1]P$  according to scheme 21.

**Theorem 8.** *The resulting cost of the DOS precomputation scheme to compute the  $L$  points  $[3]P, [5]P, \dots, [2L+1]P$  is given by*

$$\text{cp}_{\mathcal{A}} = (10L - 2)M + (4L + 4)S \tag{22}$$

and it requires  $2L+4$  registers.

*Proof.* see [DOS07] □

Having done the precomputation, please note that we can also use the DA Algorithm mentioned in section 4.1 to reduce the cost of a complete scalar multiplication.

---

## 6 Performance Comparison

The two main aspects of a restricted environment like smart cards are a limited number of memory registers and a characteristically high  $I/M$  ratio. We differentiate between

1. Straightforward precomputation in  $\mathcal{A}$  using multiple inversions
2. Sole Inversion Schemes resulting in precomputed points in  $\mathcal{A}$
3. Inversion-free precomputation in  $\mathcal{J}^c$

First, we consider  $I/M$  break even points if the precomputation is done in  $\mathcal{A}$  using multiple inversions. The analysis provided in [DOS07] p.8 remains valid but since we use even faster methods, we obtain  $I/M$  break even points  $<25$ , irrelevant in our environments with  $I/M$  ratios of 100 or higher (see introduction of section 3).

Second, we provide a detailed analysis of the precomputation schemes presented in sections 4.2 and 5. We compare them to other sole inversion schemes and show their superiority. Since all of these methods use exactly one inversion, the  $I/M$  ratio is not of interest in this step.

Third, we determine the  $I/M$  break even points for the inversion-free precomputation in  $\mathcal{J}^c$ . For reasons of comparability, we then perform the whole scalar multiplication since not only the including precomputation in the respective coordinate system but also the DA Algorithm used is different.

### 6.1 Sole Inversion Schemes

Straightforward sole inversion schemes in environments where inversions are relatively expensive perform the whole precomputation in  $\mathcal{J}$ ,  $\mathcal{J}^c$  or  $\mathcal{J}^m$  and use Montgomery's method [HMOV04] to reduce the number of inversions to one. Let  $\mathcal{C}$  be one of the above coordinate systems. Then we can improve the technique presented in [LM08] by using the following scheme:

1. Perform the necessary doubling  $2P$  as  $t(2\mathcal{A} = \mathcal{C})$
2. Compute  $3P$  as mixed addition  $P+2P$  as  $t(\mathcal{A} + \mathcal{C} = \mathcal{C})$
3. Compute all further points  $5P, 7P, \dots$  as addition  $t(\mathcal{C} + \mathcal{C})$
4. Use the "Modified Montgomery's method" [LM08] p.24 to simultaneously convert to  $\mathcal{A}$  and reduce the number of inversion to one

---

The respective updated costs are (calculation see Appendix B):

$$\text{cp}_{\mathcal{J} \rightarrow \mathcal{A}} = (15L - 4)M + (6L + 4)S + I \quad (23)$$

$$\text{cp}_{\mathcal{J}^c \rightarrow \mathcal{A}} = (14L - 2)M + (5L + 5)S + I \quad (24)$$

$$\text{cp}_{\mathcal{J}^m \rightarrow \mathcal{A}} = (16L - 2)M + (8L + 3)S + I \quad (25)$$

where the  $\mathcal{J}$  method needs  $3L+6$ , the  $\mathcal{J}^c$  method  $5L+6$  and the  $\mathcal{J}^m$  method  $4L+6$  registers.

We see that all of these methods are *inferior* and need *more memory* than the two sole inversion schemes presented in sections 4.2 and 5, independent of the DA Algorithm running time and the  $I/M$  ratio.

Let us assume now a certain number of available registers  $r$ . We look at a performance comparison of the sole inversion schemes presented in sections 4.2 and 5. First, we determine the respective maximum number of storable precomputation points  $L$ . Then we calculate the respective cost  $c$  in field multiplications for a complete scalar multiplication with an  $n$ -bit scalar. This is done for  $n = 192$  (table 3),  $n = 224$  (table 4) and  $n = 256$  (table 5) where the latter two omit the values for  $L$  that remain identical. Furthermore, all methods need exactly *one* field inversion, which is omitted in the tables. Note that for all three cases, values  $L > 7$  are not efficient since the precomputation is more costly than the resulting savings during scalar multiplication; in this case, we simply use  $L = 7$ . The MATLAB codes can be found in Appendix C.

Table 3: Total cost for a 192-bit scalar multiplication

$r$	6,7	8	9	10-11	12	13
$L$ ( $DOS07$ )	1	2	2	3	4	4
$L$ ( $LM08_1$ )	1	1	2	2	3	3
$L$ ( $LM08_2$ )	1	1	2	2	2	3
$c$ ( $DOS07$ )	<b>1819,2</b>	<b>1781,2</b>	1781,2	<b>1753,4</b>	<b>1749,1</b>	<b>1749,1</b>
$c$ ( $LM08_1$ )	1820,2	1820,2	1780,4	1780,4	1750,8	1750,8
$c$ ( $LM08_2$ )	1820,2	1820,2	<b>1779,6</b>	1779,6	1779,6	1749,2

  

$r$	14	15	16	17	18-20	21-23
$L$ ( $DOS07$ )	5	5	6	6	7	7
$L$ ( $LM08_1$ )	3	4	4	4	5	6
$L$ ( $LM08_2$ )	3	3	3	4	4	5
$c$ ( $DOS07$ )	<b>1746,3</b>	1746,3	1745	1745	1744,8	1744,8
$c$ ( $LM08_1$ )	1750,8	<b>1744,7</b>	<b>1744,7</b>	1744,7	<b>1740,1</b>	1737
$c$ ( $LM08_2$ )	1749,2	1749,2	1749,2	<b>1742,3</b>	1742,3	<b>1736,9</b>

  

