Affine Precomputation with Sole Inversion in Elliptic Curve Cryptography

Erik Dahmen,[1] Katsuyuki Okeya,[2] and Daniel Schepers[1]

[1] Technische Universität Darmstadt, Fachbereich Informatik,
Hochschulstr.10, D-64289 Darmstadt, Germany
{dahmen,schepers}@cdc.informatik.tu-darmstadt.de
[2] Hitachi, Ltd., Systems Development Laboratory,
1099, Ohzenji, Asao-ku, Kawasaki-shi, Kanagawa-ken, 215-0013, Japan
katsuyuki.okeya.ue@hitachi.com

Abstract. This paper presents a new approach to precompute all odd points $[3]P, [5]P, \dots, [2k-1]P$, $k \geq 2$ on an elliptic curve over $\mathbb{F}_p$. Those points are required for the efficient evaluation of a scalar multiplication, the most important operation in elliptic curve cryptography. The proposed method precomputes the points in affine coordinates and needs only one single field inversion for the computation. The new method is superior to all known methods that also use one field inversion. Compared to methods that require several field inversions for the precomputation, the proposed method is faster for a broad range of ratios of field inversions and field multiplications. The proposed method benefits especially from ratios as they occur on smart cards.

Keywords: *affine coordinates, elliptic curve cryptosystem, precomputation, scalar multiplication*

1 Introduction

Koblitz [Kob87] and Miller [Mil86] independently proposed to use elliptic curves for cryptographic purposes. The main advantage of elliptic curves is, that high security can be achieved by using only small key sizes [BSS99].

One of the most time-consuming operation in cryptosystems based on elliptic curves is a scalar multiplication $[u]P$, where u is the scalar and P is a point on an elliptic curve over $\mathbb{F}_p$. Scalar multiplications are computed using the double-and-add algorithm. The number of point additions required by this algorithm can be reduced by representing the scalar in a signed representation that provides fewer non-zero digits [Ava04,Möl02,Möl04,MS04,OSST04,Sol00,SST04]. In this case, the double-and-add algorithm requires several precomputed points. For efficiency reasons, those points are usually represented in affine coordinates [CMO98]. If the point P is not fixed, the precomputation cannot be performed offline and requires a significant amount of time, since expensive field inversions are required to precompute points in affine coordinates. Scalar multiplications with non-fixed points for example occur in the Diffie-Hellman key exchange [DH76] and the verification step of the elliptic curve digital signature algorithm [JM99]. One important research goal is to reduce the number of field inversions that are involved in the precomputation. In [CJLM06], a method to compute $[3]P$ with only one inversion was proposed.

This paper generalizes this method and presents a new approach to precompute points on an elliptic curve over $\mathbb{F}_p$. The proposed scheme computes all odd points $[3]P, \ldots, [2k-1]P$, $k \geq 2$ by using only one single field inversion, independent of the number of points to precompute. The main idea is to use a recursive strategy to express all values that have to be inverted using only known parameters. Then, all values are inverted simultaneously using the Montgomery trick, e.g. see [CF05] p. 209. Further, the proposed scheme does not require additional memory for temporary calculations.

Compared to previous approaches for the precomputation (e.g. [CMO98]), the proposed method benefits from a large ratio of inversions and multiplications (I/M). This ratio is especially large on smart cards that are equipped with a cryptographic coprocessor, which is usually the case [Infineon,Renesas]. In [Sey05], Seysen states that on such smart cards an I/M ratio of $I > 100M$ is realistic. In [CF05,ELM03,JP03], the authors state that on smart cards with a cryptographic coprocessor, the inversion is best computed using Fermat's little theorem. This approach requires about $\log_2 p$ field multiplications, where p is the prime that defines the field. Note that p must be at least 160 bit to guarantee security.

After introducing the proposed method, this paper states a thorough comparison with known methods for the precomputation. Rather than specifying the advantage of a certain method for a given I/M ratio, the *I/M break even points* of the different methods are estimated. The I/M break even points provide information about which method is the most efficient for a certain I/M ratio. As it will turn out, the proposed method is the most efficient for I/M ratios as they occur on smart cards.

The remainder of this paper is organized as follows: Section 2 introduces the basics of elliptic curves and scalar multiplications. Section 3 reviews known methods for the precomputation. Section 4 describes the proposed scheme. Section 5 compares the proposed scheme with known methods and Section 6 states the conclusion.

2 Scalar Multiplications in Elliptic Curve Cryptography

An elliptic curve over a prime field $\mathbb{F}_p$ is defined by the implicit equation $E : y^2 = x^3 + ax + b$, where $a, b \in \mathbb{F}_p$ and $p > 3$ prime. A further condition on a and b is, that the so-called discriminant $\Delta = 4a^3 + 27b^2$ is non-zero. The points on an elliptic curve can be used to construct an abelian group $E(\mathbb{F}_p)$ with identity element $\mathcal{O}$ called the "point at infinity" [BSS99]. Point additions ($P + Q$) and doublings ($2P$) are denoted by ECADD and ECDBL, respectively. Points on an elliptic curve can be represented in several coordinate systems, such as affine ($\mathcal{A}$), projective ($\mathcal{P}$), Jacobian ($\mathcal{J}$), modified Jacobian ($\mathcal{J}^m$), and Chudnovsky Jacobian ($\mathcal{J}^c$) coordinates [CMO98]. The number of field multiplications (M), squarings (S), and inversions (I) required for an ECADD or ECDBL operation depends on the coordinate system used to represent the points. See [CMO98] for an overview of the costs and explicit formulas.

A scalar multiplication $[u]P$ of a point $P \in E(\mathbb{F}_p)$ and a scalar $u > 0$ is defined by adding P to itself u times. An efficient method to compute a scalar multiplication is the *double-and-add algorithm* shown in Algorithm 1. This algorithm uses an n-bit *base-2 representation* $(u_{n-1}, \ldots, u_0)$ of u, e.g. the binary representation or one of the representations proposed in [Ava04,Möl02,Möl04,MS04,OSST04,Sol00,SST04].

