Extended Double-Base Number System with applications to Elliptic Curve Cryptography

Christophe Doche[1] and Laurent Imbert[2]

[1] Department of Computing
Macquarie University, Australia
doche@ics.mq.edu.au

[2] Laboratoire d'Informatique, de Robotique
et de Microélectronique de Montpellier
CNRS UMR 5506, Montpellier, France
Laurent.Imbert@lirmm.fr

Abstract. We investigate the impact of larger digit sets on the length of Double-Base Number system (DBNS) expansions. We present a new representation system called *extended DBNS* whose expansions can be extremely sparse. When compared with double-base chains, the average length of extended DBNS expansions of integers of size in the range 200–500 bits is approximately reduced by 20% using one precomputed point, 30% using two, and 38% using four. We also discuss a new approach to approximate an integer n by $d2^a3^b$ where d belongs to a given digit set. This method, which requires some precomputations as well, leads to realistic DBNS implementations. Finally, a left-to-right scalar multiplication relying on extended DBNS is given. On an elliptic curve where operations are performed in Jacobian coordinates, improvements of up to 13% overall can be expected with this approach when compared to window NAF methods using the same number of precomputed points. In this context, it is therefore the fastest method known to date to compute a scalar multiplication on a generic elliptic curve.

Keywords. Double-Base Number System, Elliptic Curve Cryptography.

1 Introduction

Curve-based Cryptography, especially Elliptic Curve Cryptography, has attracted more and more attention since its introduction about twenty years ago [Mil86,Kob87,Kob89], as reflected by the abundant literature on the subject, *cf.* [BSS99,HMV03,ACD$^+$05,BSS05]. In curve based cryptosystems, the core operation that needs to be optimized as much as possible is a scalar multiplication. The standard method, based on ideas well known already more than two thousand years ago, to efficiently compute such a multiplication is the double and add method, whose complexity is linear in terms of the size of the input. Several ideas have been introduced to improve this method, *cf.* [Doc05] for an overview. In the remainder, we will mainly focus on two approaches:

- Use a representation such that the expansion of the scalar multiple is sparse. For instance, the Non-Adjacent Form (NAF), *cf.* [MO90], has a density of $1/3$ whereas the average density of a binary expansion is $1/2$. This improvement is mainly obtained by adding -1 to the set $\{0, 1\}$ of possible coefficients used in binary notation. Another example is the Double-Base Number System (DBNS), *cf.* [DJM99], in which an integer is represented as a sum of products of powers of 2 and 3. Such expansions can be extremely sparse, *cf.* Section 2.
- Introduce precomputations to enlarge the set of possible coefficients in the expansion and reduce its density. The k-ary and sliding window methods as well as window NAF methods [MOC97,TYW04] fall under this category.

In the present work, we mix these two ideas. Namely, we investigate how precomputations can be used with the DBNS and we evaluate their impact on the overall complexity of a scalar multiplication.

Also, computing a sparse DBNS expansion can be very time-consuming although it is often neglected when compared with other representations. We introduce several improvements that considerably speed up the computation of a DBNS expansion, *cf.* Section 4 and Appendix A.

The plan of the paper is as follows. In Section 2, we recall the definition and basic properties of the DBNS. In Section 3, we describe how precomputations can be efficiently used with the DBNS. Section 4 is devoted to implementation aspects and explains how to quickly compute DBNS expansions. In Section 5, we present a series of tests and comparisons with existing methods before concluding in Section 6.

2 Overview of the DBNS

In *Double-Base Number System* first considered by Dimitrov *et al.* [DJM98] in a cryptographic context, any positive integer n is represented as

$$n = \sum_{i=1}^{\ell} d_i 2^{a_i} 3^{b_i}, \text{ with } d_i \in \{-1, 1\}. \tag{1}$$

This representation is obviously not unique and is in fact highly redundant. Given an integer n, it is straightforward to find a DBNS expansion using a greedy-type approach. Indeed, starting with $t = n$, the main task at each step is to find the $\{2,3\}$-integer z that is the closest to t (i.e. the integer z of the form $2^a 3^b$ such that $|t - z|$ is minimal) and then set $t = t - z$. This is repeated until t becomes 0. See Example 2, for an illustration.

Remark 1. *Finding the best* $\{2,3\}$*-approximation of an integer* t *in the most efficient way is an interesting problem on its own. One option is to scan all the points with integer coordinates near the line* $y = -x \log_3 2 + \log_3 t$ *and keep only the best approximation. A much more sophisticated method involves continued fractions and Ostrowski's number system, cf. [BI04]. It is to be noted that these methods are quite time-consuming. See Section 4, for a more efficient approach.*

Example 2. *Take the integer* $n = 841232$*. We have the sequence of approximations*

$$\begin{aligned} 841232 &= 2^7 3^8 + 1424, \\ 1424 &= 2^1 3^6 - 34, \\ 34 &= 2^2 3^2 - 2. \end{aligned}$$

As a consequence, $841232 = 2^7 3^8 + 2^1 3^6 - 2^2 3^2 + 2^1$.

It has been shown that every positive integer n can be represented as the sum of at most $O\left(\frac{\log n}{\log \log n}\right)$ signed $\{2,3\}$-integers. For instance, see [DJM98] for a proof. Note that a greedy algorithm is suitable to find such short expansions.

This initial class of DBNS is therefore very sparse. When one endomorphism is virtually free, like for instance triplings on supersingular curves defined over $\mathbb{F}_3$, the DBNS can be used to efficiently compute $[n]P$ with $\max a_i$ doublings, a very low number of additions, and the necessary number of triplings [CS05]. Note that this idea has recently been extended to Koblitz curves [AS06]. Nevertheless, it is not really suitable to compute scalar multiplications in general. For generic curves where both doublings and triplings are expensive, it is essential to minimize the number of applications of these two endomorphisms. Now, one needs at least $\max a_i$ doublings and $\max b_i$ triplings to compute $[n]P$ using (1). However, given the DBNS expansion of n returned by a greedy approach, it seems to be highly non-trivial, if not impossible, to attain these two lower bounds *simultaneously*.

