

# An Advanced Method for Joint Scalar Multiplications on Memory Constraint Devices

Erik Dahmen,<sup>1</sup> Katsuyuki Okeya,<sup>2</sup> and Tsuyoshi Takagi<sup>3</sup>

<sup>1</sup> Technische Universität Darmstadt, Fachbereich Informatik,  
Hochschulstr.10, D-64289 Darmstadt, Germany  
`dahmen@rbg.informatik.tu-darmstadt.de`

<sup>2</sup> Hitachi, Ltd., Systems Development Laboratory,  
1099, Ohzenji, Asao-ku, Kawasaki-shi, Kanagawa-ken, 215-0013, Japan  
`ka-okeya@sdl.hitachi.co.jp`

<sup>3</sup> Future University - Hakodate,  
116-2 Kamedanakano-cho Hakodate Hokkaido, 041-8655, Japan  
`takagi@fun.ac.jp`

**Abstract.** One of the most frequent operations in modern cryptosystems is a multi-scalar multiplication with two scalars. Common methods to compute it are the Shamir method and the Interleave method whereas their speed mainly depends on the (joint) Hamming weight of the scalars. To increase the speed, the scalars are usually deployed using some general representation which provides a lower (joint) Hamming weight than the binary representation. However, by using such general representations the precomputation and storing of some points becomes necessary and therefore more memory is required. Probably the most famous method to speed up the Shamir method is the joint sparse form (JSF). The resulting representation has an average joint Hamming weight of  $1/2$  and it uses the digits  $0, \pm 1$ . To compute a multi-scalar multiplication with the JSF, the precomputation of two points is required. While for two precomputed points both the Shamir and the Interleave method provide the same efficiency, until now the Interleave method is faster in any case where more points are precomputed. This paper extends the used digits of the JSF in a natural way, namely we use the digits  $0, \pm 1, \pm 3$  which results in the necessity to precompute ten points. We will prove that using the proposed scheme, the average joint Hamming density is reduced to  $239/661 \approx 0.3615$ . Hence, a multi-scalar multiplication can be computed more than 10% faster, compared to the JSF. Further, our scheme is superior to all known methods using ten precomputed points and is therefore the first method to improve the Shamir method such that it is faster than the Interleave method. Another advantage of the new representation is, that it is generated starting at the most significant bit. More specific, we need to store only up to 5 joint bits of the new representation at a time. Compared to representations which are generated starting at the least significant bit, where we have to store the whole representation, this yields a significant saving of memory.

**Keywords:** *elliptic curve cryptosystem, joint sparse form, left-to-right, multi-scalar multiplication, shamir method*

## 1 Introduction

In our modern society it becomes more and more necessary to communicate and authenticate electronically in a secure way. Because of their mobility and tamper resistance, smart cards are often used for this task. However, since smart cards are quite small, the available memory and computational power is very limited. The Elliptic Curve Cryptosystem (ECC) [Kob87,Mil86] is an efficient cryptosystem, which can attain high security with very short key length. Therefore, the ECC is suitable for implementation on devices where computational power is limited. The basic operation for verifying a signature with the ECC is a so-called multi-scalar multiplication

$$uP + vQ$$

for given scalars  $u, v$  and points on the elliptic curve  $P, Q$ . The research goal from practical requirement is to efficiently compute this multi-scalar multiplication by minimizing both memory usage and computational costs [Ava02,Gor98,Möl01]. Another example for resource constraint devices where the ECC can be implemented are sensors. Those devices are mainly used to monitor a given physical environment. Different from smart cards, sensors have no external power source, therefore efficiency is not only required to save time and resources, but also to save energy.

Two efficient methods to compute a multi-scalar multiplication are the Shamir method [ElG85] and the Interleave method [Möl01]. The speed of those methods depends on the (joint) Hamming weight of the scalars. If some memory for precomputation is available they can be sped up by deploying a redundant representation of the scalars. Those representations provide a smaller (joint) Hamming weight than the binary representation, but use a larger digit set. The downside is, that the size of the digit set determines the number of points to precompute, and thus a trade-off between memory usage and speed has to be made.

Probably the most popular representation to speed up the Shamir method is the Joint Sparse Form (JSF) proposed by Solinas [Sol01]. The used digits in this representation are  $0, \pm 1$  and it requires the precomputation of two points. The resulting average joint Hamming density is  $1/2$ . While for two precomputed points, the Shamir method and the Interleave method can provide the same efficiency, until now the Interleave method is faster in any case where more points are precomputed.

In this paper we propose a new representation to further speed up the Shamir method. This is achieved by naturally extending the digit set of the JSF and allowing the digits  $0, \pm 1, \pm 3$ . For those digits, the precomputation of ten points is required. The main idea of the algorithm is to apply a sliding window method with variable width on both scalars. The widths are chosen such that the resulting density of non-zero columns increases from step to step. Also, if certain conditions are satisfied we reuse already converted columns in the proceeding step. We will prove that the average joint Hamming density of the proposed

scheme is  $239/661 \approx 0.3615$ , which is superior to any known method using ten precomputed points. Therefore, our method is the first to improve the Shamir method such that it is faster than the Interleave method. Compared to the JSF, the computation of  $uP + vQ$  can be sped up by more than 10%. Another advantage of the proposed scheme is, that it is generated starting at the most significant bit, which is more natural and memory saving in conjunction with the ECC (see Section 2). More specific, we need to scan only up to 6 joint bits of the binary representations of the scalars at once.

The rest of the paper is organized as follows: In Section 2 we give an overview of multi-scalar multiplication. In Section 3 we review several known methods to speed up the computation of  $uP + vQ$ . In Section 4 the proposed scheme is described and its computational cost is calculated. In Section 5 we compare our scheme to the methods of Section 3 and Section 6 states our conclusion.