$r$	24	25-28	$\geq 33$
$L$ ( $DOS07$ )	7	7	7
$L$ ( $LM08_1$ )	7	7	7
$L$ ( $LM08_2$ )	5	6	7
$c$ ( $DOS07$ )	1744,8	1744,8	1744,8
$c$ ( $LM08_1$ )	<b>1735</b>	1735	1735
$c$ ( $LM08_2$ )	1736,9	<b>1733</b>	<b>1730,2</b>

For example, if we have a 192-bit scalar and 16 available registers, using the DOS precomputation scheme, we need 1745M+I to perform a whole scalar multiplication. Using LM scheme 1 or 2, we need 1744,7M+I respectively 1749,2M+I. Hence, for  $r = 16$  it is best to use LM precomputation scheme 1. We see that (except for  $r = 9$ ) if we have 14 or less registers available, it is best to use the DOS scheme and we can save up to 2,1% (for  $r = 8$ ); otherwise we use one of the LM schemes. If both schemes can operate with optimal  $L = 7$ , the LM scheme 2 outperforms the DOS scheme by 0.8%. Looking at higher values for  $n$  we recognize the same patterns but the DOS method now outperforms the LM methods until 20 registers.

Table 4: Total cost for a 224-bit scalar multiplication

$r$	6,7	8	9	10-11	12	13
$c(DOS07)$	<b>2120</b>	<b>2073,5</b>	2073,5	<b>2038,9</b>	<b>2031,6</b>	<b>2031,6</b>
$c(LM08_1)$	2121	2121	2072,7	2072,7	2036,3	2036,3
$c(LM08_2)$	2121	2121	<b>2071,9</b>	2071,9	2071,9	2034,7
$r$	14	15	16	17	18-20	21-23
$c(DOS07)$	<b>2026,2</b>	<b>2026,2</b>	<b>2022,4</b>	<b>2022,4</b>	<b>2020</b>	2020
$c(LM08_1)$	2036,3	2027,2	2027,2	2027,2	2020	<b>2014,4</b>
$c(LM08_2)$	2034,7	2034,7	2034,7	2024,8	2024,8	2016,8
$r$	24	25-28	$\geq 33$			
$c(DOS07)$	2020	2020	2020			
$c(LM08_1)$	<b>2010,2</b>	<b>2010,2</b>	2010,2			
$c(LM08_2)$	2016,8	2010,4	<b>2005,4</b>			

Table 5: Total cost for a 256-bit scalar multiplication

$r$	6,7	8	9	10-11	12	13
$c(DOS07)$	<b>2420,8</b>	<b>2365,7</b>	2365,7	<b>2324,3</b>	<b>2314,1</b>	<b>2314,1</b>
$c(LM08_1)$	2421,8	2421,8	2364,9	2364,9	2321,7	2321,7
$c(LM08_2)$	2421,8	2421,8	<b>2364,1</b>	2364,1	2364,1	2320,1
$r$	14	15	16	17	18-20	21-23
$c(DOS07)$	<b>2306</b>	<b>2306</b>	<b>2299,8</b>	<b>2299,8</b>	<b>2295,2</b>	2295,2
$c(LM08_1)$	2321,7	2309,7	2309,7	2309,7	2299,8	<b>2291,8</b>
$c(LM08_2)$	2320,1	2320,1	2320,1	2307,3	2307,3	2296,6
$r$	24	25-28	$\geq 33$			
$c(DOS07)$	2295,2	2295,2	2295,2			
$c(LM08_1)$	<b>2285,4</b>	<b>2285,4</b>	2285,4			
$c(LM08_2)$	2296,6	2287,8	<b>2280,6</b>			

---

## 6.2 Inversion-free Scheme in $J^c$

The methods before trade field inversions for field multiplications. Depending on the  $I/M$  ratio, this trade might be useful or not. In the following, we determine the maximal  $I/M$  ratio, for which the above methods are faster than this inversion-free scheme.

An efficient precomputation scheme for  $J^c$  coordinates is presented in [LM08] (see Appendix B). It costs

$$cp_{J^c} = (10L - 1)M + (4L + 5)S$$

and needs  $5L+6$  registers.

Since now our precomputed points are in  $J^c$  coordinates, we have to use a different DA Algorithm. As above, we use  $t(2J^m)=3M+5S$  and  $t(2J^m = J)=2M+5S$  (for the latter of which the last multiplication in the pseudocode is omitted, see Appendix Algorithm 6), but for addition we have to use  $t(J+J^c=J^m) = 10M + 6S$  (see Appendix Algorithm 10) and get

$$\text{Cost}_{J^c} = [n \cdot \text{AHD}(12M + 11S) + n(1 - \text{AHD})(3M + 5S)] + cp_{J^c} \quad (26)$$

The following tables show the  $I/M$  break even points for the respective method corresponding to a complete scalar multiplication for different prime fields  $\mathbb{F}_p$  (where  $p$  is an  $n$ -bit prime) and a restricted number of registers  $r$ , where we need at least  $r = 11$  to be able to apply the  $J^c$  method. The respective number  $L$  of precomputed points is shown in table 6 and is also optimal with  $L = 7$  for 41 or more available registers.

Table 6:  $I/M$  break even points for  $n=192$

$r$	11	12	13	14	15	16	17	18-20
$L(J^c)$	1	1	1	1	1	2	2	2
B/E ( <i>DOS07</i> )	269	274	274	276	276	217	217	217
B/E ( <i>LM08<sub>1</sub></i> )	242	272	272	272	278	218	218	222
B/E ( <i>LM08<sub>2</sub></i> )	243	243	273	273	273	213	220	220
$r$	21-23	24	25	26-28	29-30	31-35	36-40	$\geq 41$
$L(J^c)$	3	3	3	4	4	5	6	7
B/E ( <i>DOS07</i> )	172	172	172	160	160	150	142	136
B/E ( <i>LM08<sub>1</sub></i> )	180	182	182	169	169	160	152	146
B/E ( <i>LM08<sub>2</sub></i> )	180	180	184	172	174	165	157	151

For example, if we have a 192-bit scalar and 16 available registers, the  $\mathcal{J}^c$  method outperforms the DOS precomputation scheme for all  $I/M > 217$ , the LM scheme 1 for all  $I/M > 218$  and the LM scheme 2 for all  $I/M > 213$ . Hence, choosing the minimum value of the optimal scheme, for  $r = 16$  it is more efficient to use a sole inversion scheme if  $I/M < 213$ . If the sole inversion schemes can perform optimally (for 41 or more registers), we use LM scheme 2 which outperforms the  $\mathcal{J}^c$  scheme for  $I/M < 151$ .