Algorithm 1 Double-and-Add Algorithm

```
Require: Point $P \in E(\mathbb{F}_p)$, $n$-bit scalar $u$.
Ensure: Scalar multiplication $[u]P$
1: $X \leftarrow \mathcal{O}$
2: for $i = n-1$ down to 0 do
3:     $X \leftarrow$ ECDBL($X$)
4:     if $u_i \neq 0$ then $X \leftarrow$ ECADD($X, [u_i]P$)
5: end for
6: return $X$
```

Algorithm 1 performs a point doubling in each iteration (line 3) and a point addition each time the current digit u_i is non-zero (line 4). Hence a scalar multiplication needs $n \cdot$ AHD ECADD $+\, n$ ECDBL, where AHD denotes the *average Hamming density*, i.e. the average density of non-zero digits in the base-2 representation of u. The points $[u_i]P$ required in line 4 are precomputed beforehand. Which and how many points must be precomputed depends on the base-2 representation used for u.

To reduce the required number of field operations in the different steps of Algorithm 1, the authors of [CMO98] represent the points using mixed coordinates. They use $\mathcal{J}^m$ coordinates for the result of a doubling followed by a doubling ($u_i = 0$) and $\mathcal{J}$ coordinates for the result of a doubling followed by an addition ($u_i \neq 0$). The costs for a doubling then are $4M + 4S$ and $3M + 4S$, respectively. The precomputed points $[u_i]P$ are represented either in $\mathcal{A}$ or $\mathcal{J}^c$ coordinates. The costs for an addition then are $9M+5S$ or $12M+5S$, respectively. Using mixed coordinates, a scalar multiplication with Algorithm 1 requires

$$\mathrm{cs}_{\mathcal{A}} = n \cdot \mathrm{AHD}(9M+5S) + n\big(\mathrm{AHD}(3M+4S) + (1-\mathrm{AHD})(4M+4S)\big) \quad (1)$$

$$\mathrm{cs}_{\mathcal{J}^c} = n \cdot \mathrm{AHD}(12M+5S) + n\big(\mathrm{AHD}(3M+4S) + (1-\mathrm{AHD})(4M+4S)\big) \quad (2)$$

with precomputed points in $\mathcal{A}$ and $\mathcal{J}^c$ coordinates, respectively.

A very flexible base-2 representation is the *fractional window recoding* method [Möl02,Möl04,SST04]. For an arbitrary $k \geq 1$, this representation uses the digits in the *digit set* $\mathcal{D}_k = \{0, \pm1, \pm3, \ldots, \pm(2k-1)\}$. When used with Algorithm 1, the $k-1$ points $[3]P, [5]P, \ldots, [2k-1]P$ must be precomputed. Note, that only the positive points must be precomputed, since point inversions are virtually for free, e.g. if $[-3]P$ is required by Algorithm 1, it is obtained from $[3]P$ by an "on-the-fly" point inversion [BSS99]. The AHD of this representation is

$$\mathrm{AHD}_k = \left(\frac{k}{2^{\lfloor \log_2 k \rfloor}} + \lfloor \log_2 k \rfloor + 2\right)^{-1} \tag{3}$$

which is minimal among all base-2 representations that use this digit set [Möl04]. Note, that if $k = 2^{w-2}$ for some $w \geq 2$, the fractional window recoding method has the same AHD as the *width-w non adjacent form* [Sol00] and its analogs [Ava04,MS04,OSST04], i.e. $1/(w+1)$.

Increasing the parameter k on the one hand decreases the AHD and therefore the number of ECADD operations in Algorithm 1 and on the other hand increases the number of points that must be precomputed. Therefore, increasing k does not automatically yield a better total performance, since additional ECADD and ECDBL operations are required for the precomputation.

3 Precomputing the required Points

In this section, several methods for the precomputation of the $k-1$ points $[3]P, [5]P, \dots, [2k-1]P$ required by the fractional window recoding method are reviewed. Recall that according to [CMO98], the precomputed points should be represented in $\mathcal{A}$ or $\mathcal{J}^c$ coordinates. The most straightforward method is to compute each point separately using the chain $P \to [2]P \to [3]P \to [5]P \to \dots \to [2k-1]P$. This method needs

$$\mathrm{cp}_{\mathcal{A}} = 2kM + (k+1)S + kI \tag{4}$$
$$\mathrm{cp}_{\mathcal{J}^c} = (11k-6)M + (3k+3)S \tag{5}$$

when using $\mathcal{A}$ or $\mathcal{J}^c$ coordinates for the precomputed points, respectively. Storing the points requires $2(k-1)$ registers for affine coordinates and $5(k-1)$ registers for Chudnovsky Jacobian coordinates.

The following methods compute the points in $\mathcal{A}$ coordinates and trade inversions for multiplications using the *Montgomery trick* for simultaneous inversions [CF05] p. 209. This algorithm computes n inverses using $3nM + I$.

Let $k = 2^{w-2}$ for some $w \geq 2$. In [CMO98] the authors compute the points using the chain $P \to 2P \to [3]P, [4]P \to [5]P, [7]P, [8]P \to \dots \to [2^{w-3}+1]P, \dots, [2^{w-2}-1]P, [2^{w-2}]P \to [2^{w-2}+1]P, \dots, [2^{w-1}-1]P$. The inversions required in each of the $w-1$ steps are computed simultaneously using the Montgomery trick. In terms of k, this method needs

$$\mathrm{cp}_{\mathrm{CMO}} = (5k + 2\lceil \log_2 k \rceil - 8)M + (k + 2\lceil \log_2 k \rceil - 1)S + (\lceil \log_2 k \rceil + 1)I. \tag{6}$$

The logarithm has to be rounded up to cover the case where k is chosen such that it is not a power of 2. Storing the points requires $2(k-1)$ registers.

The last method is a straightforward method that first computes the points separately in $\mathcal{P}, \mathcal{J}, \mathcal{J}^m$, or $\mathcal{J}^c$ coordinates. Then the points are converted to $\mathcal{A}$

coordinates. A conversion from $\mathcal{P}$ to $\mathcal{A}$ needs $2M + I$. A conversion from $\mathcal{J}$, $\mathcal{J}^c$, or $\mathcal{J}^m$ to $\mathcal{A}$ needs $3M + S + I$. The inversions required for the conversion are computed simultaneously using the Montgomery trick. These methods need

$$\begin{aligned}
\mathrm{cp}_{\mathcal{P}\to\mathcal{A}} &= (17k-10)M + (2k+3)S + I &(7)\\
\mathrm{cp}_{\mathcal{J}\to\mathcal{A}} &= (18k-14)M + (5k+1)S + I &(8)\\
\mathrm{cp}_{\mathcal{J}^c\to\mathcal{A}} &= (17k-12)M + (4k+2)S + I &(9)\\
\mathrm{cp}_{\mathcal{J}^m\to\mathcal{A}} &= (19k-15)M + (7k-3)S + I &(10)
\end{aligned}$$

Storing the points in affine coordinates requires $2(k-1)$ registers. However, it has to be considered that the points require more memory prior to conversion to affine coordinates. The required number of registers is $3(k-1)$ for $\mathcal{P}$ and $\mathcal{J}$ coordinates, $5(k-1)$ for $\mathcal{J}^c$ coordinates, and $4(k-1)$ for $\mathcal{J}^m$ coordinates.