## 2 Preliminaries

### 2.1 Notations

A scalar  $d$  is a positive integer and there are several ways to represent it. The most common one is the uniquely determined binary representation  $(d_{n-1}, \dots, d_0)$ , where  $d = \sum_{i=0}^{n-1} d_i \cdot 2^i$  and  $d_i \in \{0, 1\}, \forall i = 0, \dots, n-1$ . Here,  $n$  is the bit length of the representation. Another way is a more general approach. Now we don't restrict the digits to the set  $\{0, 1\}$  but to an arbitrary digit set  $\mathcal{D}$ . We call  $(d_{n-1}, \dots, d_0)$  a  $\mathcal{D}$ -representation of  $d$ , if  $d = \sum_{i=0}^{n-1} d_i \cdot 2^i$  and  $d_i \in \mathcal{D}, \forall i = 0, \dots, n-1$ . For example, if  $\mathcal{D} = \{0, \pm 1\}$  we call the underlying  $\mathcal{D}$ -representation a signed binary representation. In general,  $\mathcal{D}$ -representations loose the property of uniqueness.

The Hamming weight (HW) of a  $\mathcal{D}$ -representation is its number of non-zero entries. The Hamming density (HD) is defined as  $\text{HW}/n$ . The average Hamming density (AHD) is the expected HD for a random representation with bit length  $n \rightarrow \infty$ . If we consider more than one  $\mathcal{D}$ -representation simultaneously, we may want to examine non-zero columns rather than non-zero entries. The number of non-zero columns of an arbitrary number of  $\mathcal{D}$ -representations is given by the joint Hamming weight (JHW). The joint Hamming density (JHD) and the average joint Hamming density (AJHD) are defined in accordance to the HD and the AHD, respectively. For simplicity, we consider only representations with the same bit length  $n$ . This can be achieved by padding with zeros to the left.

Let  $K = GF(p)$  be a finite field, where  $p > 3$  is a prime. Let  $E$  be an elliptic curve over  $K$ . The elliptic curve  $E$  has an abelian group structure with identity element  $\mathcal{O}$  called the point of infinity. A point  $P \in E$  is represented as  $P = (x, y)$ . The inverse of point  $P = (x, y)$  is equal to  $-P = (x, -y)$ , hence it can be computed virtually free. For that reason, it is advisable to use a signed binary representation of the scalars [MO90]. Note that this is also true for elliptic curves over different fields, e.g. binary curves. The elliptic curve operations  $P + Q$  and  $2P$  are denoted by ECADD and ECDBL, respectively, where  $P, Q \in E$ .

## 2.2 Multi-Scalar Multiplication Algorithms

In this section we explain how the Shamir method and the Interleave method compute  $uP + vQ$ . This is done in the so-called *evaluation stage*: at first an accumulator  $X$  is initialized with the neutral group element  $\mathcal{O}$ , then the following steps are performed.

Shamir Method	Interleave Method
<pre> <b>for</b> <math>i = n - 1</math> <b>down to</b> <math>0</math> <b>do</b>   <math>X \leftarrow ECDBL(X)</math>   <b>if</b> <math>(u_i, v_i) \neq (0, 0)</math> <b>then</b>     <math>X \leftarrow ECADD(X, u_iP + v_iQ)</math> </pre>	<pre> <b>for</b> <math>i = n - 1</math> <b>down to</b> <math>0</math> <b>do</b>   <math>X \leftarrow ECDBL(X)</math>   <b>if</b> <math>u_i \neq 0</math> <b>then</b>     <math>X \leftarrow ECADD(X, u_iP)</math>   <b>if</b> <math>v_i \neq 0</math> <b>then</b>     <math>X \leftarrow ECADD(X, v_iQ)</math> </pre>

After the last iteration  $X$  contains the result  $uP + vQ$  and is returned. Because both methods frequently use points of the form  $v_iQ$ ,  $u_iP$  and  $u_iP + v_iQ$  it is preferable to precompute and store those points. This is done in the *precomputation stage* which is executed prior to the evaluation stage. Note, that since point inversions can be performed online, we don't have to precompute all required points. Typically one uses a symmetric digit set of the form  $\mathcal{D} = \{0, \pm 1, \dots, \pm x\}$ . In that case only half of all used points have to be precomputed. The Interleave method computes  $t_1P, \forall t_1 \in \mathcal{D}_1 : t_1 > 1$  and  $t_2Q, \forall t_2 \in \mathcal{D}_2 : t_2 > 1$ , where  $\mathcal{D}_1$  and  $\mathcal{D}_2$  are the digit sets of the scalars  $u$  and  $v$ , respectively. The Shamir method computes the points  $tP, tQ, \forall t \in \mathcal{D} : t > 1$  and  $t_1P + t_2Q, \forall t_1, t_2 \in \mathcal{D} : t_1 > 0$ , where  $\mathcal{D}$  is the digit set of both scalars. Hence, the total number of precomputed points is  $(|\mathcal{D}_1| - 3)/2 + (|\mathcal{D}_2| - 3)/2$  and  $(|\mathcal{D}| - 1)^2/2 + |\mathcal{D}| - 3$  for the Interleave method and the Shamir method, respectively.

The average speed of both methods is determined by the number of ECDBL and ECADD operations used. The ECDBL operation is performed in each iteration in both methods, i.e.  $n$  times. The Shamir method performs an ECADD operation every time a non-zero column is found, therefore the average number of ECADD operations equals  $n$  times the AJHD of the scalars. The Interleave method performs an ECADD operation every time a non-zero entry is found in any of the scalars, therefore the average number of ECADD operations equals  $n$  times the sum of the AHD of the scalars.

## 2.3 Left-to-Right vs. Right-to-Left

Now we explain why it is preferable to perform the evaluation starting at the most significant bit, i.e. left-to-right (LtR), rather than the least significant bit, i.e. right-to-left (RtL). Although both the Shamir method and the Interleave method use a LtR evaluation stage, there also exist methods which use a RtL evaluation stage. The main drawback of those methods is that they are very inefficient when used with general  $\mathcal{D}$ -representations. Namely, in each iteration, they have to perform one ECDBL operation for all points which might be required in the ECADD step. Those points are all precomputed points plus the

base points  $P$  and  $Q$ . On the other hand, the LtR methods always use the same, fixed points for the ECADD step. Therefore, it is possible to speed up this step significantly if those points are represented in affine coordinates [CMO98].