So, for generic curves the DBNS needs to be adapted to compete with other methods. The concept of *double-base chain*, introduced in [DIM05], is a special type of DBNS. The idea is still to represent n as in (1) but with the extra requirements $a_1 \geqslant a_2 \geqslant \cdots \geqslant a_\ell$ and $b_1 \geqslant b_2 \geqslant \cdots \geqslant b_\ell$. These properties allow to compute $[n]P$ from right-to-left very easily. It is also possible to use a Horner-like scheme that operates from left-to-right. These two methods are illustrated after Example 3.

Note that, it is easy to accommodate these requirements by restraining the search of the best exponents (a_{j+1}, b_{j+1}) to the interval $[0, a_j] \times [0, b_j]$.

Example 3. *A double-base chain of n can be derived from the following sequence of equalities:*

$$\begin{aligned} 841232 &= 2^7 3^8 + 1424, \\ 1424 &= 2^1 3^6 - 34, \\ 34 &= 3^3 + 7, \\ 7 &= 3^2 - 2, \\ 2 &= 3^1 - 1. \end{aligned}$$

As a consequence, $841232 = 2^7 3^8 + 2^1 3^6 - 3^3 - 3^2 + 3^1 - 1$.

In that particular case, the length of this double-base chain is strictly bigger than the one of the DBNS expansion in Example 2. This is true in general as well and the difference can be quite large. It is not known whether the bound $O\big(\frac{\log n}{\log\log n}\big)$ on the number of terms is still valid for double-base chains.

However, computing $[841232]P$ is now a trivial task. From right-to-left, we need two variables. The first one, T being initialized with P and the other one, S set to point at infinity. The successive values of T are then P, $[3]P$, $[3^2]P$, $[3^3]P$, $[2^1 3^6]P$, and $[2^7 3^8]P$, and at each step T is added to S. Doing that, we obtain $[n]P$ with 7 doublings, 8 triplings, and 5 additions. To proceed from left-to-right, we notice that the expansion that we found can be rewritten as

$$841232 = 3\big(3\big(3\big(2^1 3^3 (2^6 3^2 + 1) - 1\big) - 1\big) + 1\big) - 1,$$

which implies that

$$[841232]P = [3]\big([3]\big([3]\big([2^1 3^3]([2^6 3^2]P + P) - P\big) - P\big) + P\big) - P.$$

Again, 7 doublings, 8 triplings, and 5 additions are necessary to obtain $[n]P$.

More generally, one needs exactly a_1 doublings and b_1 triplings to compute $[n]P$ using double-base chains. The value of these two parameters can be optimized depending on the size of n and the respective complexities of a doubling and a tripling, see Figures 2 and 4.

To further reduce the complexity of a scalar multiplication, one option is to reduce the number of additions, that is minimize the density of DBNS expansions. A standard approach to achieve this goal is to enlarge the set of possible coefficients, which ultimately means using precomputations.

3 Precomputations for DBNS scalar multiplication

We suggest to use precomputations in two ways. The first idea, which applies only to double-base chains, can be viewed as a two-dimensional window method.

3.1 Window method

Given integers w_1 and w_2, we represent n as in (1) but with coefficients d_i in the set $\{\pm 1, \pm 2^1, \pm 2^2, \ldots, \pm 2^{w_1}, \pm 3^1, \pm 3^2, \ldots, \pm 3^{w_2}\}$. This is an indirect way to relax the conditions $a_1 \geqslant a_2 \geqslant \cdots \geqslant a_\ell$ and $b_1 \geqslant b_2 \geqslant \cdots \geqslant b_\ell$ in order to find better approximations and hopefully sparser expansions. This method, called (w_1, w_2)-*double-base chain*, lies somewhere between normal DBNS and double-base chain methods.

Example 4. *The DBNS expansion of* $841232 = 2^7 3^8 + 2^1 3^6 - 2^2 3^2 + 2^1$, *can be rewritten as* $841232 = 2^7 3^8 + 2^1 3^6 - \mathbf{2} \times 2^1 3^2 + 2^1$, *which is a* $(1,0)$*-window-base chain. The exponent* a_3 *that was bigger than* a_2 *in Example 2 has been replaced by* a_2 *and the coefficient* d_3 *has been multiplied by* 2 *accordingly. As a result, we now have two decreasing sequences of exponents and the expansion is only four terms long.*

It remains to see how to compute $[841232]P$ from this expansion. The right-to-left scalar multiplication does not provide any improvement, but this is not the case for the left-to-right approach. Namely, writing

$$841232 = 2\big(3^2\big(3^4(2^6 3^2 + 1) - 2\big) + 1\big),$$

we see that

$$[841232]P = [2]\big([3^2]\big([3^4]([2^6 3^2]P + P) - [2]P\big) + P\big).$$

If $[2]P$ is stored along the computation of $[2^6 3^2]P$ then 7 doublings, 8 triplings and only 3 additions are necessary to obtain $[841232]P$.

It is straightforward to design an algorithm to produce (w_1, w_2)-double-base chains. We present a more general version in the following, *cf.* Algorithm 1. See Remark 6 (v) for specific improvements to (w_1, w_2)-double-base chains. Also a left-to-right scalar multiplication algorithm can easily be derived from this method, *cf.* Algorithm 2.

The second idea to obtain sparser DBNS expansions is to generalize the window method such that any set of coefficients is allowed.