4 Proposed Scheme

This section describes the proposed scheme. The proposed scheme computes the required points $[3]P, [5]P, \ldots, [2k-1]P$, $k \geq 2$ directly in affine coordinates using only one field inversion. The proposed scheme needs $(10k-11)M + (4k)S + I$ for the precomputation and $2(k-1)$ registers to store the points.

The proposed scheme computes $[2i-1]P = (x_{i+1}, y_{i+1})$ as $[2]P + [2i-3]P$, $i = 2, \ldots, k$ and therefore the computation of $[2]P$ is also required. The formulas to compute the points in affine coordinates are

$$\begin{array}{rll}
[2]P = (x_2, y_2): & \lambda_1 = \frac{(3x_1^2+a)}{(2y_1)} & \begin{array}{l} x_2 = \lambda_1^2 - 2x_1 \\ y_2 = \lambda_1(x_1 - x_2) - y_1 \end{array} \\
[3]P = (x_3, y_3): & \lambda_2 = \frac{(y_2-y_1)}{(x_2-x_1)} & \begin{array}{l} x_3 = \lambda_2^2 - x_2 - x_1 \\ y_3 = \lambda_2(x_2 - x_3) - y_2 \end{array} \\
[2i-1]P = (x_{i+1}, y_{i+1}): & \lambda_i = \frac{(y_i-y_2)}{(x_i-x_2)} & \begin{array}{l} x_{i+1} = \lambda_i^2 - x_2 - x_i \\ y_{i+1} = \lambda_i(x_2 - x_{i+1}) - y_2 \end{array}
\end{array} \quad (11)$$

The most time consuming operation when computing points in affine coordinates is the field inversion required to invert the denominator of the λ_i. Call those denominators δ_i. According to the last section, it is possible to compute field inversions simultaneously using the Montgomery trick [CF05]. However to do so, *all values to invert must be known.* For the precomputation this is not the case, since each point depends on a previous computed point, e.g. $[7]P = [2]P + [5]P$.

The main idea of the proposed scheme is to write down all δ_i using only the base point $P = (x_1, y_1)$ and the elliptic curve parameters a and b. Then, all δ_i are known and can be inverted simultaneously using the Montgomery trick. The proposed strategy is divided into four steps. The pseudocode of those steps can be found in Appendix A.

Step 1: The first step computes $d_1, \ldots, d_k$, such that $d_i = d_1^2 \cdot \ldots \cdot d_{i-1}^2 \cdot \delta_i$ holds for $i = 1, \ldots, k$. This is done by the following recursive strategy which successively substitutes the formulas for x_i, y_i in the formulas for x_{i+1}, y_{i+1}.

$$\begin{array}{rl@{\qquad}rl}
[2]P: & d_1 = 2y_1 & [7]P: & d_4 = A_4^2 - D_4 - B_4 \\
 & & & A_4 = -d_3 \cdot A_3 - C_4 \\
[3]P: & d_2 = A_2^2 - B_2 & & B_4 = d_3^2\,(B_3 + 3D_3) \\
 & A_2 = 3x_1^2 + a & & C_4 = D_3\,(2A_3 + C_3) \\
 & B_2 = d_1^2 \cdot 3x_1 & & D_4 = d_3^3 \\
 & & & \\
[5]P: & d_3 = A_3^2 - 2D_3 - B_3 & [2i-1]P: & d_i = A_i^2 - D_i - B_i \\
 & A_3 = -d_2 \cdot A_2 - C_3 & i > 4 & A_i = -d_{i-1} \cdot A_{i-1} - C_i \\
 & B_3 = d_2^2 \cdot B_2 & & B_i = d_{i-1}^2 \cdot B_{i-1} \\
 & C_3 = d_1^4 & & C_i = D_{i-1} \cdot C_{i-1} \\
 & D_3 = d_2^3 & & D_i = d_{i-1}^3
\end{array}$$

For example, $d_1 = 2y_1 = \delta_1$ and

$$\begin{aligned}
d_2 &= A_2^2 - B_2 \\
&= (3x_1^2 + a)^2 - (2y_1)^2 \cdot 3x_1 \\
&= (2y_1)^2 \left(\left(\frac{3x_1^2 + a}{2y_1} \right)^2 - 2x_1 - x_1 \right) \\
&= (2y_1)^2 \left((\lambda_1^2 - 2x_1) - x_1 \right) \\
&= (2y_1)^2 \,(x_2 - x_1) = d_1^2 \cdot \delta_2.
\end{aligned}$$

Step 2: The second step computes the inverses of $d_1, \ldots, d_k$ using the Montgomery Trick [CF05]. At first, the values $e_i = \prod_{j=1}^{i} d_i$ are computed for $i = 1, \ldots, k$. Next, the inverse of e_k,

$$e_k^{-1} = (d_1 \cdot \ldots \cdot d_k)^{-1} = d_1^{-1} \cdot \ldots \cdot d_k^{-1}$$

is computed. Then, the inverses of $d_1, \ldots, d_k$ are obtained as

$$\begin{aligned}
d_k^{-1} &= e_{k-1} \cdot (d_1 \cdot \ldots \cdot d_k)^{-1} \\
d_i^{-1} &= e_{i-1} \cdot (d_1 \cdot \ldots \cdot d_k)^{-1} \cdot d_k \cdot \ldots \cdot d_{i+1}, \quad i = k-1, \ldots, 2 \\
d_1^{-1} &= (d_1 \cdot \ldots \cdot d_k)^{-1} \cdot d_k \cdot \ldots \cdot d_2
\end{aligned}$$

Step 3. The third step recovers the inverses of the denominators $\delta_1^{-1}, \ldots, \delta_k^{-1}$ from $d_1^{-1}, \ldots, d_k^{-1}$ computed in Step 2. According to Step 1,

$$d_i = d_1^2 \cdot \ldots \cdot d_{i-1}^2 \cdot \delta_i \iff \delta_i^{-1} = d_1^2 \cdot \ldots \cdot d_{i-1}^2 \cdot d_i^{-1}$$

holds. Therefore, δ_i^{-1} can be recovered as

$$\delta_i^{-1} = e_{i-1}^2 \cdot d_i^{-1}, \quad i = 1, \ldots, k$$

using $e_1, \ldots, e_k$ computed in Step 2.