Table 7: I/M break even points for n=224

$r$	11	12	13	14	15	16	17	18-20
B/E ( <i>DOS07</i> )	318	325	325	331	331	262	262	264
B/E ( <i>LM08<sub>1</sub></i> )	284	321	321	321	330	257	257	264
B/E ( <i>LM08<sub>2</sub></i> )	285	285	322	322	322	250	260	260
$r$	21-23	24	25	26-28	29-30	31-35	36-40	$\geq 41$
B/E ( <i>DOS07</i> )	209	209	209	193	193	179	168	159
B/E ( <i>LM08<sub>1</sub></i> )	214	219	219	202	202	189	178	168
B/E ( <i>LM08<sub>2</sub></i> )	212	212	218	202	207	194	182	173

Table 8: I/M break even points for n=256

$r$	11	12	13	14	15	16	17	18-20
B/E ( <i>DOS07</i> )	367	377	377	385	385	307	307	311
B/E ( <i>LM08<sub>1</sub></i> )	326	370	370	370	382	297	297	307
B/E ( <i>LM08<sub>2</sub></i> )	327	327	371	371	371	286	299	299
$r$	21-23	24	25	26-28	29-30	31-35	36-40	$\geq 41$
B/E ( <i>DOS07</i> )	246	246	246	226	226	208	193	181
B/E ( <i>LM08<sub>1</sub></i> )	249	256	256	235	235	218	203	191
B/E ( <i>LM08<sub>2</sub></i> )	245	245	253	233	240	223	208	196

---

## 7 Conclusion

This paper provides updated operation costs in standard coordinate systems. Using these updated costs, it compares the currently most promising DOS and LM precomputation schemes focussing on given memory constraints. For all  $I/M$  ratios below 150, so in particular for ratios between 80-150 as they typically occur on smart cards, the most efficient way to perform a scalar multiplication with today's standard of 192-bit scalars, is using one of these two sole inversion schemes, in particular if the memory is restricted to 40 registers or less.

A general conclusion which of the DOS or LM scheme should be used is not possible, for it is necessary to scrutinize the respective environment. To perform a scalar multiplication, we need at least 6 registers and if we assume today's standard of  $n=192$  bit, the limit for which it is most efficient to use the DOS scheme is 14 registers. For higher bitlengths  $n$  (224, 256), this limit increases to 20 registers. Above this limit, it is more efficient to use one of the LM schemes. If all schemes can operate optimally with 33 or more registers available, we use LM scheme 2.

---

## References

- [RSA78] Rivest, R.L., Shamir, A., Adleman, L.M., *A Method for Obtaining Digital Signatures and Public-key Cryptosystems*, Communications of the ACM, 21(2), 1978, pp.120-126.
- [ELG85] ElGamal, T., *A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms*, Advances in Cryptology - CRYPTO '84, LNCS 196, Springer, 1985, pp. 10-18.
- [Kob87] Koblitz, N., *Elliptic Curve Cryptosystems*, Mathematics of Computation, vol. 48, no. 177, 1987, pp.203-209.
- [Mil86] Miller, V.S., *Use of Elliptic Curves in Cryptography*, Advances in Cryptology - CRYPTO '85, LNCS 218, Springer, 1986, pp.417-426.
- [HMOV04] Hankerson, D., Menezes, A., Vanstone, S., *Guide to Elliptic Curve Cryptography*, Springer, 2004.
- [Buc04] Buchmann, J., *Einführung in die Kryptographie* Springer, 2004.
- [Sey05] Seysen, M., *Using an RSA Accelerator for Modular Inversion*, Cryptographic Hardware and Embedded Systems, LNCS 3659, Springer, 2005, pp.226-236.
- [CF05] Cohen, H., Frey, G., *Handbook of elliptic and hyperelliptic curve cryptography*, CRC Press, 2005.
- [ELM03] Eisenträger, K., Lauter, K., Montgomery, P., *Fast elliptic curve arithmetic and improved Weil pairing evaluation*, Cryptographers' Track - CT-RSA 2003, LNCS 2612, Springer, 2003, pp.343-354.
- [JP03] Joye, P., Paillier, P., *GCD-Free Algorithms for Computing Modular Inverses*, Cryptographic Hardware and Embedded Systems, LNCS 2779, Springer, 2003, pp.243-253.
- [DOS07] Dahmen, E., Okeya, K. and Schepers, D., *Affine Precomputation with Sole Inversion in Elliptic Curve Cryptography*, Lecture Notes in Computer Science, vol. 4586, Springer, 2007.
- [LM08] Longa, P., Miri, A., *New Composite Operations and Precomputation Scheme for Elliptic Curve Cryptosystems over Prime Fields*, Lecture Notes in Computer Science, vol. 4939, Springer, 2008.
- [LM07] Longa, P., Miri, A., *Fast and Flexible Elliptic Curve Point Arithmetic over Prime Fields*, IEEE Transaction on Computers, vol. 57, no. 3, 2008.