## 2.4 A Special Signed Binary Representation

Now we introduce a special signed binary representation which is required to generate our proposed representation. This signed binary representation was proposed independently by two parties and is called the "alternating greedy expansion" [GHPT03] or the "mutual opposite form" [OSST04]. Let  $(d_{n-1}, \dots, d_0)$  be the binary representation of an integer  $d$ . We define  $\mu_i = d_{i-1} - d_i$  for  $i = 0, \dots, n$ , where  $d_n = d_{-1} = 0$ . Since

$$(\mu_n, \dots, \mu_0) = (d_{n-1}, \dots, d_0, 0) - (0, d_{n-1}, \dots, d_0) = 2d - d = d$$

this operation indeed yields a signed representation of  $d$ . Note that this representation, from now on called MOF, can be obtained from LtR and from RtL likewise. The MOF of a non-zero scalar satisfies the following properties

1. The signs of adjacent non-zero bits (without considering zero bits) are opposite.
2. The most non-zero bit and the least non-zero bit are 1 and  $\bar{1}$ , respectively.

Further, MOF uses the digit set  $\mathcal{D} = \{0, \pm 1\}$  and provides a AHD of  $1/2$ . Also it has been proven that each  $n$ -bit integer has a unique representation as  $(n+1)$ -bit MOF.

## 3 The Shamir Method vs. the Interleave Method

As we saw in Section 2.2, the number of ECADD operations of the Shamir method and the Interleave method depends on the JHW and the HW of the scalars, respectively. Hence, in order to speed up those methods these numbers have to be decreased. This is achieved by applying a so-called *recoding algorithm* which rewrites the binary representation of the scalars into some  $\mathcal{D}$ -representation. There are two kinds of recoding algorithms: those which decrease the HW and therefore speed up the Interleave method and those which decrease the JHW and therefore speed up the Shamir method. Also, the direction in which the scalars are recoded is important. In the case of a RtL recoding algorithm the scalars must be recoded in a separate stage prior to the precomputation stage, because we use a LtR evaluation stage. Then it is necessary to store the whole recoded scalars, which requires  $O(n)$  bits memory for each scalar. In the case of a LtR recoding algorithm the recoding can be performed "on-the-fly" during the evaluation stage. The advantage is obvious, now we don't have to store the whole recoded scalars, but only a small part which leads to a significant memory saving.

In Section 2.2 we also saw that the size of the digit set determines the number of points to precompute. However, the size of the digit set also affects the AHD

or AJHD of the  $\mathcal{D}$ -representation produced by a recoding algorithm, but in a non-proportional way. Therefore we face a trade-off between memory usage for the precomputed points and speed for the multi-scalar multiplication.

This section serves two purposes. At first we review several known recoding methods and explain how they speed up the computation of  $uP+vQ$ . Second, we explain why the optimal choice for the Shamir method is to use representations which require ten precomputed points. Note, that all representations reviewed in this section are uniquely determined and at most one bit longer than the corresponding binary representation (see the respective reference).

**Known methods using two precomputed points** The most common recoding algorithm to decrease the JHW is the joint sparse form (JSF) proposed by Solinas [Sol01]. Its AJHD is  $1/2$  and it uses the digit set  $\mathcal{D} = \{0, \pm 1\}$ . The drawback of the JSF is, that it can only be generated from RtL. A similar method was proposed in [HKPR04]. It uses the same digit set and provides the same AJHD as the JSF, but it can be applied from LtR. The main idea of this algorithm is to apply a LtR sliding window method with different widths on the MOF of both scalars. At first the width  $w = 2$  is tested and if no zero column can be generated the width is increased to  $w = 3$ . The precomputed points for those two methods are  $\{P + Q, P - Q\}$ .

To decrease the HW of the scalars, there also exists RtL and LtR variants. One RtL method is the famous width- $w$  non adjacent form ( $w$ NAF) [Sol00,BSS99,MOC97]. It uses the digit set  $\mathcal{D} = \{0, \pm 1, \dots, \pm 2^{w-1} - 1\}$  and its AHD is  $1/(w + 1)$ . Its LtR equivalent is called the width- $w$  mutual opposite form ( $w$ MOF) and is generated by applying a width- $w$  sliding window method from LtR on the MOF each scalar separately [OSST04]. Another LtR method, which is directly applied on the binary representation was proposed in [Ava04]. Both these methods also use the same digit set and provide the same AHD as  $w$ NAF. In the case of two precomputed points we choose  $w = 3$  and precompute  $\{3P, 3Q\}$ . The resulting AHD of each scalar then is  $1/4$ . Therefore, the average density of ECADD operations used by the Interleave method is  $1/2$ , the same as for the Shamir method using the JSF or the scheme described in [HKPR04].

However, two precomputed points require only 80 bytes of memory and since the current smart card technology offers several kbytes of memory, a lot of memory is wasted. If we want to use more memory, the logical step is to extend the digit set. And if we want to preserve the nice properties of signed representations in conjunction with the ECC, the natural extension of the digit set for the Shamir method is  $\mathcal{D} = \{0, \pm 1, \pm 3\}$ , which requires the precomputation of 10 points. To store those points, 400 bytes of memory are required.

**Known methods using ten precomputed points** The first known method using 10 precomputed points is an extension of the algorithm to create the JSF, called JSF<sub>3</sub> which was proposed by Kuang, Zhu and Zhang [KZZ04]. They use the digit set  $\mathcal{D} = \{0, \pm 1, \pm 3\}$  and the resulting AJHD is  $121/326 \approx 0.3712$ . This method requires the precomputation of  $\{3P, 3Q, P + Q, P - Q, P + 3Q, P - 3Q, 3P + Q, 3P - Q, 3P + 3Q, 3P - 3Q\}$ . Another method was proposed by Avanzi [Ava02]. He lets a width-2 window method slide from LtR over the JSF

of two scalars to increase the number of zero columns. His method uses the digit set  $\mathcal{D} = \{0, \pm 1, \pm 2, \pm 3\}$ , but because of the properties of the JSF only the points  $\{P+Q, P-Q, P+2Q, P-2Q, 2P+Q, 2P-Q, 2P+3Q, 2P-3Q, 3P+2Q, 3P-2Q\}$  have to be precomputed. The resulting AJHD of this method is  $3/8 = 0.3750$ . Since both methods reduce the JHW, they are suitable for the Shamir method. However, since they both originate in the JSF, they can only be generated from RtL.