Step 4. The fourth step computes the points $[3]P, [5]P, \dots, [2k-1]P$, using the inverses of the denominators $\delta_1^{-1}, \dots, \delta_k^{-1}$ recovered in Step 3 and the formulas for point additions and doublings shown in Equation (11).

Theorem 1. *In total, the proposed scheme needs*

$$\mathrm{cp}_{\mathrm{Prop}} = (10k-11)M + (4k)S + I \tag{12}$$

to compute the points $[3]P, [5]P, \dots, [2k-1]P$*. Further, the proposed scheme requires* $2(k-1)$ *registers to store the points and no additional memory for temporary calculations.*

The proof of this theorem can be found in Appendix B.

5 Analysis

The proposed method as well as the methods reviewed in Section 3 trade field inversions for multiplications and squarings. Hence, the advantage of a respective method depends on the ratio of inversions and multiplications I/M and the ratio of squarings and multiplications S/M. In this analysis, the S/M ratio is set to $S = 0.8M$. For software implementations of an inversion in a prime field, the I/M ratios vary between $I = 4M$ [ELM03,BSS99] and $I = 80M$ [HMV04]. These ratios depend on many factors like the architecture, the methods used for multiplication, modular reduction, and inversion, and the size of the prime field. In software implementations, the inverse is usually computed using the binary GCD algorithm [HMV04]. However, this algorithm is hardly available in embedded devices like smart cards. On a smart card equipped with a cryptographic coprocessor it is faster to compute the inverse using Fermat's little theorem, i.e. $a^{-1} = a^{p-2} \bmod p$, since it uses only operations that are supported by hardware [CF05,ELM03,JP03]. When using Fermat's little theorem to compute an inversion in a prime field $\mathbb{F}_p$ the I/M ratio becomes very large, i.e. about $I = \log_2 p\; M$, since the inverse is computed using a modular exponentiation. According to [Sey05], I/M ratios of $I > 100M$ are realistic on smart cards equipped with a cryptographic coprocessor. In the following, the I/M break even points for the methods introduced in Section 3 and the proposed scheme are estimated.

***I/M* Break Even Points for the Precomputation** At first, the proposed scheme is compared to the last four methods introduced in Section 3. Note that all those methods require only one single inversion. If the S/M ratio $S = 0.8M$ is substituted in Equations (7)-(10) and (12) one gets

$$\begin{aligned}
\mathrm{cp}_{\mathcal{P}\to\mathcal{A}} &= (17k-10)M + (2k+3)S + I = (18.6k - 7.6)M + I\\
\mathrm{cp}_{\mathcal{J}\to\mathcal{A}} &= (18k-14)M + (5k+1)S + I = (22.0k - 13.2)M + I\\
\mathrm{cp}_{\mathcal{J}^c\to\mathcal{A}} &= (17k-12)M + (4k+2)S + I = (20.2k - 10.4)M + I\\
\mathrm{cp}_{\mathcal{J}^m\to\mathcal{A}} &= (19k-15)M + (7k-3)S + I = (24.6k - 17.4)M + I\\
\mathrm{cp}_{\mathrm{Prop}} &= (10k-11)M + (4k)S + I = (13.2k - 11.0)M + I
\end{aligned}$$

This shows that, regardless of the I/M ratio, the proposed method is more efficient than precomputing the points in a different coordinate system and converting them to $\mathcal{A}$ coordinates using the Montgomery trick.

The next step is to estimate the I/M break even points of the proposed scheme, the precomputation proposed in [CMO98], and the straightforward precomputation in $\mathcal{A}$ coordinates. A comparison with the straightforward precomputation in $\mathcal{J}^c$ coordinates will be done only for a complete scalar multiplication. This is because the computation of a scalar multiplication is more expensive if the precomputed points are represented in $\mathcal{J}^c$ coordinates (see Equations (1) and (2)). Table 1 shows for different k, for which I/M ratios the proposed scheme and the affine precomputation are the most efficient. The method proposed in [CMO98] is the fastest for the values in between.

k	2	3	4	5	6	7	8	9	10
Proposed	$\geqslant 9.0$	$\geqslant 9.7$	$\geqslant 9.9$	$\geqslant 10.0$	$\geqslant 10.5$	$\geqslant 12.9$	$\geqslant 15.4$	$\geqslant 12.5$	$\geqslant 14.4$
Affine	$\leqslant 9.0$	$\leqslant 9.7$	$\leqslant 9.9$	$\leqslant 10.0$	$\leqslant 9.6$	$\leqslant 7.4$	$\leqslant 6.3$	$\leqslant 8.0$	$\leqslant 7.0$
k	11	12	13	14	15	16	17	18	19
Proposed	$\geqslant 16.2$	$\geqslant 18.0$	$\geqslant 19.9$	$\geqslant 21.8$	$\geqslant 23.6$	$\geqslant 25.5$	$\geqslant 21.1$	$\geqslant 22.6$	$\geqslant 24.0$
Affine	$\leqslant 6.3$	$\leqslant 5.8$	$\leqslant 5.5$	$\leqslant 5.2$	$\leqslant 5.0$	$\leqslant 4.8$	$\leqslant 5.4$	$\leqslant 5.2$	$\leqslant 5.0$

Table 1. I/M break even points for the precomputation

For example if $k = 8$, the most efficient method is: the proposed method if $I/M \geq 15.4$, the [CMO98] method if $6.3 \leq I/M \leq 15.4$, and the affine method if $I/M \leq 6.3$. This table is visualized in Figure 1. Obviously, the advantage of one method is small if the I/M ratio is close to the break even point and large if the I/M ratio is far away from the break even point. Also, the I/M break even points shown in Table 1 are independent of the bit length of the scalar or the size of the prime field, whereas the actual I/M ratio on a certain platform is not. This comparison shows, that the affine and the [CMO98] method perform worse than the proposed method on devices with a large I/M ratio such as smart cards [Sey05]. See Appendix C for timings and a comparison of I/M ratios.