- 
- [Mel06] Meloni, N., *Fast and Secure Elliptic Curve Scalar Multiplication Over Prime Fields Using Special Addition Chains*, Cryptology ePrint Archive, Report 2006/216, 2006.
- [CMO98] Cohen, H., Miyaji, A., Ono, T., *Efficient Elliptic Curve Exponentiation Using Mixed Coordinates*, Advances in Cryptology - ASIACRYPT '98, LNCS 1514, Springer, 1998, pp.51-65.
- [BHLM01] Brown, M., Hankerson, D., Lopez, J., Menezes, A., *Software Implementation of the NIST Elliptic Curves over Prime Fields*, Topics in Cryptology - CT-RSA, 2001, pp.250-265.
- [GAST05] Großschädl, J., Avanzi, R., Savas, E., Tillich, S., *Energy-Efficient Software Implementations of Long Integer Modular Arithmetic*, Proc. Seventh Int'l Workshop Cryptographic Hardware and Embedded Systems, 2003, pp.75-90.
- [LH00] Lim, C.H., Hwang, H.S., *Fast Implementation of Elliptic Curve Arithmetic in  $GF(p^m)$* , Proc. Third Int'l Workshop Practice and Theory in Public Key Cryptography, 2000, pp.405-421.
- [GG03] Gebotys, C.H., Gebotys, R.J., *Secure Elliptic Curve Implementations: An Analysis of Resistance to Power-Attacks in a DSP Processor*, Proc. Fifth Int'l Workshop Cryptographic Hardware and Embedded Systems, 2003, pp.114-128.
- [Ava04] Avanzi, R., *Aspects of Hyperelliptic Curves over Large Prime Fields in Software Implementations*, Proc. Sixth Int'l Workshop Cryptographic Hardware and Embedded Systems, 2004, pp.148-162.
- [BJ03] Billet, O., Joye, M., *Fast Point Multiplication on Elliptic Curves through Isogenies*, Applied Algebra, Algebraic Algorithms and Error-Correcting Codes, LNCS vol. 2643, Springer, 2003, pp.43-50.
- [IEEE] IEEE Standard Specifications for Public-Key Cryptography, *IEEE Std 1363-2000*, The Institute of Electrical and Electronics Engineers (IEEE), 2000.
- [MOC97] Miyaji, A., Ono, T., and Cohen, H., *Efficient Elliptic Curve Exponentiation*, Information and Communication Security - ICICS 1997, LNCS 1334, Springer, 1997, pp. 282-291.
- [DH76] Diffie, W., and Hellman, M., *New directions in cryptography*, IEEE Transactions on Information Theory, vol. IT-22, no. 6, 1976, pp. 644-654.

- 
- [Möl02] Möller, B., *Improved Techniques for Fast Exponentiation*, Information Security and Cryptology - ICISC 2002, LNCS 2587, Springer, 2003, pp. 298-312.
- [Möl04] Möller, B., *Fractional Windows Revisited: Improved Signed-Digit Representations for Efficient Exponentiation*, Information Security and Cryptology - ICISC 2004, LNCS 3506, Springer, 2005, pp. 137-153.
- [MS04] Muir, J., Stinson, D., *New Minimal Weight Representations for Left-to-Right Window Methods*, Cryptographers' Track - CT-RSA 2005, LNCS 3376, Springer, 2005, pp. 366-383.
- [SST04] Schmidt-Samoa, K., Semay, O., Takagi, T., *Analysis of Some Fractional Window Recoding Methods and their Application to Elliptic Curve Cryptosystems*, IEEE Transactions on Computers, vol. 55, no. 1, 2006, pp. 1-10.
- [Sol00] Solinas, J.A., *Efficient Arithmetic on Koblitz Curves*, Design, Codes and Cryptography, vol. 19, 2000, pp. 195-249.
- [BSS99] Blake, I., Seroussi, G., and Smart, N., *Elliptic Curves in Cryptography*, London Mathematical Society, Lecture Note Series 265, Cambridge University Press, 1999.
- [BL07] Bernstein, D.J., Lange, T., *Faster Addition and Doubling on Elliptic Curves*, Lecture Notes in Computer Science, vol. 4833, Springer, 2007.
- [Infineon] Infineon Technologies, <http://www.infineon.com/>.
- [Renesas] Renesas Technologies, <http://www.renesas.com/homepage.jsp/>.

---

## Appendix A

### Pseudocodes of field operations

---

#### Algorithm 2 $t(2P)$

---

**Require:** Point  $P = (X_1, Y_1, Z_1) \in E(\mathbb{F}_p)$

**Ensure:** Point  $2P = (X_3, Y_3, Z_3)$

- 1:  $T_1 \leftarrow X_1^2$
- 2:  $T_2 \leftarrow Z_1^2$
- 3:  $T_3 \leftarrow a \cdot T_2$
- 4:  $T_4 \leftarrow 3T_1$
- 5:  $T_3 \leftarrow T_3 + T_4$  {w}
- 6:  $T_4 \leftarrow Y_1 \cdot Z_1$
- 7:  $T_4 \leftarrow 2T_4$  {s}
- 8:  $T_2 \leftarrow Y_1 \cdot s$  {R}
- 9:  $T_5 \leftarrow T_2^2$
- 10:  $T_2 \leftarrow X_1 + T_2$
- 11:  $T_2 \leftarrow T_2^2$
- 12:  $T_2 \leftarrow T_2 - T_1$
- 13:  $T_2 \leftarrow T_2 - T_5$  {B}
- 14:  $T_1 \leftarrow 2T_2$
- 15:  $T_6 \leftarrow T_3^2$
- 16:  $T_6 \leftarrow T_6 - T_1$  {h}
- 17:  $T_2 \leftarrow T_2 - T_6$
- 18:  $T_2 \leftarrow T_3 \cdot T_2$
- 19:  $T_1 \leftarrow 2T_5$
- 20:  $T_1 \leftarrow T_2 - T_1$  {Y<sub>3</sub>}
- 21:  $T_2 \leftarrow T_4^2$
- 22:  $T_2 \leftarrow T_2 \cdot T_4$  {Z<sub>3</sub>}
- 23:  $T_3 \leftarrow T_6 \cdot T_4$  {X<sub>3</sub>}
- 24: **return**  $(X_3, Y_3, Z_3)$