3.2 Extended DBNS

In a (w_1, w_2)-double-base chain expansion, the coefficients are signed powers of 2 or 3. Considering other sets $\mathcal{S}$ of coefficients, for instance odd integers coprime with 3, should further reduce the average length of DBNS expansions. We call this approach *extended DBNS* and denote it by $\mathcal{S}$-DBNS.

Example 5. *We have* $841232 = 2^7 3^8 + \mathbf{5} \times 2^5 3^2 - 2^4$. *The exponents form two decreasing sequences, but the expansion has only three terms. Assuming that* $[5]P$ *is precomputed, it is possible to obtain* $[841232]P$ *as*

$$[2^4]\big([2^1 3^2]([2^4 3^6]P + [5]P) - P\big)$$

with 7 *doublings,* 8 *triplings, and only* 2 *additions*

This strategy applies to any kind of DBNS expansion. In the following, we present a greedy-type algorithm to compute extended double-base chains.

Algorithm 1. Extended double-base chain greedy algorithm

INPUT: A positive integer n, a parameter a_0 such that $a_0 \leqslant \lceil \log_2 n \rceil$, and a set $\mathcal{S}$ containing 1.

OUTPUT: Three sequences $(d_i, a_i, b_i)_{1 \leqslant i \leqslant \ell}$ such that $n = \sum_{i=1}^{\ell} d_i 2^{a_i} 3^{b_i}$ with $|d_i| \in \mathcal{S}$, $a_1 \geqslant a_2 \geqslant \cdots \geqslant a_\ell$, and $b_1 \geqslant b_2 \geqslant \cdots \geqslant b_\ell$.

---

1. $b_0 \leftarrow \lceil (\log_2 n - a_0) \log_2 3 \rceil$ [See Remark 6 (ii)]
2. $i \leftarrow 1$ and $t \leftarrow n$
3. $s \leftarrow 1$ [to keep track of the sign]
4. **while** $t > 0$ **do**
5. find the best approximation $z = d_i 2^{a_i} 3^{b_i}$ of t with $d_i \in \mathcal{S}$, $0 \leqslant a_i \leqslant a_{i-1}$, and $0 \leqslant b_i \leqslant b_{i-1}$
6. $d_i \leftarrow s \times d_i$

7. **if** $t < z$ **then** $s \leftarrow -s$
8. $t \leftarrow |t - z|$
9. $i \leftarrow i + 1$
10. **return** (d_i, a_i, b_i)

Remarks 6.

(i) Algorithm 1 processes the bits of n from left-to-right. It terminates since the successive values of t form a strictly decreasing sequence.

(ii) The parameters a_0 and b_0 are respectively the biggest power of 2 and 3 allowed in the expansion. Their values have a great influence on the density of the expansion, *cf.* Section 5 for details.

(iii) To compute normal DBNS sequences instead of double-base chains, replace the two conditions $0 \leqslant a_i \leqslant a_{i-1}$, $0 \leqslant b_i \leqslant b_{i-1}$ in Step 5 by $0 \leqslant a_i \leqslant a_0$ and $0 \leqslant b_i \leqslant b_0$.

(iv) In the following, we explain how to find the best approximation $d_i 2^{a_i} 3^{b_i}$ of t in a very efficient way. In addition, the proposed method has a time-complexity that is mainly independent of the size of $\mathcal{S}$ and not directly proportional to it as with a naïve search. See Section 4 for details.

(v) To obtain (w_1, w_2)-double-base chains, simply ensure that $\mathcal{S}$ contains only powers 2 and 3. However, there is a more efficient way. First, introduce two extra variables $a_{\max}$ and $b_{\max}$, initially set to a_0 and b_0 respectively. Then in Step 5, search for the best approximation z of t of the form $2^{a_i} 3^{b_i}$ with $(a_i, b_i) \in [0, a_{\max} + w_1] \times [0, b_{\max} + w_2] \setminus [a_{\max} + 1, a_{\max} + w_1] \times [b_{\max} + 1, b_{\max} + w_2]$. In other words, we allow one exponent to be slightly bigger than its current maximal bound, but the (exceptional) situation where $a_i > a_{\max}$ and $b_i > b_{\max}$ simultaneously is forbidden. Otherwise, we should be obliged to include in $\mathcal{S}$ products of powers of 2 and 3 and increase dramatically the number of precomputations. Once the best approximation has been found, if a_i is bigger than $a_{\max}$, then a_i is changed to $a_{\max}$ while d_i is set to $2^{a_i - a_{\max}}$. If b_i is bigger than $b_{\max}$, then b_i is changed to $b_{\max}$ while d_i is set to $3^{b_i - b_{\max}}$. Finally, do $a_{\max} \leftarrow \min(a_i, a_{\max})$ and $b_{\max} \leftarrow \min(b_i, b_{\max})$ and the rest of the Algorithm remains unchanged.

(vi) Examples of sets $\mathcal{S}$ used in Section 5 are all subset of $\{1, 5, 7, 11, 13, 17, 19, 23, 25\}$.

We now give an algorithm to compute a scalar multiplication from the expansion returned by Algorithm 1.

Algorithm 2. Extended double-base chain scalar multiplication

INPUT: A point P on an elliptic curve E, a positive integer n represented by the sequence $(d_i, a_i, b_i)_{1 \leqslant i \leqslant \ell}$ as returned by Algorithm 1, and the points $[k]P$ for each $k \in \mathcal{S}$.
OUTPUT: The point $[n]P$ on E.