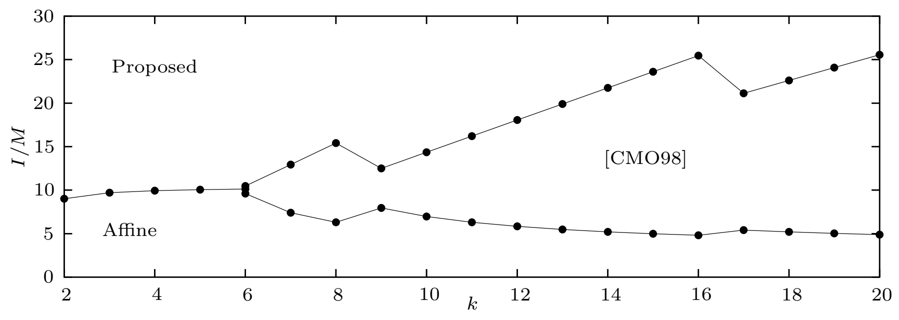

Fig. 1. I/M break even points

***I/M* Break Even Points for a Scalar Multiplication** In section 2 it was shown that a scalar multiplication requires three additional field multiplications for each point addition if the precomputed points are represented in $\mathcal{J}^c$ coordinates instead of $\mathcal{A}$ coordinates. In order to compare the proposed scheme with the straightforward precomputation in $\mathcal{J}^c$ coordinates (from now on called *$\mathcal{J}^c$ method*), the total costs for a scalar multiplication must be considered. In this case, the size of the prime field and the bit length n of the scalar is also important. It is assumed that the scalar is *recoded* using the fractional window recoding method and therefore has an AHD as shown in Equation (3). Using Equations (1),(2),(5), and (12) one obtains that the proposed method is more efficient than the $\mathcal{J}^c$ method if

$$I/M < 0.2k + 7.4 + 3n \cdot \mathrm{AHD}_k.$$

Table 2 shows the I/M break even points corresponding to a complete scalar multiplication for different prime fields $\mathbb{F}_{p_n}$, where p_n is an n bit prime. Smaller I/M ratios benefit the proposed method.

k	2	3	4	5	6	7	8	9	10
p_{192}	151.8	136.0	123.4	118.1	113.3	109.0	105.0	103.2	101.6
p_{224}	175.8	157.3	142.6	136.4	130.8	125.7	121.0	118.9	116.9
p_{256}	199.8	178.7	161.8	154.7	148.2	142.4	137.0	134.6	132.3

Table 2. I/M break even points for the proposed and $\mathcal{J}^c$ method

The I/M break even point gets smaller if k grows. However, the total costs for a scalar multiplication are minimal if $k = 8$. This can be determined by comparing the total costs of the proposed method ((1)+(12)) and the $\mathcal{J}^c$ method ((2)+(5)) for different k. The optimal value for k is independent from the I/M ratio, since the proposed method requires only one inversion regardless of k. Note, that such large I/M ratios as shown in Table 2 actually do occur, especially on smart cards where the field inversion is computed using Fermat's little theorem [CF05,ELM03,JP03,Sey05].

The above comparison has one flaw, it does not consider the memory requirement of the precomputed points. Note, that the $\mathcal{J}^c$ method requires 2.5 times the memory of the proposed method for the same k. This is due to the fact that a point in $\mathcal{J}^c$ coordinates consists of five coordinates, whereas a point in $\mathcal{A}$ coordinates consists of only two coordinates [CMO98]. Let r denote the maximum number of registers that can be used for the precomputed points. Then $k_p = \lfloor (r+2)/2 \rfloor$ and $k_c = \lfloor (r+5)/5 \rfloor$ denote the maximum value of k that can be used for the proposed method and the $\mathcal{J}^c$ method, respectively. For example, if $r = 15$ then $k_p = 8$ and $k_c = 4$. The proposed method with $k = 8$ needs $1861M + I$ and the $\mathcal{J}^c$ method with $k = 4$ needs $2008.4M$ for a scalar multiplication with a 192 bit scalar. This means, that the proposed method is more efficient as long as $I/M \leq 147.4$. Table 3 shows the I/M break even point corresponding to a complete scalar multiplication for different limitations on the number of registers r and different prime fields $\mathbb{F}_{p_n}$, where p_n is an n bit prime. Again, smaller I/M ratios benefit the proposed method.

r	5	6,7	8,9	10,11	12,13	14	15-19	20-24	25-29	30-34	≥ 35
k_p	3	4	5	6	7	8	8	8	8	8	8
k_c	2	2	2	3	3	3	4	5	6	7	8
p_{192}	202.6	240.6	249.3	189.5	194.5	198.0	147.4	133.4	121.8	112.5	105.0
p_{224}	237.3	283.8	296.2	226.3	234.4	240.7	179.4	160.8	145.1	131.9	121.0
p_{256}	271.9	327.0	343.1	263.2	274.3	283.3	211.4	188.2	168.4	151.4	137.0

Table 3. I/M break even points for fixed registers

If less than five registers are available, the only option is to use the proposed method. If more than 14 registers are available, the proposed method still uses $k = 8$ since using a larger value would decrease the total performance. The same argument holds for the $\mathcal{J}^c$ method if more than 35 registers are available. Table 3 shows, that including the number of registers in the comparison increases the I/M break even point of the proposed method and the $\mathcal{J}^c$ method compared to Table 2. The I/M break even points of the CMO method, the $\mathcal{A}$ method, and the proposed method shown in Table 1 still hold, since all three methods require the same number of registers for storing the precomputed points.

To summarize, the proposed method provides the most efficient precomputation for I/M ratios as they occur on smart cards [Sey05]. Another advantage of the proposed method is, that it precomputes the points in affine coordinates which require less storage space than $\mathcal{J}^c$ coordinates. If the memory for the precomputed points is limited, it is possible to choose larger values of k which further improves a scalar multiplication compared to the $\mathcal{J}^c$ method.

6 Conclusion

This paper presented a new method to precompute all odd points $[3]P, \ldots, [2k-1]P$, $k \geq 2$ on an elliptic curve defined over a prime field $\mathbb{F}_p$ in affine coordinates. The proposed method requires only one field inversion regardless of the number of points to precompute. In total, the proposed scheme requires $(10k-11)M + (4k)S + I$ field operations for the precomputation and no additional memory for temporary calculations. The proposed method is the most efficient for a large range of I/M ratios, especially for ratios as they occur on smart cards. Further research includes an implementation of the proposed scheme on a smart card.

References

[Ava04] Avanzi, R., *A Note on the Signed Sliding Window Integer Recoding and a Left-to-Right Analogue*, Selected Areas in Cryptography - SAC 2004, LNCS 3357, Springer, 2004, pp. 130-143.