---

---

**Algorithm 3**  $t(2\mathcal{P})$  if  $a = -3$

---

**Require:** Point  $P = (X_1, Y_1, Z_1) \in E(\mathbb{F}_p)$

**Ensure:** Point  $2P = (X_3, Y_3, Z_3)$

- 1:  $T_1 \leftarrow X_1 - Z_1$
- 2:  $T_2 \leftarrow X_1 + Z_1$
- 3:  $T_2 \leftarrow 3T_2$
- 4:  $T_2 \leftarrow T_2 \cdot T_1$  { $w$ }
- 5:  $T_3 \leftarrow Y_1 \cdot Z_1$
- 6:  $T_3 \leftarrow 2T_3$  { $s$ }
- 7:  $T_1 \leftarrow Y_1 \cdot s$  { $R$ }
- 8:  $T_4 \leftarrow T_1^2$
- 9:  $T_1 \leftarrow X_1 \cdot T_1$
- 10:  $T_1 \leftarrow 2T_1$  { $B$ }
- 11:  $T_5 \leftarrow 2T_1$
- 12:  $T_6 \leftarrow T_2^2$
- 13:  $T_6 \leftarrow T_6 - T_5$  { $h$ }
- 14:  $T_1 \leftarrow T_1 - T_6$
- 15:  $T_1 \leftarrow T_2 \cdot T_1$
- 16:  $T_5 \leftarrow 2T_4$
- 17:  $T_5 \leftarrow T_1 - T_5$  { $Y_3$ }
- 18:  $T_1 \leftarrow T_3^2$
- 19:  $T_1 \leftarrow T_1 \cdot T_3$  { $Z_3$ }
- 20:  $T_2 \leftarrow T_6 \cdot T_3$  { $X_3$ }
- 21: **return**  $(X_3, Y_3, Z_3)$

---

---

**Algorithm 4**  $t(2\mathcal{J})$ 

---

**Require:** Point  $P = (X_1, Y_1, Z_1) \in E(\mathbb{F}_p)$ **Ensure:** Point  $2P = (X_3, Y_3, Z_3)$ 

- 1:  $T_1 \leftarrow Z_1^2$
  - 2:  $T_2 \leftarrow T_1^2$   $\{Z_1^4\}$
  - 3:  $T_2 \leftarrow a \cdot T_2$
  - 4:  $T_3 \leftarrow X_1^2$
  - 5:  $T_4 \leftarrow 3T_3$
  - 6:  $T_4 \leftarrow T_4 + T_2$   $\{\alpha\}$
  - 7:  $T_2 \leftarrow Y_1^2$
  - 8:  $T_5 \leftarrow T_2^2$   $\{Y_1^4\}$
  - 9:  $T_3 \leftarrow T_3 + T_5$
  - 10:  $T_1 \leftarrow T_2 + T_1$
  - 11:  $T_2 \leftarrow X_1 + T_2$
  - 12:  $T_2 \leftarrow T_2^2$
  - 13:  $T_2 \leftarrow T_2 - T_3$
  - 14:  $T_2 \leftarrow 2T_2$   $\{\beta\}$
  - 15:  $T_3 \leftarrow T_4^2$
  - 16:  $T_6 \leftarrow 2T_2$
  - 17:  $T_3 \leftarrow T_3 - T_6$   $\{X_3\}$
  - 18:  $T_5 \leftarrow 8T_5$
  - 19:  $T_2 \leftarrow T_2 - T_3$
  - 20:  $T_2 \leftarrow T_2 \cdot T_4$
  - 21:  $T_2 \leftarrow T_2 - T_5$   $\{Y_3\}$
  - 22:  $T_5 \leftarrow Y_1 + Z_1$
  - 23:  $T_5 \leftarrow T_5^2$
  - 24:  $T_5 \leftarrow T_5 - T_1$   $\{Z_3\}$
  - 25: **return**  $(X_3, Y_3, Z_3)$
-

---

**Algorithm 5**  $t(2\mathcal{J}^c)$ 

---

**Require:** Point  $P = (X_1, Y_1, Z_1, Z_1^2, Z_1^3) \in E(\mathbb{F}_p)$ **Ensure:** Point  $2P = (X_3, Y_3, Z_3, Z_3^2, Z_3^3)$ 

- 1:  $T_2 \leftarrow (Z_1^2)^2$   $\{Z_1^4\}$
  - 2:  $T_2 \leftarrow a \cdot T_2$
  - 3:  $T_3 \leftarrow X_1^2$
  - 4:  $T_4 \leftarrow 3T_3$
  - 5:  $T_4 \leftarrow T_4 + T_2$   $\{\alpha\}$
  - 6:  $T_2 \leftarrow Y_1^2$
  - 7:  $T_5 \leftarrow T_2^2$   $\{Y_1^4\}$
  - 8:  $T_3 \leftarrow T_3 + T_5$
  - 9:  $T_1 \leftarrow T_2 + (Z_1^2)$
  - 10:  $T_2 \leftarrow X_1 + T_2$
  - 11:  $T_2 \leftarrow T_2^2$
  - 12:  $T_2 \leftarrow T_2 - T_3$
  - 13:  $T_2 \leftarrow 2T_2$   $\{\beta\}$
  - 14:  $T_3 \leftarrow T_4^2$
  - 15:  $T_6 \leftarrow 2T_2$
  - 16:  $T_3 \leftarrow T_3 - T_6$   $\{X_3\}$
  - 17:  $T_5 \leftarrow 8T_5$
  - 18:  $T_2 \leftarrow T_2 - T_3$
  - 19:  $T_2 \leftarrow T_2 \cdot T_4$
  - 20:  $T_2 \leftarrow T_2 - T_5$   $\{Y_3\}$
  - 21:  $T_5 \leftarrow Y_1 + Z_1$
  - 22:  $T_5 \leftarrow T_5^2$
  - 23:  $T_5 \leftarrow T_5 - T_1$   $\{Z_3\}$
  - 24:  $T_1 \leftarrow T_5^2$   $\{Z_3^2\}$
  - 25:  $T_4 \leftarrow T_1 \cdot T_5$   $\{Z_3^3\}$
  - 26: **return**  $(X_3, Y_3, Z_3, Z_3^2, Z_3^3)$
-