1. $T \leftarrow O_E$ [O_E is the point at infinity on E]
2. set $a_{\ell+1} \leftarrow 0$ and $b_{\ell+1} \leftarrow 0$
3. **for** $i = 1$ **to** ℓ **do**
4. $\quad T \leftarrow T \oplus [d_i]P$
5. $\quad T \leftarrow [2^{a_i - a_{i+1}} 3^{b_i - b_{i+1}}]T$
6. **return** T

Example 7. *For $n = 841232$, the sequence returned by Algorithm 2 with $a_0 = 8$, $b_0 = 8$, and $\mathcal{S} = \{1, 5\}$ is $(1, 7, 8), (5, 5, 2), (-1, 4, 0)$. Next Table shows the intermediate values of T in Algorithm 2 when it is applied to the sequence above. The computation is the same as in Example 5.*

i	d_i	$a_i - a_{i+1}$	$b_i - b_{i+1}$	T
1	1	2	6	$[2^2 3^6]P$
2	5	1	2	$[2^1 3^2]([2^2 3^6]P + [5]P)$
3	-1	4	0	$[2^4]\big([2^1 3^2]([2^2 3^6]P + [5]P) - P\big)$

Remark 8. *The length of the chain returned by Algorithm 1 greatly determines the performance of Algorithm 2. However, no precise bound is known so far, even in the case of simple double-base chains. So, at this stage our knowledge is only empirical, cf. Figures 2 and 4. More work is therefore necessary to establish the complexity of Algorithm 2.*

4 Implementation aspects

This part describes how to efficiently compute the best approximation of any integer n in terms of $d_1 2^{a_1} 3^{b_1}$ for some $d_1 \in \mathcal{S}$, $a_1 \leqslant a_0$, and $b_1 \leqslant b_0$. The method works on the binary representation of n denoted by $(n)_2$. It operates on the most significant bits of n and uses the fact that a multiplication by 2 is a simple shift.

To make things clear, let us explain the algorithm when $\mathcal{S} = \{1\}$. First, take a suitable bound B and form a two-dimensional array of size

$(B+1) \times 2$. For each $b \in [0, B]$, the corresponding row vector contains $[(3^b)_2, b]$. Then sort this array with respect to the first component using lexicographic order denoted by $\preccurlyeq$ and store the result.

To compute an approximation of n in terms of $2^{a_1}3^{b_1}$ with $a_1 \leqslant a_0$ and $b_1 \leqslant b_0$, find the two vectors v_1 and v_2 such that $v_1[1] \preccurlyeq (n)_2 \preccurlyeq v_2[1]$. This can be done with a binary search in $O(\log B)$ operations.

The next step is to find the first vector v_1' that is before v_1 in the sorted array and that is suitable for the approximation. More precisely, we require that:

- the difference δ_1 between the binary length of n and the length of $v_1'[1]$ satisfies $0 \leqslant \delta_1 \leqslant a_0$,
- the corresponding power of 3, *i.e.* $v_1'[2]$, is less than b_0.

This operation is repeated to find the first vector v_2' that is after v_2 and fulfills the same conditions as above. The last step is to decide which approximation, $2^{\delta_1}3^{v_1'[2]}$ or $2^{\delta_2}3^{v_2'[2]}$, is closer to n.

In case $|\mathcal{S}| > 1$, the only difference is that the array is of size $\big(|\mathcal{S}|(B+1)\big) \times 3$. Each row vector is of the form $[(d3^b)_2, b, d]$ for $d \in \mathcal{S}$ and $b \in [0, B]$. Again the array is sorted with respect to the first component using lexicographic order. Note that multiplying the size of the table by $|\mathcal{S}|$ has only a negligible impact on the time complexity of the binary search. See Appendix A, for a concrete example and some improvements to this approach.

This approximation method ultimately relies on the facts that lexicographic and natural orders are the same for binary sequences of the same length and also that it is easy to adjust the length of a sequence by multiplying it by some power of 2. The efficiency comes from the sorting operation (done once at the beginning) that allows to retrieve which precomputed binary expansions are close to n, by looking only at the most significant bits.

For environments with constrained memory, it may be difficult or even impossible to store the full table. In this case, we suggest to precompute only the first byte or the first two bytes of the binary expansions of $d3^b$ together with their binary length. This information is sufficient to find two approximations A_1, A_2 in the table such that $A_1 \leqslant n \leqslant A_2$, since the algorithm operates only on the most significant bits. However, this technique is more time-consuming since it is necessary to actually *compute* at least one approximation and sometimes more, if the first bits are not enough to decide which approximation is the closest to n.

In Table 1, we give the precise amount of memory (in bytes) that is required to store the vectors used for the approximation for different values of B. Three situations are investigated, *i.e.* when the first byte, the first two bytes, and the full binary expansions $d3^b$, for $d \in \mathcal{S}$ and $b \leqslant B$ are precomputed and stored.

Bound B	25	50	75	100	125	150	175	200
$\mathcal{S} = \{1\}$								
First byte	33	65	96	127	158	190	221	251
First two bytes	54	111	167	223	279	336	392	446
Full expansion	85	293	626	1,084	1,663	2,367	3,195	4,108
$\mathcal{S} = \{1, 5, 7\}$								
First byte	111	214	317	420	523	627	730	829
First two bytes	178	356	534	712	890	1,069	1,247	1,418
Full expansion	286	939	1,962	3,357	5,122	7,261	9,769	12,527
$\mathcal{S} = \{1, 5, 7, 11, 13\}$								
First byte	185	357	529	701	873	1,045	1,216	1,381
First two bytes	300	597	894	1,191	1,488	1,785	2,081	2,366
Full expansion	491	1,589	3,305	5,642	8,598	12,173	16,364	20,972
$\mathcal{S} = \{1, 5, 7, 11, 13, 17, 19, 23, 25\}$								
First byte	334	643	952	1,262	1,571	1,881	2,190	2,487
First two bytes	545	1,079	1,613	2,148	2,682	3,217	3,751	4,264
Full expansion	906	2,909	6,026	10,255	15,596	22,056	29,630	37,947

Table 1. Precomputations size (in bytes) for various bounds B and sets $\mathcal{S}$

See [DocGP] for an implementation in GP-PARI of Algorithm 1 using the techniques described in this section.