If we want to use even more memory, i.e. extend the digit set even more, the logical choice is the digit set  $\mathcal{D} = \{0, \pm 1, \pm 3, \pm 5, \pm 7\}$ . Now it becomes necessary to precompute 38 points, which require 1520 bytes of memory. While this might fit on a smart card, there is another point of concern. If we consider the customary 160-bit scalars, the computational effort (ECDBL and ECADD operations) to precompute those 38 points would be almost as high as the expected effort to compute the actual multi-scalar multiplication. Therefore it is unwise to use larger digit sets and we can conclude that the digit set  $\mathcal{D} = \{0, \pm 1, \pm 3\}$ , i.e. the use of ten precomputed points is optimal for the Shamir method.

Now we consider two improvements of the Interleave method which also use 10 precomputed points. The first is to use the  $w$ MOF with different widths, namely  $w = 4$  for the first scalar and  $w = 5$  for the second ((4,5)MOF). The used digit sets are  $\mathcal{D}_1 = \{0, \pm 1, \dots, \pm 7\}$  and  $\mathcal{D}_2 = \{0, \pm 1, \dots, \pm 15\}$  and the resulting AHDs are  $1/5 = 0.2$  and  $1/6 \approx 0.1667$  for the first and second scalar, respectively. Then, the average density of ECADD operations is  $11/30 \approx 0.3666$ . The points to precompute are  $\{3P, 3Q, 5P, 5Q, 7P, 7Q, 9Q, 11Q, 13Q, 15Q\}$ . The second method is to apply a fractional sliding window method on MOF from LtR [SST04, Möl02, Möl04]. The resulting representation uses the degenerated digit set  $\mathcal{D} = \{0, \pm 1, \dots, \pm 2^{w-1} + m\}$  and the AHD is  $1/(w + \frac{m+1}{2^{w-1}} + 1)$ . In order to obtain 10 precomputed points we chose  $w = 4$  and  $m = 3$  for both scalars. The resulting digit set is  $\mathcal{D} = \{0, \pm 1, \dots, \pm 11\}$  and the AJHD is  $2/11 \approx 0.1818$  for each scalar. This leads to an average density of ECADD operations of  $4/11 \approx 0.3636$ . Also we have to precompute the points  $\{3P, 3Q, 5P, 5Q, 7P, 7Q, 9P, 9Q, 11P, 11Q\}$ .

From this one can see that in the case of ten precomputed points the Interleave method currently wins over the Shamir method.

## 4 Proposed Scheme

In this section we describe the proposed scheme. At first glance our scheme is similar to [HKPR04], namely the main idea is to apply a sliding window method (SWM) with different widths on the MOF of both scalars from LtR. The difference is that we chose a larger digit set and can therefore use larger window widths. For the reasons explained in Section 3 we chose the digit set  $\mathcal{D} = \{0, \pm 1, \pm 3\}$  and therefore need ten precomputed points. The algorithm is divided in three parts: the *Main Routine*, the *Calculation of Z* and the *Conversion Routine*. Further, the recoding can be performed with the knowledge of at

most 6 bits of each scalar at a time and we will show that the resulting AJHD is 239/661 with the method of stochastic processes.

#### 4.1 First Considerations

First, we want to examine how we can use the MOF representation to decrease the JHW. The first MOF property implies that the absolute value of any  $w$  consecutive MOF bits is at most  $2^{w-1} - 1$ . Therefore, if we take any  $w$  consecutive MOF bits it is possible to represent them using  $w - 1$  zero entries and 1 non-zero entry with absolute value of at most  $2^{w-1} - 1$ . Since we want to use the digit set  $\mathcal{D} = \{0, \pm 1, \pm 3\}$ ,  $w = 3$  holds in our case and by extending this to two scalars, we get

**Lemma 1.** *Given two MOF representations, a SWM can create at most two consecutive zero columns without exceeding the digit set  $\mathcal{D} = \{0, \pm 1, \pm 3\}$ . After that, at least one non-zero column must follow.*

Next, we are interested in the position of the columns which are candidates to become zero.

**Lemma 2.** *Let  $\mu_0$  and  $\mu_1$  be two  $k$ -bit MOF representations. Further, let  $f_0$  and  $f_1$  be the digit of the least non-zero entry of  $\mu_0$  and  $\mu_1$  respectively. The set*

$$Z := \{k - 1, \dots, 0\} \setminus \{f_0, f_1\}$$

*contains the indices of the columns which are candidates to become zero columns.*

Note that for two scalars, we have to scan at least three and at most four columns to create two zero columns.

#### 4.2 The Main Routine

The purpose of this part is to decide on the window width used in a certain step. The widths and the required number of zero columns to create are chosen such that the resulting JHD of the recoded window increases from step to step. In other words, at first we try a width which results in a low JHD and if that fails, we increase the width and accept slightly worse JHD. Table 1 shows the sequence in which the widths and the required zero columns are chosen.

If a recoding with one of the first three widths is possible we recode the window, write it out and proceed to the next column. Otherwise after using the last width, where a recoding is always possible, we check the following two conditions to decide how to proceed.

1. If the last two columns remained unchanged after the recoding we write out the first two columns and proceed the scan with the third column.
2. If the last column has been changed, but does not contain any entries equal to  $\pm 3$  we write out the first three columns and proceed with the last column.

If those two conditions fail we write out all four columns and proceed with the next column.

However, in the case where we reuse an already recoded column, some problems might occur. Now, it is no longer guaranteed that adjacent non-zero bits have opposite signs. Therefore, Lemma 1 doesn't hold anymore and we have to reduce it to

**Lemma 3.** *If we reuse a converted column, a SWM can create at most one consecutive zero column without exceeding the digit set  $\mathcal{D} = \{0, \pm 1, \pm 3\}$ . After that at least one non-zero column must follow.*

According to Lemma 2 now we have to scan at least two and at most three columns in order to create one zero column. Therefore we use a different sequence of widths as shown in Table 1.

Sequence of conversion	without reusing				with reusing			
	1.	2.	3.	4.	1.	2.	3.	4.
zero columns required	1	2	3	2	1	1	2	1
window width	1	3	5	4	1	2	4	3
resulting JHD	0	0.33	0.4	0.5	0	0.5	0.5	0.66

**Table 1.** Sequence of window widths with and without reusing

Again, if a recoding using one of the first three widths is possible we recode the window, write it out and proceed to the next column. Otherwise we apply the fourth conversion and perform the same checks as above. Note that in all cases where we don't reuse an already converted column, Lemma 1 holds again in the next step.