---

**Algorithm 6**  $t(2\mathcal{J}^m)$ 

---

**Require:** Point  $P = (X_1, Y_1, Z_1, aZ_1^4) \in E(\mathbb{F}_p)$ **Ensure:** Point  $2P = (X_3, Y_3, Z_3, aZ_3^4)$ 

- 1:  $T_1 \leftarrow X_1^2$
  - 2:  $T_2 \leftarrow 3T_1$
  - 3:  $T_2 \leftarrow T_2 + (aZ_1^4)$   $\{\alpha\}$
  - 4:  $T_3 \leftarrow Y_1^2$
  - 5:  $T_4 \leftarrow T_3^2$   $\{Y_1^4\}$
  - 6:  $T_1 \leftarrow T_1 + T_4$
  - 7:  $T_3 \leftarrow X_1 + T_3$
  - 8:  $T_3 \leftarrow T_3^2$
  - 9:  $T_3 \leftarrow T_3 - T_1$
  - 10:  $T_3 \leftarrow 2T_3$   $\{\beta\}$
  - 11:  $T_1 \leftarrow T_2^2$
  - 12:  $T_5 \leftarrow 2T_3$
  - 13:  $T_1 \leftarrow T_1 - T_5$   $\{X_3\}$
  - 14:  $T_4 \leftarrow 8T_4$
  - 15:  $T_3 \leftarrow T_3 - T_1$
  - 16:  $T_3 \leftarrow T_3 \cdot T_2$
  - 17:  $T_3 \leftarrow T_3 - T_4$   $\{Y_3\}$
  - 18:  $T_2 \leftarrow Y_1 \cdot Z_1$
  - 19:  $T_2 \leftarrow 2T_4$   $\{Z_3\}$
  - 20:  $T_5 \leftarrow 16T_4$
  - 21:  $T_5 \leftarrow T_5 \cdot (aZ_1^4)$
  - 22: **return**  $(X_3, Y_3, Z_3, aZ_3^4)$
-

---

**Algorithm 7**  $t(2\mathcal{J})$  if  $a = -3$

---

**Require:** Point  $P = (X_1, Y_1, Z_1) \in E(\mathbb{F}_p)$

**Ensure:** Point  $2P = (X_3, Y_3, Z_3)$

- 1:  $T_1 \leftarrow Z_1^2$
- 2:  $T_2 \leftarrow X_1 + T_1$
- 3:  $T_3 \leftarrow X_1 - T_1$
- 4:  $T_3 \leftarrow 3T_3$
- 5:  $T_3 \leftarrow T_3 \cdot T_2$  { $\alpha$ }
- 6:  $T_2 \leftarrow Y_1^2$
- 7:  $T_4 \leftarrow T_2^2$  { $Y_1^4$ }
- 8:  $T_1 \leftarrow T_2 + T_1$
- 9:  $T_2 \leftarrow X_1 \cdot T_2$
- 10:  $T_2 \leftarrow 4T_2$  { $\beta$ }
- 11:  $T_5 \leftarrow T_3^2$
- 12:  $T_6 \leftarrow 2T_2$
- 13:  $T_5 \leftarrow T_5 - T_6$  { $X_3$ }
- 14:  $T_4 \leftarrow 8T_4$
- 15:  $T_2 \leftarrow T_2 - T_5$
- 16:  $T_2 \leftarrow T_2 \cdot T_3$
- 17:  $T_2 \leftarrow T_2 - T_4$  { $Y_3$ }
- 18:  $T_3 \leftarrow Y_1 + Z_1$
- 19:  $T_3 \leftarrow T_3^2$
- 20:  $T_3 \leftarrow T_3 - T_1$  { $Z_3$ }
- 21: **return**  $(X_3, Y_3, Z_3)$

---

---

**Algorithm 8**  $t(2\mathcal{J}^c)$  if  $a = -3$

---

**Require:** Point  $P = (X_1, Y_1, Z_1, Z_1^2, Z_1^3) \in E(\mathbb{F}_p)$

**Ensure:** Point  $2P = (X_3, Y_3, Z_3, Z_3^2, Z_3^3)$

- 1:  $T_1 \leftarrow X_1 + (Z_1^2)$
- 2:  $T_2 \leftarrow X_1 - (Z_1^2)$
- 3:  $T_2 \leftarrow 3T_2$
- 4:  $T_2 \leftarrow T_2 \cdot T_1$  { $\alpha$ }
- 5:  $T_1 \leftarrow Y_1^2$
- 6:  $T_3 \leftarrow T_1^2$  { $Y_1^4$ }
- 7:  $T_4 \leftarrow T_1 + (Z_1^2)$
- 8:  $T_1 \leftarrow X_1 \cdot T_1$
- 9:  $T_1 \leftarrow 4T_1$  { $\beta$ }
- 10:  $T_5 \leftarrow T_2^2$
- 11:  $T_6 \leftarrow 2T_1$
- 12:  $T_5 \leftarrow T_5 - T_6$  { $X_3$ }
- 13:  $T_3 \leftarrow 8T_3$
- 14:  $T_1 \leftarrow T_1 - T_5$
- 15:  $T_1 \leftarrow T_1 \cdot T_2$
- 16:  $T_1 \leftarrow T_1 - T_3$  { $Y_3$ }
- 17:  $T_2 \leftarrow Y_1 + Z_1$
- 18:  $T_2 \leftarrow T_2^2$
- 19:  $T_2 \leftarrow T_2 - T_1$  { $Z_3$ }
- 20:  $T_3 \leftarrow T_2^2$  { $Z_3^2$ }
- 21:  $T_4 \leftarrow T_2 \cdot T_3$  { $Z_3^3$ }
- 22: **return**  $(X_3, Y_3, Z_3, Z_3^2, Z_3^3)$