5 Tests and results

In this part, we present some tests that we run to evaluate the relevance of extended double-base chains for scalar multiplications on generic and special elliptic curves defined over $\mathbb{F}_p$, for p of size between 200 and 500 bits. Comparisons with the best systems known so far, including ℓ-NAF$_w$ and normal double-base chains are given.

In the following, we assume that we have three basic operations on a curve E to perform scalar multiplications, namely addition/subtraction, doubling, and tripling. In turn, each one of these elliptic curve operations can be seen as a sequence of inversions I, multiplications M, and squarings S in the underlying field $\mathbb{F}_p$.

There exist different systems of coordinates with different complexities. For many platforms, projective-like coordinates are quite efficient since they do not require any field inversion for addition and doubling, *cf.* [DL05] for a comparison. Thus, our tests will not involve any inversion. Also, to ease comparisons between different scalar multiplication methods, we will make the standard assumption that S is equivalent to 0.8M. Thus, the complexity of a scalar multiplication will be expressed in terms of a number of field multiplications only and will be denoted by N_{M}.

We start with generic curves before considering special families of curves admitting a fast tripling.

5.1 Generic curves

Given any curve $E/\mathbb{F}_p$ in Weierstraß form, it is possible to directly obtain $[3]P$ more efficiently than computing a doubling followed by an addition. Until now, all these direct formulas involved at least one inversion, *cf.* [CJLM06], but recently, an inversion-free tripling formula has been devised for Jacobian projective coordinates [DIM05]. Our comparisons will be made using this system. Within Jacobian coordinates, a point represented by $(X_1 : Y_1 : Z_1)$ corresponds to the affine point $(X_1/Z_1^2, Y_1/Z_1^3)$, if $Z_1 \neq 0$, and to the point at infinity O_E otherwise. A doubling can be done with $4\mathrm{M} + 6\mathrm{S}$, a tripling with $10\mathrm{M} + 6\mathrm{S}$ and a mixed addition, *i.e.* an addition between a point in Jacobian coordinates and an affine point, using $8\mathrm{M} + 3\mathrm{S}$.

With these settings, we display in Figure 1, the number of multiplications N_{M} necessary to compute a scalar multiplication on a 200bit curve with Algorithm 2, for different choices of a_0 and various DBNS methods. Namely, we investigate double-base chains as in [DIM05], window double-base chains with 2 and 8 precomputations, and extended double-base chains with $\mathcal{S}_2 = \{1, 5, 7\}$ and $\mathcal{S}_8 = \{1, 5, 7, 11, 13, 17, 19, 23, 25\}$, as explained in Section 3.2. Comparisons are done on $1,000$ random 200bit scalar multiples. Note that the costs of the precomputations are not included in the results.

Figure 1 indicates that $a_0 = 120$ is close to the optimal choice for every method. This implies that the value of b_0 should be set to 51. Similar computations have been done for sizes between 250 and 500. It appears that a simple and good heuristic to minimize N_{M} is to set $a_0 = \lceil 120 \times \mathrm{size}/200 \rceil$ and b_0 accordingly. These values of a_0 and b_0 are used in the remainder for sizes in $[200, 500]$.

In Figure 2, we display the average length of different extended DBNS expansions in function of the size of the scalar multiple n. Results show that the length of a classic double-base chain is reduced by more than 25% with only 2 precomputations and by 43% with 8 precomputations.

In Table 5, we give the average expansion length ℓ, as well as the maximal power a_1 (resp. b_1) of 2 (resp. 3) in the expansion for different methods and different sizes. The symbol $\#\mathcal{P}$ is equal to the number of precomputed points for a given method and the set $\mathcal{S}_m$ contains the first $m+1$ elements of $\{1, 5, 7, 11, 13, 17, 19, 23, 25\}$. Again, $1,000$ random integers have been considered in each case.

In Table 6, we give the corresponding complexities in terms of the number of multiplications and the gain that we can expect with respect to a window NAF method involving the same number of precomputations.

5.2 Special curves

Similar computations have been performed for the family of special curves described in [DIK06] where a tripling can be computed with 6M+6S. The complexities of a doubling and an addition are respectively 4M + 5S and 8M + 3S. Figure 3 suggests that a good value for a_0 for 200bit exponents is around 100. Again, a good heuristic to minimize N_M is to set $a_0 = \lceil 100 \times \mathrm{size}/200 \rceil$. These values are used in the computations displayed in Figure 4. Also, Tables 7 and 8 are similar to Tables 5 and 6 with these settings.

The results concerning the average expansion length are consistent with the generic case with improvements ranging from 25% to more than 40% depending on the size of $\mathcal{S}$. The scalar multiplication speed-ups are also in line with the generic case.

6 Conclusion

In this work, we introduce a new family of DBNS, called extended DBNS, where the coefficients in the expansion belong to a given set $\mathcal{S}$. A scalar multiplication algorithm relying on this representation and involving precomputations is presented. Also, we desribe a new method to quickly find the best approximation of an integer by $d2^a3^b$ with $d \in \mathcal{S}$. This approach greatly improves the practicality of the DBNS. Extended DBNS sequences give rise to the fastest scalar multiplications known to date for generic elliptic curves. In particular, given a fixed number of precomputations, the extended DBNS is more efficient than any corresponding window NAF method. Gains are especially important for a small number of precomputations, typically up to three points. Improvements larger than 10% over already extremely optimized methods can be expected. Also, this system is more flexible, since it can be used with any given set of coefficients, unlike window NAF methods.