[BSS99] Blake, I., Seroussi, G., and Smart, N., *Elliptic Curves in Cryptography*, London Mathematical Society, Lecture Note Series 265, Cambridge University Press, 1999.

[CF05] Cohen, H., Frey, G., *Handbook of elliptic and hyperelliptic curve cryptography*, CRC Press, 2005.

[CJLM06] Ciet, M., Joye, M., Lauter, K., Montgomery, P., *Trading Inversions for Multiplications in Elliptic Curve Cryptography*, Designs, Codes and Cryptography, Volume 39, Issue 2, 2006, pp. 189-206.

[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.

[DH76] Diffie, W., and Hellman, M., *New directions in cryptography*, IEEE Transactions on Information Theory, vol. IT-22, no. 6, 1976, pp. 644-654.

[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.

[Flexi] FlexiProvider, available at `http://www.flexiprovider.de/`.

[HMV04] Hankerson, D., Menezes, A., Vanstone, S., *Guide to Elliptic Curve Cryptography*, Springer, 2004.

[Infineon] Infineon Technologies, `http://www.infineon.com/`.

[Java] The Source for Java Developers, `http://java.sun.com/`.

[JM99] Johnson, D., and Menezes, A., *The Elliptic Curve Digital Signature Algorithm (ECDSA)* University of Waterloo, Technical Report CORR 99-34, 1999, available at `http://www.cacr.math.uwaterloo.ca`.

[JP03] Joye, P., Paillier, P., *GCD-Free Algorithms for Computing Modular Inverses*, Cryptographic Hardware and Embedded Systems - CHES 2003, LNCS 2779, Springer, 2003, pp. 243-253.

[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.

[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.

[NIST01] Daley, W., Kammerer, R., *Digital Signature Standard (DSS)*, National Institute of Standards and Technology - NIST, Federal Information Processing Standards - FIPS 186-2, 2001, available at `http://csrc.nist.gov/publications/fips/index.html`.

[OSST04] Okeya, K., Schmidt-Samoa, K., Spahn, C., Takagi, T., *Signed Binary Representations Revisited*, Advances in Cryptology - CRYPTO 2004, LNCS 3152, Springer, 2004, pp. 123-139.

[Renesas] Renesas Technologies, `http://www.renesas.com/homepage.jsp/`.

[Sey05] Seysen, M., *Using an RSA Accelerator for Modular Inversion*, Cryptographic Hardware and Embedded Systems CHES 2005, LNCS 3659, Springer, 2005, pp. 226-236.

[Sol00] Solinas, J.A., *Efficient Arithmetic on Koblitz Curves*, Design, Codes and Cryptography, vol. 19, 2000, pp. 195-249.

[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.

A Pseudocode of the Proposed Scheme

This section contains the pseudocode of the four steps of the proposed scheme.

Algorithm 2 Step 1: Computation of $d_1, \ldots, d_k$

```
Require: $P = (x_1, y_1), k, a$
Ensure: $d_1, \ldots, d_k$
 1: $d_1 \leftarrow 2y_1$
 2: $C \leftarrow d_1^2$
 3: $A \leftarrow 3x_1^2 + a$
 4: $B \leftarrow C \cdot 3x_1$
 5: $d_2 \leftarrow A^2 - B$
 6: $E \leftarrow d_2^2$
 7: $B \leftarrow E \cdot B$
 8: $C \leftarrow C^2$
 9: $D \leftarrow E \cdot d_2$
10: $A \leftarrow -d_2 \cdot A - C$
11: $d_3 \leftarrow A^2 - 2D - B$
12: $E \leftarrow d_3^2$
13: $B \leftarrow E\,(B + 3D)$
14: $C \leftarrow D\,(2A + C)$
15: $D \leftarrow E \cdot d_3$
16: $A \leftarrow -d_3 \cdot A - C$
17: $d_4 \leftarrow A^2 - D - B$
18: for $i = 5$ to $k$ do
19:     $E \leftarrow d_{i-1}^2$
20:     $B \leftarrow E \cdot B$
21:     $C \leftarrow D \cdot C$
22:     $D \leftarrow E \cdot d_{i-1}$
23:     $A \leftarrow -d_{i-1} \cdot A - C$
24:     $d_i \leftarrow A^2 - D - B$
25: end for
26: return $d_1, \ldots, d_k$.
```

Algorithm 3 Step 2: Simultaneous inversion of $d_1, \dots, d_k$

```
Require: $d_i$, $i = 1, \dots, k$
Ensure: $f_i = d_i^{-1}, e_i = \prod_{j=1}^{i} d_i$, $i = 1, \dots, k$
 1: $e_1 \leftarrow d_1$
 2: for $i = 2$ to $k$ do
 3:     $e_i \leftarrow e_{i-1} \cdot d_i$
 4: end for
 5: $T_1 \leftarrow e_k^{-1}$
 6: for $i = k$ down to 2 do
 7:     $T_2 \leftarrow d_i$
 8:     $f_i \leftarrow e_{i-1} \cdot T_1$
 9:     $T_1 \leftarrow T_1 \cdot T_2$
10: end for
11: $f_1 \leftarrow T_1$
12: return $e_1, \dots, e_k, f_1, \dots, f_k$
```

Algorithm 4 Step 3: Retrieval of the inverses of the $\delta_1, \dots, \delta_k$

```
Require: $f_i$ and $e_i$, $i = 1, \dots, k$
Ensure: Inverse of denominators $l_i = \delta_i^{-1}$, $i = 1, \dots, k$
 1: $l_1 \leftarrow f_1$
 2: for $i = 2$ to $k$ do
 3:     $l_i \leftarrow e_{i-1}^2 \cdot f_i$
 4: end for
 5: return $l_1, \dots, l_k$
```

Algorithm 5 Step 4: Computation of the required points

```
Require: $P = (x_1, y_1), k, a$ and $l_i$, $i = 1, \dots, k$
Ensure: $3P = (x_3, y_3), 5P = (x_4, y_4), \dots, (2k-1)P = (x_{k+1}, y_{k+1})$
 1: $T \leftarrow (3x_1^2 + a) \cdot l_1$
 2: $x_2 \leftarrow T^2 - 2x_1$
 3: $y_2 \leftarrow T(x_1 - x_2) - y_1$
 4: $T \leftarrow (y_2 - y_1) \cdot l_2$
 5: $x_3 \leftarrow T^2 - x_2 - x_1$
 6: $y_3 \leftarrow T(x_2 - x_3) - y_2$
 7: for $i = 3$ to $k$ do
 8:     $T \leftarrow (y_i - y_2) \cdot l_i$
 9:     $x_{i+1} \leftarrow T^2 - x_2 - x_i$
10:     $y_{i+1} \leftarrow T(x_2 - x_{i+1}) - y_2$
11: end for
12: return $x_3, \dots, x_{k+1}, y_3, \dots, y_{k+1}$
```

B Proof of Theorem 1.