### 4.3 The Calculation Of Z

This method computes the number of zero columns which can be created in a certain window. Therefore it is used by the main routing to decide whether a certain width should be used or not. Further it computes the positions of the columns to become zero, which are needed by the conversion routine. Hence, at first we calculate the set  $Z$  according to Lemma 2. Next, we select a set  $\tilde{Z} \subset Z$  which represents the columns that will actually be converted to zero. This choice is performed according to Lemma 1 or Lemma 3. If we have more than one possibility for  $\tilde{Z}$ , we start picking the leftmost candidates first. In the following examples let  $\bar{x} = -x$ .

*Example 1.*

- a) *Without reusing.* Let  $\mu_0 = \bar{1}01\bar{1}1$ ,  $\mu_1 = 10\bar{1}00$ . Therefore  $f_0 = 0$  and  $f_1 = 2$  holds.

$$\text{Lemma 2} \implies Z := \{4, 3, 2, 1, 0\} \setminus \{2, 0\} = \{4, 3, 1\} \xrightarrow{\text{Lemma 1}} \tilde{Z} = \{4, 3, 1\}$$

- b) *With reusing.* Let  $\mu_0 = \bar{1}\bar{1}\bar{1}1$ ,  $\mu_1 = \bar{1}0\bar{1}1$ . Therefore  $f_0 = 0$  and  $f_1 = 0$  holds.

$$\text{Lemma 2} \implies Z := \{3, 2, 1, 0\} \setminus \{0\} = \{3, 2, 1\} \xrightarrow{\text{Lemma 3}} \tilde{Z} = \{3, 1\}$$

#### 4.4 The Conversion Routine

This part performs the actual recoding of the window. At this point we know which columns shall become zero, therefore it is possible to recode each scalar separately. Each window is scanned from LtR and if a non-zero entry which should become zero is detected, we scan for the next non-zero entry on the right and apply one of the following conversions.

$$\begin{array}{ll}
 (1) & 100\dots 0\bar{1} \mapsto 011\dots 11 \\
 (2) & \bar{1}00\dots 01 \mapsto 0\bar{1}\bar{1}\dots \bar{1}\bar{1} \\
 (3) & 100\dots 01 \mapsto 03\bar{1}\dots \bar{1}\bar{1} \\
 (4) & \bar{1}00\dots 0\bar{1} \mapsto 0\bar{3}1\dots 11
 \end{array}$$

Note, that because of Lemma 2 we are always able to find a non-zero entry to the right in the current window.

*Example 2.*

a) Let  $\mu_0 = \bar{1}01\bar{1}1$ ,  $\mu_1 = 10\bar{1}00$ ,  $\tilde{Z} = \{4, 3, 1\}$ . Applying (1) – (4) yields

$$\begin{array}{ccccccc}
 \bar{1}01\bar{1}1 & \xrightarrow{(2)} & 0\bar{1}\bar{1}\bar{1}1 & \xrightarrow{(4)} & 00\bar{3}\bar{1}1 & \xrightarrow{(2)} & 00\bar{3}0\bar{1} \\
 \uparrow & & \uparrow & & \uparrow & & \\
 10\bar{1}00 & \xrightarrow{(1)} & 01100 & \xrightarrow{(3)} & 00300 & \mapsto & 00300 \\
 \uparrow & & \uparrow & & \uparrow & & 
 \end{array}$$

b) Let  $\mu_0 = \bar{1}\bar{1}\bar{1}1$ ,  $\mu_1 = \bar{1}0\bar{1}1$ ,  $\tilde{Z} = \{3, 1\}$ . Applying (1) – (4) yields

$$\begin{array}{ccccccc}
 \bar{1}\bar{1}\bar{1}1 & \xrightarrow{(4)} & 0\bar{3}\bar{1}1 & \xrightarrow{(2)} & 0\bar{3}0\bar{1} & & \bar{1}0\bar{1}1 & \xrightarrow{(4)} & 0\bar{3}11 & \xrightarrow{(3)} & 0\bar{3}03 \\
 \uparrow & & \uparrow & & & & \uparrow & & \uparrow & & 
 \end{array}$$

#### 4.5 Implementation

The implementation of the three parts of the proposed scheme can be found in Algorithms 1, 2 and 3. They use the following notations: The variables  $u$  and  $l$  denote the first and the last index of the current window, respectively. The variable  $c$  denotes the current case, namely  $c = 0$  if we are reusing an already converted column and  $c = 1$  otherwise. The set  $Z$  to determine which columns should be converted, is represented as a  $k$ -bit array  $z$ , where  $z_j = 1$ , if the  $j$ -th column in the current window is to be converted and  $z_j = 0$ , otherwise. Here  $k$  is the width of the current window and  $j = k - 1, \dots, 0$ . The notation  $d_{i,j}$  denotes the  $j$ -th bit of the  $i$ -th scalar and substrings are denoted by  $d_{0,u..l} := (d_{0,u}, d_{0,u-1}, \dots, d_{0,l})$ . Also,  $\ominus$  denotes the bitwise subtraction which is used for the on-the-fly MOF generation.