Further research will include an extension of these ideas to Koblitz curves, for which DBNS-based scalar multiplication techniques without precomputations exist already, see [AS06,DJJ+06,ADD+06]. This will most likely lead to appreciable performance improvements.

References

[ACD+05] R. M. Avanzi, H. Cohen, C. Doche, G. Frey, K. Nguyen, T. Lange, and F. Vercauteren, *Handbook of Elliptic and Hyperelliptic Curve Cryptography*, Discrete Mathematics and its Applications (Boca Raton), CRC Press, Inc., 2005.

[ADD+06] R. M. Avanzi, V. S. Dimitrov, C. Doche, and F. Sica. *E*xtending Scalar Multiplication using Double Bases, to appear in Proceedings of Asiacrypt 2006, Lecture Notes in Comput. Sci., Springer, 2006.

[AS06] R. M. Avanzi and F. Sica, *Scalar Multiplication on Koblitz Curves using Double Bases*, to appear in Proceedings of Vietcrypt 2006, Lecture Notes in Comput. Sci., Springer, 2006. See also Cryptology ePrint Archive, Report 2006/067, `http://eprint.iacr.org/`.

[BI04] V. Berthé and L. Imbert, *On converting numbers to the double-base number system*, In F. T. Luk, editor, Advanced Signal Processing Algorithms, Architecture and Implementations XIV, volume 5559 of Proceedings of SPIE, 2004, pp. 70–78.

[BSS99] I. F. Blake, G. Seroussi, and N. P. Smart, *Elliptic curves in cryptography*, London Mathematical Society Lecture Note Series, vol. 265, Cambridge University Press, Cambridge, 1999.

[BSS05] ———, *Advances in elliptic curve cryptography*, London Mathematical Society Lecture Note Series, vol. 317, Cambridge University Press, Cambridge, 2005.

[CJLM06] M. Ciet, M. Joye, K. Lauter, and P. L. Montgomery, *Trading inversions for multiplications in elliptic curve cryptography*, Des. Codes Cryptogr. **39** (2006), no. 2, 189–206.

[CS05] M. Ciet and F. Sica, *An Analysis of Double Base Number Systems and a Sublinear Scalar Multiplication Algorithm*, Progress in Cryptology – Proceedings of Mycrypt 2005, Lecture Notes in Comput. Sci., vol. 3715, Springer, 2005, pp. 171–182.

[DIK06] C. Doche, T. Icart, and D. R. Kohel, *Efficient scalar multiplication by isogeny decompositions*, Public Key Cryptography – PKC 2006, Lecture Notes in Comput. Sci., vol. 3958, Springer, 2006, pp. 191–206.

[DIM05] V. S. Dimitrov, L. Imbert, and P. K. Mishra, *Efficient and secure elliptic curve point multiplication using double-base chains*, Advances in Cryptology – Asiacrypt 2005, Lecture Notes in Comput. Sci., vol. 3788, Springer, 2005, pp. 59–78.

[DJJ+06] V. S. Dimitrov, K. Jarvinen, M. J. Jacobson Jr, W. F. Chan, and Z. Huang, *FPGA Implementation of Point Multiplication on Koblitz Curves Using Kleinian Integers*, to appear in Proceedings of Cryptographic Hardware and Embedded Systems – CHES 2006, Lecture Notes in Comput. Sci., Springer, 2006.

[DJM98] V. S. Dimitrov, G. A. Jullien, and W. C. Miller, *An algorithm for modular exponentiation*, Information Processing Letters **66** (1998), no. 3, 155–159.

[DJM99] V. S. Dimitrov, G. A. Jullien, and W. C. Miller, *Theory and applications of the double-base number system*, IEEE Trans. on Computers **48** (1999), no. 10, 1098–1106.

[DL05] C. Doche and T. Lange, *Arithmetic of Elliptic Curves*, in [ACD+05], pp. 267–302.

[Doc05] C. Doche, *Exponentiation*, in [ACD+05], pp. 145–168.

[DocGP] C. Doche, *A set of GP-PARI functions to compute DBNS expansions.* See `http://www.ics.mq.edu.au/~doche/dbns_basis.gp`.

[HMV03] D. Hankerson, A. J. Menezes, and S. A. Vanstone, *Guide to elliptic curve cryptography*, Springer-Verlag, Berlin, 2003.

[Kob87] N. Koblitz, *Elliptic curve cryptosystems*, Math. Comp. **48** (1987), no. 177, 203–209.

[Kob89] ———, *Hyperelliptic cryptosystems*, J. Cryptology **1** (1989), 139–150.

[Mil86] V. S. Miller, *Use of elliptic curves in cryptography*, Advances in Cryptology – Crypto 1985, Lecture Notes in Comput. Sci., vol. 218, Springer-Verlag, Berlin, 1986, pp. 417–426.

[MO90] F. Morain and J. Olivos, *Speeding up the Computations on an Elliptic Curve using Addition-Subtraction Chains*, Inform. Theor. Appl. **24** (1990), 531–543.

[MOC97] A. Miyaji, T. Ono, and H. Cohen, *Efficient Elliptic Curve Exponentiation*, Information and Communication – ICICS'97, Lecture Notes in Comput. Sci., vol. 1334, Springer, 1997, pp. 282–291.

[TYW04] T. Takagi, S.-M. Yen, and B.-C. Wu, *Radix-r non-adjacent form*, Information Security Conference – ISC 2004, Lecture Notes in Comput. Sci., vol. 3225, Springer-Verlag, Berlin, 2004, pp. 99–110.

Appendix A: Approximation examples

In the following, we display complete examples of the approximation method described in Section 4.