This section states the proof of the Theorem 1 of Section 4.

Theorem 1. *In total, the proposed scheme requires*

$$(10k-11)M+(4k)S+I$$

field operations to compute the points $3P, 5P, \ldots, (2k-1)P$*. Further, the proposed scheme requires* $2(k-1)$ *registers to store the points and no additional memory for temporary calculations.*

Proof. The costs of each algorithm are calculated separately and summed up. Additions and multiplications with small numbers are neglected since they can be computed very fast. Algorithm 2 requires $8M+8S+(k-4)(4M+2S)=(4k-8)M+(2k)S$ to compute the d_i. Algorithm 3 requires $3(k-1)M+I$ to invert the d_i and compute the e_i. Algorithm 4 requires $(k-1)(S+M)=(k-1)M+(k-1)S$ to recover the l_i. Algorithm 5 requires $(4M+3S)+(k-2)(2M+S)=(2k)M+(k+1)S$ to compute the points $[3]P, [5]P, \ldots, [2k-1]P$. The sum of the costs of all four steps is given as $(10k-11)M+(4k)S+I$.

To store the points $[3]P, [5]P, \ldots, [2k-1]P$, $2(k-1)$ registers are required. Note, that since the double-and-add algorithm stores the intermediate results in modified Jacobian coordinates, which are represented using four coordinates, 4 additional registers are required for the evaluation of a scalar multiplication. Hence, $2k+2$ registers are available in total. Algorithm 2 requires $k+5$ registers to hold d_i and the temporary variables A, B, C, D, E. Algorithm 3 requires $2k+2$ registers to hold e_i, f_i and the temporary variables T_1, T_2. The f_i can use the same registers as the d_i which explains the necessity of line 7. Algorithm 4 requires k registers to hold l_i. The l_i can use the same registers as the f_i. Algorithm 5 requires $2k+1$ registers to hold x_i, y_i and one temporary variable T. The x_i and y_i can use the same registers as the e_i and l_i. In total, $2k+2$ registers are required and therefore no additional memory has to be allocated.

$\square$

C Timings

This Section states timings for the precomputation using Java implementations of the proposed scheme, the [CMO98] method and the straightforward affine method. The base points P are randomly chosen points on the elliptic curves P-192, P-224 and P-256 recommended in [NIST01], which are defined over an 192, 224 and 256 bit prime field $\mathbb{F}_p$, respectively. The timings are CPU time in microseconds and were obtained on two machines using three versions of Java [Java]. Table 4 shows the timings on a Pentium Dualcore 1.83Ghz. Table 5 shows the timings on an AMD Athlon 64 X2 Dualcore 4200+ 2.22GHZ. Both machines have 1GB RAM and use Windows XP. The fastest method is marked bold. For each combination of Java version and prime field, the timings for field multiplications and field inversions as well as the resulting I/M ratio are stated.

k	2	3	4	5	6	7	**8**	9	10
192 bit prime field									
Java v1.4.2_12, $M = 3.8$, $I = 59.9$, $I/M = 15.9$									
Affine	158.9	238.6	317.8	396.0	476.3	552.4	632.2	709.7	789.5
[CMO98]	159.4	284.7	299.4	446.9	462.0	476.0	**491.9**	688.8	702.6
Proposed	**132.5**	**192.8**	**255.6**	**316.8**	**376.7**	**438.3**	496.6	**557.2**	**617.0**
Java v1.5.0_10, $M = 3.6$, $I = 59.0$, $I/M = 16.6$									
Affine	156.0	232.9	311.9	390.0	467.6	545.7	622.5	699.2	778.7
[CMO98]	156.3	278.3	291.7	434.3	450.0	459.9	475.4	666.8	679.8
Proposed	**128.4**	**186.1**	**245.5**	**302.0**	**358.7**	**415.1**	**473.2**	**529.7**	**586.9**
Java v1.6.0, $M = 2.9$, $I = 52.0$, $I/M = 17.8$									
Affine	135.0	203.9	272.7	341.7	408.7	476.5	544.5	614.1	680.6
[CMO98]	136.5	238.7	250.5	372.4	384.8	397.0	407.5	566.0	580.2
Proposed	**105.8**	**154.8**	**203.4**	**250.5**	**298.0**	**342.1**	**390.3**	**437.0**	**483.4**
224 bit prime field									
Java v1.4.2_12, $M = 4.7$, $I = 73.1$, $I/M = 15.7$									
Affine	198.9	288.2	382.3	475.9	573.6	668.3	762.6	859.1	953.6
[CMO98]	199.9	340.1	359.2	534.3	553.5	572.0	**588.3**	822.5	842.3
Proposed	**165.2**	**233.0**	**307.0**	**380.3**	**450.7**	**522.5**	596.4	**667.9**	**738.3**
Java v1.5.0_10, $M = 4.4$, $I = 72.4$, $I/M = 16.5$									
Affine	194.5	285.0	377.1	471.6	568.0	659.7	754.0	848.3	942.1
[CMO98]	196.6	334.5	351.2	523.7	540.5	556.5	571.2	798.6	819.3
Proposed	**160.7**	**224.3**	**295.8**	**364.4**	**431.3**	**502.1**	**569.8**	**640.8**	**704.4**
Java v1.6.0, $M = 3.6$, $I = 63.9$, $I/M = 17.7$									
Affine	173.1	249.2	332.5	414.6	497.6	581.8	665.9	748.3	831.8
[CMO98]	171.4	292.4	307.0	455.6	470.2	483.7	500.5	693.4	707.6
Proposed	**137.5**	**191.7**	**250.8**	**309.5**	**365.9**	**423.7**	**482.3**	**539.4**	**598.8**
256 bit prime field									
Java v1.4.2_12, $M = 5.7$, $I = 88.7$, $I/M = 15.4$									
Affine	240.7	347.4	461.7	574.6	687.9	802.7	917.2	1039.6	1146.2
[CMO98]	239.0	407.4	430.3	643.6	667.3	687.3	**707.8**	999.3	1015.1
Proposed	**200.2**	**281.1**	**369.6**	**457.4**	**543.9**	**631.7**	718.6	**807.2**	**893.9**
Java v1.5.0_10, $M = 5.4$, $I = 88.2$, $I/M = 16.3$									
Affine	237.4	344.0	457.6	569.3	684.3	796.4	908.6	1021.2	1135.8
[CMO98]	238.0	404.0	425.1	633.2	654.0	674.5	**692.3**	969.2	990.9
Proposed	**196.4**	**273.6**	**358.0**	**443.3**	**527.2**	**610.3**	693.5	**779.2**	**859.7**
Java v1.6.0, $M = 4.5$, $I = 78.4$, $I/M = 17.5$									
Affine	210.6	303.3	403.6	505.1	605.7	705.1	807.0	907.0	1006.4
[CMO98]	208.0	353.5	371.4	551.6	569.9	586.0	603.7	840.0	857.2
Proposed	**168.4**	**233.3**	**304.4**	**374.8**	**443.6**	**511.0**	**579.3**	**650.4**	**722.2**