---

**Algorithm 1** The Main Routine

---

**Require:** two  $n$ -bit scalars  $d_0$  and  $d_1$  in their binary representation

**Ensure:** recoded representation  $\delta_0$  and  $\delta_1$

```
1:  $d_{0,-1} \leftarrow 0$ ;  $d_{1,-1} \leftarrow 0$ ;  $d_{0,n} \leftarrow 0$ ;  $d_{1,n} \leftarrow 0$ 
2:  $u \leftarrow n$ ;  $c \leftarrow 1$ 
3: while  $u > 0$  do
4:   while  $d_{0,u} = d_{0,u-1} \wedge d_{1,u} = d_{1,u-1} \wedge u > 0$  do
5:      $\mu_{0,u} \leftarrow 0$ ;  $\mu_{1,u} \leftarrow 0$ 
6:      $u \leftarrow u - 1$ ;  $c \leftarrow 1$ 
7:   end while
8:    $l \leftarrow u - 1 - c$ 
9:    $\mu_{0,u+c-1..l} \leftarrow d_{0,u+c-1..l} \ominus d_{0,u+c-2..l-1}$ 
10:   $\mu_{1,u+c-1..l} \leftarrow d_{1,u+c-1..l} \ominus d_{1,u+c-2..l-1}$ 
11:   $z \leftarrow \text{calculateZ}(\mu_{0,u..l}, \mu_{1,u..l}, c)$ 
12:  if  $z_{u-l} + \dots + z_0 \geq 1 + c \vee l \leq 0$  then
13:     $(\mu_{0,u..l}, \mu_{1,u..l}) \leftarrow \text{convert}(\mu_{0,u..l}, \mu_{1,u..l}, z)$ 
14:     $u \leftarrow u - 2 - c$ ;  $c \leftarrow 1$ 
15:  else
16:     $l \leftarrow u - 3 - c$ 
17:     $\mu_{0,u+c-1..l} \leftarrow d_{0,u+c-1..l} \ominus d_{0,u+c-2..l-1}$ 
18:     $\mu_{1,u+c-1..l} \leftarrow d_{1,u+c-1..l} \ominus d_{1,u+c-2..l-1}$ 
19:     $z \leftarrow \text{calculateZ}(\mu_{0,u..l}, \mu_{1,u..l}, c)$ 
20:    if  $z_3 = 1 \wedge z_2 = 0 \wedge z_1 = 1 \wedge z_0 = 0$  then
21:       $(\mu_{0,u..l}, \mu_{1,u..l}) \leftarrow \text{convert}(\mu_{0,u..l}, \mu_{1,u..l}, z)$ 
22:       $u \leftarrow u - 4 - c$ ;  $c \leftarrow 1$ 
23:    else
24:       $l \leftarrow u - 2 - c$ 
25:       $\mu_{0,u+c-1..l} \leftarrow d_{0,u+c-1..l} \ominus d_{0,u+c-2..l-1}$ 
26:       $\mu_{1,u+c-1..l} \leftarrow d_{1,u+c-1..l} \ominus d_{1,u+c-2..l-1}$ 
27:       $z \leftarrow \text{calculateZ}(\mu_{0,u..l}, \mu_{1,u..l}, c)$ 
28:       $(\mu_{0,u..l}, \mu_{1,u..l}) \leftarrow \text{convert}(\mu_{0,u..l}, \mu_{1,u..l}, z)$ 
29:      if  $\mu_{0,l+1..l} = d_{0,l+1..l} \ominus d_{0,l..l-1} \wedge \mu_{1,l+1..l} = d_{1,l+1..l} \ominus d_{1,l..l-1}$  then
30:         $u \leftarrow u - 1 - c$ ;  $c \leftarrow 1$ 
31:      else if  $\mu_{0,l} \neq \pm 3 \wedge \mu_{1,l} \neq \pm 3 \wedge (\mu_{0,l}, \mu_{1,l}) \neq (0, 0)$  then
32:         $u \leftarrow u - 2 - c$ ;  $c \leftarrow 0$ 
33:      else
34:         $u \leftarrow u - 3 - c$ ;  $c \leftarrow 1$ 
35:      end if
36:    end if
37:  end if
38: end while
39: return  $\mu_0, \mu_1$ .
```

---

---

**Algorithm 2** Calculation of  $z$  *calculateZ*

---

**Require:** two  $k$ -bit MOF strings  $\mu_0$  and  $\mu_1$  and the current case  $c$

**Ensure:** the vector  $z$

```
1: for  $i = 0$  to 1 do
2:    $f_i \leftarrow -1$ 
3:   for  $j = k - 1$  down to 0 do
4:     if  $\mu_{i,j} \neq 0$  then
5:        $f_i \leftarrow j$ 
6:     end if
7:   end for
8: end for
9:  $r \leftarrow 0$ 
10: for  $j = k - 1$  down to 0 do
11:   if  $j = f_0 \vee j = f_1 \vee r = 2$  then
12:      $z_j \leftarrow 0$ ;  $r \leftarrow 0$ 
13:   else
14:      $z_j \leftarrow 1$ ;  $r \leftarrow r + 1$ 
15:   end if
16:   if  $c = 0 \wedge j = k - 1 \wedge z_{k-1} = 1$  then
17:      $r \leftarrow 2$ 
18:   end if
19: end for
20: return  $z$ .
```

---

---

**Algorithm 3** Conversion routine *convert*

---

**Require:** two  $k$ -bit MOF strings  $\mu_0$  and  $\mu_1$  and the columns to convert  $z$

**Ensure:** recoded representation of  $\mu_0$  and  $\mu_1$

```
1: for  $i = 0$  to 1 do
2:   for  $j = k - 1$  down to 0 do
3:     if  $z_j = 1 \wedge \mu_{i,j} \neq 0$  then
4:        $s \leftarrow j - 1$ 
5:       while  $\mu_{i,s} = 0$  do
6:          $s \leftarrow s - 1$ 
7:       end while
8:       if  $\mu_{i,j} = -\mu_{i,s}$  then
9:         for  $t = j - 1$  down to  $s$  do
10:           $\mu_{i,t} \leftarrow \mu_{i,j}$ 
11:        end for
12:         $\mu_{i,j} \leftarrow 0$ 
13:      else if  $\mu_{i,j} = \mu_{i,s}$  then
14:        for  $t = j - 2$  down to  $s$  do
15:           $\mu_{i,t} \leftarrow -\mu_{i,j}$ 
16:        end for
17:         $\mu_{i,j-1} \leftarrow 3 \cdot \mu_{i,j}$ ;  $\mu_{i,j} \leftarrow 0$ 
18:      end if
19:    end if
20:  end for
21: end for
22: return  $\mu_0, \mu_1$ .
```

---

#### 4.6 Average Joint Hamming Density

The next step is to prove, that the representation generated by the proposed scheme indeed results in an AJHD of  $239/661 \approx 0.3615$ . We will calculate the AJHD using Markov Chains [Häg02]. Figure 1 shows the transition graph of the proposed scheme. Each state indicates the number of columns currently scanned, the number of columns which are reused (the boxed ones) and the probability with which the state changes into another. Whenever a recoding was performed, we jump back to state 1. Those changes are indicated by arrows with a dot at the end.