---

---

**Algorithm 9**  $t(2A = \mathcal{J})$ 

---

**Require:** Point  $P = (X_1, Y_1) \in E(\mathbb{F}_p)$ **Ensure:** Point  $2P = (X_3, Y_3, Z_3)$ 

- 1:  $T_2 \leftarrow Y_1^2$
  - 2:  $T_4 \leftarrow X_1 + T_2$
  - 3:  $T_4 \leftarrow T_4^2$
  - 4:  $T_2 \leftarrow T_2^2$  { $Y_1^4$ }
  - 5:  $T_1 \leftarrow X_1^2$
  - 6:  $T_4 \leftarrow T_4 - T_1$
  - 7:  $T_4 \leftarrow T_4 - T_2$
  - 8:  $T_3 \leftarrow 2T_4$
  - 9:  $T_4 \leftarrow 2T_3$
  - 10:  $T_1 \leftarrow 3T_1$
  - 11:  $T_5 \leftarrow 3T_1 + a$
  - 12:  $T_1 \leftarrow T_5^2$
  - 13:  $T_4 \leftarrow T_1 - T_4$  { $X_3$ }
  - 14:  $T_6 \leftarrow T_3 - T_4$
  - 15:  $T_6 \leftarrow T_5 \cdot T_6$
  - 16:  $T_5 \leftarrow 8T_2$
  - 17:  $T_5 \leftarrow T_6 - T_5$  { $Y_3$ }
  - 18:  $T_3 \leftarrow 2Y_1$  { $Z_3$ }
  - 19: **return**  $(X_3, Y_3, Z_3)$
-

---

**Algorithm 10**  $t(\mathcal{J}^c + \mathcal{J} = \mathcal{J}^m)$ 

---

**Require:** Points  $P = (X_1, Y_1, Z_1, Z_1^2, Z_1^3)$ ,  $Q = (X_2, Y_2, Z_2) \in E(\mathbb{F}_p)$ **Ensure:** Point  $P + Q = (X_5, Y_5, Z_5, aZ_5^4)$ 

- 1:  $T_1 \leftarrow (Z_1^2) \cdot X_2$
  - 2:  $T_2 \leftarrow Z_2^2$
  - 3:  $T_3 \leftarrow T_2 \cdot X_1$
  - 4:  $T_1 \leftarrow T_1 - T_3$  { $\beta$ }
  - 5:  $T_4 \leftarrow Z_1 + Z_2$
  - 6:  $T_4 \leftarrow T_4^2$
  - 7:  $T_4 \leftarrow T_4 - T_2$
  - 8:  $T_4 \leftarrow T_4 - (Z_1^2)$
  - 9:  $T_4 \leftarrow T_4 \cdot T_1$  { $Z_5$ }
  - 10:  $T_2 \leftarrow T_2 \cdot Z_2$  { $Z_2^3$ }
  - 11:  $T_2 \leftarrow T_2 \cdot Y_1$
  - 12:  $T_2 \leftarrow 4T_2$
  - 13:  $T_5 \leftarrow (Z_1^3) \cdot Y_2$
  - 14:  $T_5 \leftarrow T_5 - T_2$
  - 15:  $T_5 \leftarrow 2T_5$  { $\alpha$ }
  - 16:  $T_6 \leftarrow T_1^2$
  - 17:  $T_1 \leftarrow T_6 \cdot T_1$  { $\beta^3$ }
  - 18:  $T_1 \leftarrow 2T_1$
  - 19:  $T_2 \leftarrow T_2 \cdot T_1$
  - 20:  $T_6 \leftarrow T_6 \cdot T_3$
  - 21:  $T_6 \leftarrow 4T_6$
  - 22:  $T_1 \leftarrow T_1 + T_6$
  - 23:  $T_1 \leftarrow 2T_1$
  - 24:  $T_3 \leftarrow T_5^2$
  - 25:  $T_3 \leftarrow T_3 - T_1$  { $X_5$ }
  - 26:  $T_1 \leftarrow T_6 - T_3$
  - 27:  $T_1 \leftarrow T_5 \cdot T_1$
  - 28:  $T_1 \leftarrow T_1 - T_2$  { $Y_5$ }
  - 29:  $T_5 \leftarrow T_4^2$
  - 30:  $T_5 \leftarrow T_5^2$
  - 31:  $T_5 \leftarrow a \cdot T_5$  { $aZ_5^4$ }
  - 32: **return**  $(X_5, Y_5, Z_5, aZ_5^4)$
-

---

**Algorithm 11**  $t(\mathcal{J} + \mathcal{A} = \mathcal{J}^m)$ 

---

**Require:** Points  $P = (X_1, Y_1, Z_1), Q = (X_2, Y_2) \in E(\mathbb{F}_p)$ **Ensure:** Point  $P + Q = (X_5, Y_5, Z_5)$ 

- 1:  $T_1 \leftarrow Z_1^2$
  - 2:  $T_2 \leftarrow T_1 \cdot X_2$
  - 3:  $T_2 \leftarrow T_2 - X_1$   $\{\beta\}$
  - 4:  $T_4 \leftarrow T_2^2$
  - 5:  $T_3 \leftarrow Z_1 + \beta$
  - 6:  $T_3 \leftarrow T_3^2$
  - 7:  $T_3 \leftarrow T_3 - T_4$
  - 8:  $T_3 \leftarrow T_3 - T_1$   $\{Z_5\}$
  - 9:  $T_1 \leftarrow T_1 \cdot Z_1$   $\{Z_1^3\}$
  - 10:  $T_1 \leftarrow T_1 \cdot Y_2$
  - 11:  $T_1 \leftarrow T_1 - Y_1$
  - 12:  $T_1 \leftarrow 2T_1$   $\{\alpha\}$
  - 13:  $T_5 \leftarrow T_4 \cdot T_2$   $\{\beta^3\}$
  - 14:  $T_5 \leftarrow 2T_5$
  - 15:  $T_4 \leftarrow T_4 \cdot X_1$
  - 16:  $T_4 \leftarrow 4T_5$
  - 17:  $T_2 \leftarrow T_1^2$
  - 18:  $T_6 \leftarrow T_5 - T_4$
  - 19:  $T_6 \leftarrow 2T_6$
  - 20:  $T_2 \leftarrow T_2 - T_6$   $\{X_5\}$
  - 21:  $T_5 \leftarrow T_5 \cdot Y_1$
  - 22:  $T_5 \leftarrow 8T_5$
  - 23:  $T_4 \leftarrow T_4 - T_2$
  - 24:  $T_1 \leftarrow T_1 \cdot T_4$
  - 25:  $T_1 \leftarrow T_1 - T_5$   $\{Y_5\}$
  - 26:  $T_5 \leftarrow T_1^2$
  - 27:  $T_5 \leftarrow T_5^2$
  - 28:  $T_5 \leftarrow a \cdot T_5$   $\{aZ_5^4\}$
  - 29: **return**  $(X_5, Y_5, Z_5, aZ_5^4)$
-