Let $n = 841232$, $\mathcal{S} = \{1, 5\}$, $a_0 = 8$, $b_0 = 8$, and $B = 10$. The array contains 22 different vectors of the form $[(d3^b)_2, b, d]$. Once sorted with respect to the first coordinate using lexicographic order $\preccurlyeq$, it is equal to:

$$
\begin{array}{lr}
[(1)_2, & 0,\ 1] \\
[(1000000000100101)_2, & 8,\ 5] \\
[(10000111)_2, & 3,\ 5] \\
[(100010001011)_2, & 7,\ 1] \\
[(1001)_2, & 2,\ 1] \\
[(1001000000101001101)_2, & 10, 5] \\
[(10010111111)_2, & 5,\ 5] \\
[(100110011100011)_2, & 9,\ 1] \\
[(101)_2, & 0,\ 5] \\
[(1010001)_2, & 4,\ 1] \\
[(10101010110111)_2, & 7,\ 5] \\
[(101101)_2, & 2,\ 5] \\
[(1011011001)_2, & 6,\ 1] \\
[(11)_2, & 1,\ 1] \\
[(11000000001101111)_2, & 9,\ 5] \\
[(110010101)_2, & 4,\ 5] \\
[(1100110100001)_2, & 8,\ 1] \\
[(11011)_2, & 3,\ 1] \\
[(111000111101)_2, & 6,\ 5] \\
[(1110011010101001)_2, & 10, 1] \\
[(1111)_2, & 1,\ 5] \\
[(11110011)_2, & 5,\ 1]
\end{array}
$$

The binary expansion of n is $(11001101011000010000)_2$ and a binary search in the array above reveals that

$$v_1[1] \preccurlyeq (11001101011000010000)_2 \preccurlyeq v_2[1]$$

with $v_1 = [(1100110100001)_2, 8, 1]$ and $v_2 = [(11011)_2, 3, 1]$. Now, the binary length of n is 20, whereas 3^8 is 13bit long. So the difference $\delta_1 = 7$

is in $[0, a_0]$. At the same time, $v_1[2] = 8 \leqslant b_0$. Consequently, we set $v_1' = v_1$ and the first approximation of n is $2^7 3^8$. In the same way, $v_2' = v_2$ and the second approximation of n is $2^{15} 3^3$. So, we have

$$\begin{aligned}
2^7 3^8 &= (11001101000010000000)_2 \\
841232 &= (11001101011000010000)_2 \\
2^{15} 3^3 &= (11011000000000000000)_2.
\end{aligned}$$

The approximation that is the closer to n is $2^7 3^8$, since the first nine bits match.
Next, we need to approximate $n - 2^7 3^8 = 1424 = (10110010000)_2$ by an integer of the form $d_2 2^{a_2} 3^{b_2}$ with $a_2 \leqslant a_1 = 7$ and $b_2 \leqslant b_1 = 8$. Again, a binary search shows that

$$v_1[1] \preccurlyeq (10110010000)_2 \preccurlyeq v_2[1]$$

with $v_1 = [(10101010110111)_2, 7, 5]$ and $v_2 = [(101101)_2, 2, 5]$. In that case, v_1 is not approriate since the binary length of 5×3^7 is too big. The first suitable vector that is less than v_1 is $v_1' = [(1010001)_2, 4, 1]$, which gives rise to the approximation $2^4 3^4$. On the other side, one can check that $v_2' = v_2$ and that the corresponding approximation is $5 \times 2^5 3^2$. So, we have

$$\begin{aligned}
2^4 3^4 &= (10100010000)_2 \\
1424 &= (10110010000)_2 \\
5 \times 2^5 3^2 &= (10110100000)_2.
\end{aligned}$$

The best approximation is $5 \times 2^5 3^2$ and the difference is simply $16 = 2^4$.

Finally, we obtain a $\{1, 5\}$-double-base chain expansion of $n = 841232$, namely $n = 2^7 3^8 + 5 \times 2^5 3^2 - 2^4$, with minimal efforts.

Remarks 9.

(i) In some cases, the algorithm above does not return the closest approximation to n. For instance, if $n = 2045$ the best approximation for $\mathcal{S} = \{1, 5\}$, $a_0 = 10$, $b_0 = 4$, and $B = 10$ is 2160 whereas the algorithm outlined above returns 1920. A brief look at the corresponding binary expansions provides the explanation. Indeed,

$$\begin{aligned}
1920 &= (11110000000)_2 \\
2045 &= (11111111101)_2 \\
2160 &= (100001110000)_2.
\end{aligned}$$

The last vector in the array is $[(11110011)_2, 5, 1]$ but it should not be the last one to be investigated if it is not suitable. In this case, the search should continue with the first vector of the array. The only difference is that the corresponding power of 2 should not be δ_2 anymore but $\delta_2 + 1$, since the approximation is one bit longer. If we do that, the suitable vectors v'_1 and v'_2 are respectively $[(1111)_2, 1, 5]$ and $[(10000111)_2, 3, 5]$. We have $\delta_1 = 11 - 4 = 7$, $\delta_2 = 11 - 8 = 3$, and the corresponding approximations are $1920 = 5 \times 2^7 3$ and $2160 = 5 \times 2^4 3^3$, from which 2160 is chosen.

(ii) The symmetric situation needs to be addressed as well. Take $n = 2104$, with $S = \{1, 5\}$, $a_0 = 10$, $b_0 = 2$, and $B = 10$. With these parameters, the best approximation of n is 1920 whereas the algorithm returns 2304. Indeed,

$$\begin{aligned} 1920 &= (11110000000)_2 \\ 2104 &= (100000111000)_2 \\ 2304 &= (100100000000)_2 . \end{aligned}$$

Here the search should continue with the last vector since the first one is ruled out (a_0 should be at least 11 to allow using $[(1)_2, 0, 1]$). The smallest approximation will be one bit shorter and so the corresponding power of 2 should not be δ_1 but $\delta_1 - 1$. Doing that, we see that $v'_1 = [(1111)_2, 1, 5]$, $v'_2 = [(1001)_2, 2, 1]$, $\delta_1 = \delta_2 = 12 - 4 = 8$, and the corresponding approximations are $1920 = 5 \times 2^7 3$ and $2304 = 2^8 3^2$.