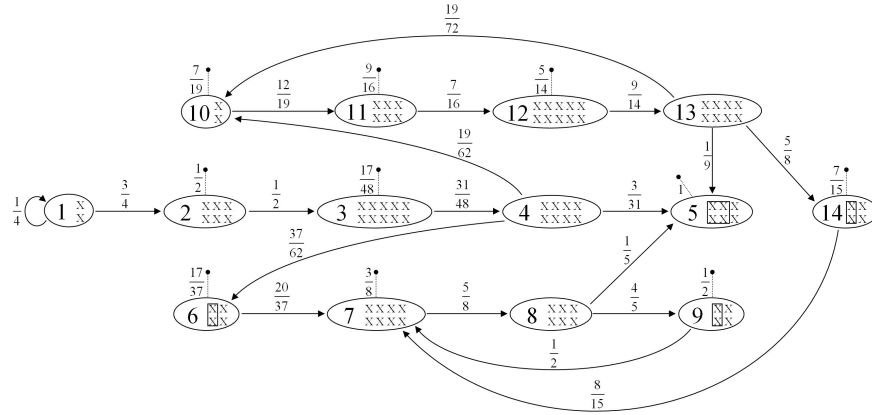


Fig. 1. Transition graph

The transition probabilities are given by the matrix  $(p_{ij}) := P(S_i \mapsto S_j)$ , where  $S_i \mapsto S_j$  indicates that state  $S_i$  changes into  $S_j$ . Those numbers were obtained by checking all cases. Also we need the matrices  $(t_{ij})$  which contains the total number of columns written out by the algorithm if  $S_i \mapsto S_j$  and  $(n_{ij})$  which contains the number of non-zero columns written out if  $S_i \mapsto S_j$ ,  $i, j = 1, \dots, 14$ . The non-zero entries of those three matrices as well as the line in Algorithm 1 where the changes of states occur are summarized in Table 2.

Since this Markov chain is irreducible and aperiodic, it exists a stationary distribution

$$\pi = \left( \frac{976}{2885}, \frac{732}{2885}, \frac{366}{2885}, \frac{1891}{23080}, \frac{227}{17310}, \frac{2257}{46160}, \frac{323}{8655}, \frac{323}{13848}, \frac{323}{17310}, \frac{76}{2885}, \frac{48}{2885}, \frac{21}{2885}, \frac{27}{5770}, \frac{27}{9232} \right)$$

Using the stationary distribution  $\pi$  and the matrices  $(p_{ij})$ ,  $(t_{ij})$  and  $(n_{ij})$  we can calculate the AJHD as follows. According to the definition of the AJHD, we need the average number of (non-zero) columns written out by the algorithm for any possible transition  $S_i \mapsto S_j$ ,  $i, j = 1, \dots, 14$ . For one fixed transition  $S_i \mapsto S_j$ , these numbers obviously are  $(n_{i\bar{j}} \cdot p_{i\bar{j}}) t_{i\bar{j}} \cdot p_{i\bar{j}}$ . If we consider a whole state  $S_i$ , we have to add all values  $(n_{i\bar{j}} \cdot p_{i\bar{j}}) t_{i\bar{j}} \cdot p_{i\bar{j}}$ ,  $j = 1, \dots, 14$ . Finally if we consider all states  $S_i$ ,  $i = 1, \dots, 14$  we have to multiply this sum with  $\pi_i$ , the probability that the algorithm currently is in this state and add them together. The AJHD

line	$S_i \mapsto S_j$	$p_{ij}$	$t_{ij}$	$n_{ij}$	line	$S_i \mapsto S_j$	$p_{ij}$	$t_{ij}$	$n_{ij}$
4	$S_1 \mapsto S_1$	1/4	1	0	20	$S_3 \mapsto S_1$	17/48	5	2
4	$S_6 \mapsto S_1$	17/37	2	1	20	$S_7 \mapsto S_1$	3/8	4	2
4	$S_9 \mapsto S_1$	1/2	2	1	20	$S_{12} \mapsto S_1$	5/14	5	2
4	$S_{10} \mapsto S_1$	7/19	1	0	23	$S_3 \mapsto S_4$	31/48	0	0
4	$S_{14} \mapsto S_1$	7/15	2	1	23	$S_7 \mapsto S_8$	5/8	0	0
4/15	$S_6 \mapsto S_7$	20/37	0	0	23	$S_{12} \mapsto S_{13}$	9/14	0	0
4/15	$S_9 \mapsto S_7$	1/2	0	0	29	$S_4 \mapsto S_5$	3/31	2	1
4/15	$S_{14} \mapsto S_7$	8/15	0	0	29	$S_8 \mapsto S_5$	1/5	1	1
7	$S_1 \mapsto S_2$	3/4	0	0	29	$S_{13} \mapsto S_5$	1/9	2	1
7	$S_{10} \mapsto S_{11}$	12/19	0	0	31	$S_4 \mapsto S_6$	37/62	3	1
12	$S_2 \mapsto S_1$	1/2	3	1	31	$S_8 \mapsto S_9$	4/5	2	1
12	$S_5 \mapsto S_1$	1	3	1	31	$S_{13} \mapsto S_{14}$	5/8	3	1
12	$S_{11} \mapsto S_1$	9/16	3	1	33	$S_4 \mapsto S_{10}$	19/62	4	2
15	$S_2 \mapsto S_3$	1/2	0	0	33	$S_{13} \mapsto S_{10}$	19/72	4	2
15	$S_{11} \mapsto S_{12}$	7/16	0	0					

**Table 2.** Non-zero entries of the matrices  $p_{ij}$ ,  $t_{ij}$  and  $n_{ij}$

then is the quotient of the value for the non-zero columns and the value for all columns, namely

$$\text{AJHD} = \frac{\sum_{i=1}^{14} \pi_i \sum_{j=1}^{14} n_{ij} \cdot p_{ij}}{\sum_{i=1}^{14} \pi_i \sum_{j=1}^{14} t_{ij} \cdot p_{ij}} = \frac{239}{661} \approx 0.3615733$$

Because such calculations involve a great number of difficult to estimate values, it is very likely that some error occurs. However, we are happy to report that this AJHD was confirmed by experimental results. While for 160-bit scalars the estimated AJDH was 0.3636836, for a larger bit length it converged against the calculated value.

## 5 Comparison

In this section we want to compare the average number of ECADD operations required for computing  $uP + vQ$  of the proposed scheme and the methods of Section 3. Table 3 shows these values and also the direction in which the scalar are recoded, i.e. LtR od RtL.