---

**Appendix B:**  
**Precomputation costs for section 6.1**

$\mathcal{J} \rightarrow \mathcal{A}$

$2P$	$t(2\mathcal{A} = \mathcal{J})$	$1M + 5S$
$3P$	$t(\mathcal{A} + \mathcal{J} = \mathcal{J})$	$7M + 4S$
$5P, 7P, \dots$	$t(\mathcal{J} + \mathcal{J})$	$(L - 1)(11M + 5S)$
modified Montgomery		$(L - 1)(4M + 5S) + 3M + S + I$
TOTAL		$(15L - 4)M + (6L + 4)S + I$

$\mathcal{J}^c \rightarrow \mathcal{A}$

$2P$	$t(2\mathcal{A} = \mathcal{J}^c)$	$2M + 5S$
$3P$	$t(\mathcal{A} + \mathcal{J}^c = \mathcal{J}^c)$	$7M + 4S$
$5P, 7P, \dots$	$t(\mathcal{J}^c + \mathcal{J}^c)$	$(L - 1)(10M + 4S)$
modified Montgomery		$(L - 1)(4M + 5S) + 3M + S + I$
TOTAL		$(14L - 2)M + (5L + 5)S + I$

$\mathcal{J}^m \rightarrow \mathcal{A}$

$2P$	$t(2\mathcal{A} = \mathcal{J}^m)$	$2M + 5S$
$3P$	$t(\mathcal{A} + \mathcal{J}^m = \mathcal{J}^m)$	$9M + 5S$
$5P, 7P, \dots$	$t(\mathcal{J}^m + \mathcal{J}^m)$	$(L - 1)(12M + 7S)$
modified Montgomery		$(L - 1)(4M + 5S) + 3M + S + I$
TOTAL		$(16L - 2)M + (8L + 3)S + I$

$\mathcal{J}^c$

$2P$	$t(2\mathcal{A} = \mathcal{J}^c)$	$2M + 5S$
$3P$	$t(\mathcal{A} + \mathcal{J}^c = \mathcal{J}^c)$	$7M + 4S$
$5P, 7P, \dots$	$t(\mathcal{J}^c + \mathcal{J}^c)$	$(L - 1)(10M + 4S)$
TOTAL		$(10L - 1)M + (4L + 5)S$

---

## Appendix C: Matlab Codes for section 6

Matlab Code: "cost.m"

```
function res = cost(n,L,x)
format short
% Comparison of Jc with
% x=1: DOS07
% x=2: LMO8 scheme 1
% x=3: LMO8 scheme 2
% x=4: Jc scheme

if L==0
    res=0;
    return
end

% calculate average hamming density (AHD) for fractional window method;
% L points; digit set D={0,+1,+2,...,+-(2L+1)}
tmp = floor(log2(L+1));
AHD = 1 / ((L+1)/2^tmp + tmp + 2);

% calculate cp (cost of precomputation of the points [3]P,[5]P,...,[2L+1]P)
% calculate cs (cost of DA algorithm:
% for x=1,2,3 according to LMO8 with precomputed points in affine coordinates
% for x=4 according to LMO8 with precomputed points in J^c coordinates
syms M;
syms S;
S = 0.8*M;

if x==1
    cp = (10*L-2)*M + (4*L+4)*S;
    cs = n*AHD*(11*M+7*S) + n*(1-AHD)*(3*M+5*S);
elseif x==2
    cp = (9*L)*M + (3*L+5)*S;
    cs = n*AHD*(11*M+7*S) + n*(1-AHD)*(3*M+5*S);
elseif x==3
    cp = (9*L)*M + (2*L+6)*S;
    cs = n*AHD*(11*M+7*S) + n*(1-AHD)*(3*M+5*S);
elseif x==4
    cp = (10*L-1)*M + (4*L+5)*S;
    cs = n*AHD*(12*M+11*S) + n*(1-AHD)*(3*M+5*S);
end
```

---

% total cost (in field multiplications)

res = (cp+cs)/M;

### Matlab Code: "mem.m"

```
function res = mem(r,x)
```

```
% number of bits of the scalar
```

```
n = 256;
```

```
% determine maximal number of resp. precomputable points for r registers
```

```
if x==1
```

```
    L = floor((r-4)/2);
```

```
elseif x==2
```

```
    L = floor((r-3)/3);
```

```
elseif x==3
```

```
    L = floor((r-1)/4);
```

```
elseif x==4
```

```
    L = floor((r-6)/5);
```

```
end
```

```
% limit L to 7; otherwise not efficient
```

```
if L>7
```

```
    L=7;
```

```
end
```

```
res = [L cost(n,L,x)];
```

### Matlab Code: "all.m"

```
% consider memory constraints from min to max;
```

```
% we need min=6 to compare the three precomputation schemes,
```

```
% we need min=11 to be able to compare with the Jc method,
```

```
% we need max=41 to compare all four methods with L=7
```

```
min = 6;
```

```
max = 41;
```

```
% result values of the three A methods and the Jc method
```

```
L = zeros(4,max-5);
```

```
c = zeros(4,max-5); % total cost
```

```
BE = zeros(3,max-5); % Break-Even points for I/M ratio
```

---

```
% loop over all memories and methods
for r=min:max
    for i=1:4
        t = mem(r,i);
        L(i,r-5) = double(t(1));
        c(i,r-5) = double(t(2));
    end
    for j=1:3
        if c(4,r-5)==0
            BE(j,r-5) = 0;
        else
            BE(j,r-5) = c(4,r-5)-c(j,r-5);
        end
    end
end
end
```