Table 4. Timings in microseconds on a Pentium dualcore 1.83 GHz

k	2	3	4	5	6	7	**8**	9	10
192 bit prime field									
Java v1.4.2_12, $M = 3.2$, $I = 53.7$, $I/M = 17.0$									
Affine	142.0	212.2	283.5	353.6	423.3	490.0	560.5	627.3	700.7
[CMO98]	144.2	250.9	264.1	391.6	406.7	418.7	429.6	597.3	611.1
Proposed	**115.6**	**169.5**	**222.4**	**273.5**	**324.1**	**375.6**	**428.0**	**481.0**	**533.3**
Java v1.5.0_10, $M = 2.9$, $I = 53.3$, $I/M = 18.1$									
Affine	140.5	208.2	275.6	343.7	411.0	480.0	549.3	616.0	688.0
[CMO98]	140.3	245.4	255.0	375.0	389.5	403.0	414.8	572.5	590.0
Proposed	**111.6**	**161.7**	**207.9**	**255.0**	**304.8**	**353.9**	**403.1**	**451.1**	**500.0**
Java v1.6.0, $M = 2.4$, $I = 44.7$, $I/M = 18.5$									
Affine	113.1	170.4	225.1	279.4	337.1	392.0	444.8	498.6	552.2
[CMO98]	113.3	197.5	207.9	309.6	318.3	329.1	338.4	469.6	475.3
Proposed	**90.9**	**132.0**	**171.6**	**213.3**	**250.0**	**289.3**	**329.6**	**366.6**	**400.8**
224 bit prime field									
Java v1.4.2_12, $M = 4.1$, $I = 66.0$, $I/M = 16.1$									
Affine	178.4	256.6	341.8	426.0	503.4	585.7	669.6	751.8	834.5
[CMO98]	178.2	305.1	318.0	477.1	482.8	500.9	**511.8**	715.9	733.9
Proposed	**149.2**	**207.2**	**274.5**	**338.3**	**390.6**	**452.1**	515.1	**578.1**	**640.7**
Java v1.5.0_10, $M = 3.7$, $I = 65.3$, $I/M = 17.8$									
Affine	174.7	255.1	336.3	420.2	503.0	588.2	675.6	756.8	833.4
[CMO98]	176.2	295.9	311.0	460.3	476.9	493.0	512.9	704.8	714.6
Proposed	**141.7**	**195.6**	**254.8**	**312.5**	**374.4**	**434.4**	**495.5**	**554.0**	**608.5**
Java v1.6.0, $M = 3.0$, $I = 54.7$, $I/M = 18.2$									
Affine	143.0	207.5	275.7	340.7	402.9	467.1	532.8	599.6	665.0
[CMO98]	140.9	239.7	251.6	372.6	380.1	389.2	399.2	552.4	564.8
Proposed	**114.9**	**158.5**	**205.9**	**253.9**	**295.2**	**337.7**	**384.3**	**431.9**	**476.5**
256 bit prime field									
Java v1.4.2_12, $M = 4.9$, $I = 79.0$, $I/M = 16.3$									
Affine	209.3	302.8	401.4	499.8	599.6	698.7	795.8	895.3	999.2
[CMO98]	208.9	353.7	372.4	554.1	573.1	589.9	609.2	849.2	870.8
Proposed	**174.5**	**242.9**	**315.1**	**388.7**	**461.8**	**535.2**	**609.1**	**683.6**	**760.4**
Java v1.5.0_10, $M = 4.5$, $I = 79.0$, $I/M = 17.4$									
Affine	210.3	305.7	403.0	502.3	599.6	696.7	807.1	904.2	1003.0
[CMO98]	212.3	353.2	372.2	551.5	566.2	580.1	610.2	848.1	866.9
Proposed	**170.3**	**236.7**	**312.7**	**381.8**	**447.3**	**518.8**	**599.7**	**674.6**	**745.8**
Java v1.6.0, $M = 3.7$, $I = 66.2$, $I/M = 17.8$									
Affine	172.9	247.0	325.9	407.0	487.8	567.3	650.0	733.8	812.8
[CMO98]	172.0	286.7	301.2	446.1	461.6	475.9	493.1	685.6	699.2
Proposed	**136.9**	**190.1**	**248.5**	**304.3**	**361.1**	**417.4**	**476.6**	**533.0**	**590.8**

Table 5. Timings in microseconds on a AMD Athlon 4200+ 2.2GHz

The implementations of the three methods will eventually be published as part of the FlexiECProvider [Flexi], an open source Java Cryptographic Service Provider. The FlexiECProvider computes the field multiplications and inversions using the `java.math.BigInteger` functions `a.multiply(b).mod(p)` and `a.modInverse(p)`, respectively, where p is the n bit prime that defines the prime field $\mathbb{F}_p$ and a, b are n bit integers.

The above timings show that the proposed method has a noticeable advantage compared to the [CMO98] method and the affine method, even for small I/M ratios. They also confirm the I/M break even points estimated in Section 5, Table 1 and show that the proposed method has only a small overhead. Further, Tables 4 and 5 indicate that the I/M ratio constantly increases if the field multiplication and inversion get faster. This is true for the different Java versions, where v1.6.0 has the largest I/M ratio and the fastest field multiplications and inversions, as well as for the faster AMD CPU. The trend definitely goes towards larger I/M ratios which further benefits the proposed method.