According to Table 3 the proposed scheme requires the least average number of additions and is therefore the first scheme with which the Shamir method wins over the Interleave method. Compared to the first two methods [Ava02,KZZ04] the memory usage for the recoding is reduced due to the LtR generation. The problem with the last two methods is that the underlying representations have

Scheme	avg. number of ECADD	direction
Shamir+[Ava02]	$\frac{3}{8}n = 0.3750n$	RtL
Shamir+[KZZ04]	$\frac{121}{326}n \approx 0.3712n$	RtL
Interleave+(4,5)MOF	$(\frac{1}{6} + \frac{1}{5})n \approx 0.3666n$	LtR
Interleave+[Möl04]	$(\frac{2}{11} + \frac{2}{11})n \approx 0.3636n$	LtR
Shamir+Section 4	$\frac{239}{661}n \approx 0.3615n$	LtR

**Table 3.** Average number of additions and direction of recoding

been proven to be minimal [Ava04,Möl04]. Therefore it is not possible to further reduce the average number of additions using methods which reduce the AHD, while for methods which reduce the AJHD a minimal representation is still unknown.

## 6 Conclusion

In this paper we proposed a new algorithm to speed up the calculation of  $uP+vQ$  using the Shamir method. The main point was extending the digit set of the JSF to  $\mathcal{D} = \{0, \pm 1, \pm 3\}$ . We proved that the AJHD of our scheme is  $239/661 \approx 0.3615$ , which is superior to any known method which uses ten precomputed points. The proposed scheme is the first to enhance the Shamir method such that it wins over the Interleave method and compared to the JSF, the multi-scalar multiplication can be sped up by more than 10%. Due to the LtR fashion of our algorithm, the memory consumption for the recoding is reduced and we need only the knowledge of 6 joint bits of the binary representations to generate the new representation. Future work may include an improvement of the AJHD and a generalisation to an arbitrary number of scalars.

## References

- [Ava02] Avanzi, R., *On multi-exponentiation in cryptography*, Cryptology ePrint Archive: Report 2002/154, 2002, available at <http://eprint.iacr.org/2002/154/>
- [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, pp. 130-143
- [BSS99] Blake, I., Seroussi, G., and Smart, N., *Elliptic Curves in Cryptography*, Cambridge University Press, 1999.
- [CMO98] Cohen, H., Miyaji, A., Ono, T., *Efficient Elliptic Curve Exponentiation Using Mixed Coordinates*, Advances in Cryptology - ASIACRYPT '98, LNCS1514, (1998), 51-65.
- [ElG85] ElGamal, T., *A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms*, IEEE Transactions on Information Theory, Vol. 31, IEEE 1985, pp. 469-472.

- [GHPT03] Grabner, P., Heuberger, C., Prodinger, H., Thuswaldner J., *Analysis of linear combination algorithms in cryptography*, available at <http://www.opt.math.tu-graz.ac.at/~cheub/publications/>
- [Gor98] Gordon, D., *A survey of fast exponentiation methods*, Journal of Algorithms, vol.27, (1998), 129-146.
- [Häg02] Häggström, O., *Finite Markov Chains and Algorithmic Applications*, London Mathematical Society Student Texts 52, Cambridge University Press, (2002).
- [HKPR04] Heuberger, C., Katti, R., Prodinger, H., Ruan, X., *The Alternating Greedy Expansion and Applications to Left-To-Right Algorithms in Cryptography* available at <http://www.opt.math.tu-graz.ac.at/~cheub/publications/>
- [Kob87] Koblitz, N., *Elliptic Curve Cryptosystems*, Math. Comp. 48, (1987), 203-209.
- [KZZ04] Kuang, B., Zhu, Y., Zhang, Y., *An Improved Algorithm for  $uP+vQ$  using  $JSF_3$* , Applied Cryptography and Network Security - ACNS 2005, LNCS 3089, pp. 467-478
- [Mil86] Miller, V.S., *Use of Elliptic Curves in Cryptography*, Advances in Cryptology - CRYPTO '85, LNCS218, (1986), 417-426.
- [MOC97] Miyaji, A., Ono, T., and Cohen, H., *Efficient Elliptic Curve Exponentiation*, Information and Communication Security, ICICS 1997, LNCS 1334, (1997), 282-291.
- [MO90] Morain, F., Olivos, J., *Speeding Up the Computations on an Elliptic Curve using Addition-Subtraction Chains*, Informa. Theor. Appl., 24, (1990), pp.531-543.
- [Möl01] Möller, B., *Algorithms for Multi-exponentiation*, Selected Areas in Cryptography - SAC 2001, LNCS 2259, pp. 165-180
- [Möl02] Möller, B., *Improved Techniques for Fast Exponentiation*, Information Security and Cryptology ICISC 2002. LNCS 2587, pp. 298312
- [Möl04] Möller, B., *Fractional Windows Revisited: Improved Signed-Digit Representations for Efficient Exponentiation*, Information Security and Cryptology ICISC 2004, to appear.
- [OSST04] Okeya, K., Schmidt-Samoa, K., Spahn, C., Takagi, T., *Signed Binary Representations Revisited*, Advances in Cryptology - CRYPTO 2004, LNCS 3152, pp.123-139, available at <http://eprint.iacr.org/2004/195/>
- [Pro03] Proos, J., *Joint Sparse Forms and Generating Zero Columns when Combining*, Technical Report of the Centre for Applied Cryptographic Research, University of Waterloo - CACR, CORR 2003-23, 2003, available at <http://www.cacr.math.uwaterloo.ca>.
- [SST04] Schmidt-Samoa, K., Semay, O., Takagi, T., *Analysis of Some Efficient Window Methods and their Application to Elliptic Curve Cryptosystems*, Technical Report No. TI-3/04, 16. August 2004.
- [Sol00] Solinas, J.A., *Efficient Arithmetic on Koblitz Curves*, Design, Codes and Cryptography, 19, (2000), 195-249.
- [Sol01] Solinas, J.A., *Low-weight binary representations for pairs of integers*, Technical Report of the Centre for Applied Cryptographic Research, University of Waterloo - CACR, CORR 2001-41, 2001, available at <http://www.cacr.math.uwaterloo.ca>