Appendix B: Graphs and tables

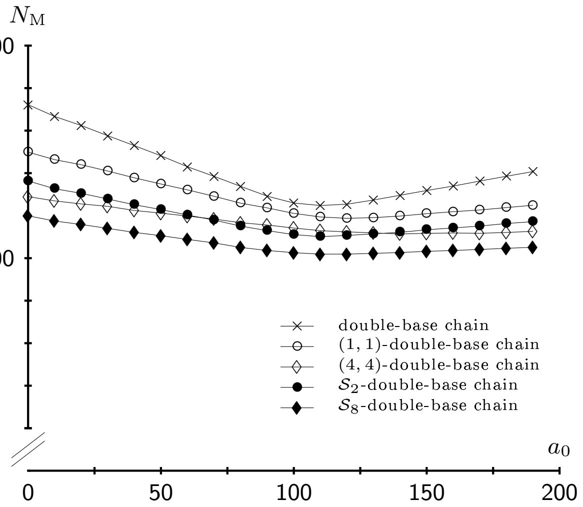

Fig. 1. Average number of multiplications to perform a random scalar multiplication on a generic 200bit curve with various DBNS methods parameterized by a_0

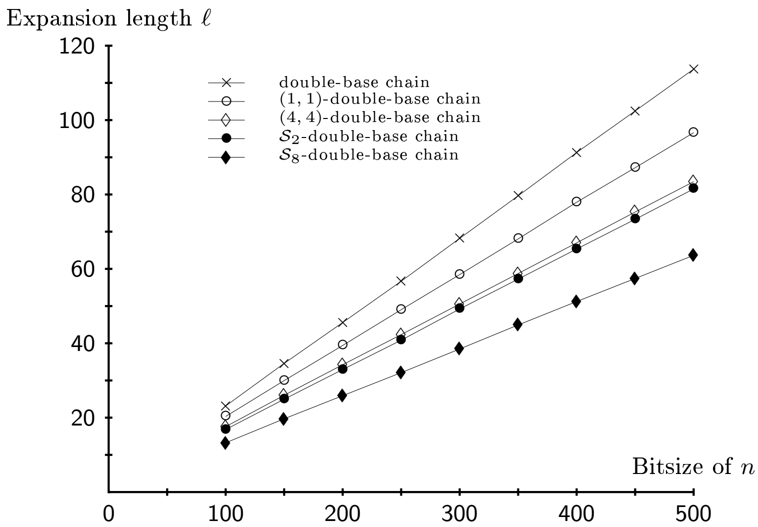

Fig. 2. Average expansion length ℓ of random scalar integers n with various DBNS

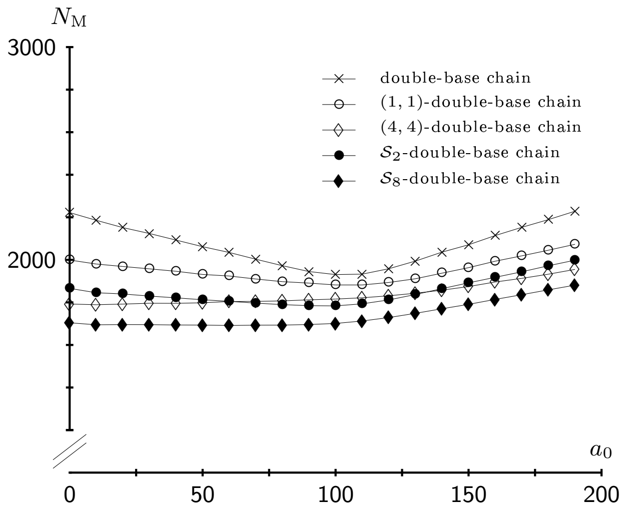

Fig. 3. Average number of multiplications to perform a random scalar multiplication on a special 200bit curve with various DBNS methods parameterized by a_0

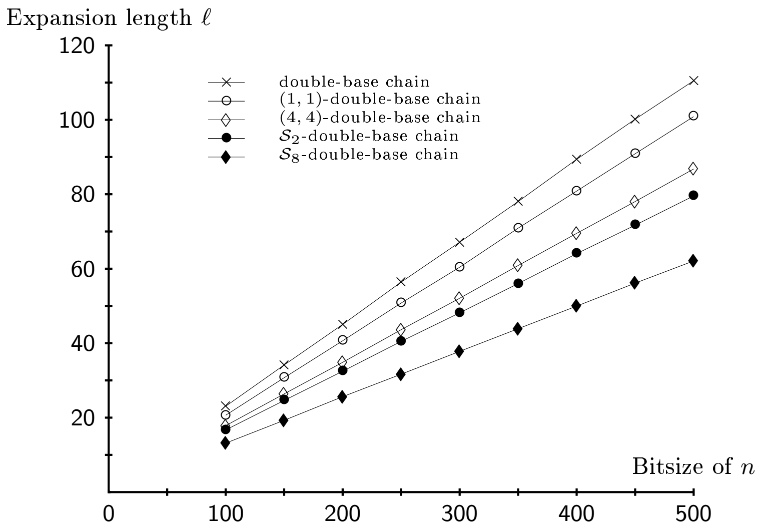

Fig. 4. Average expansion length ℓ of random scalar integers n with various DBNS

Size		200 bits			300 bits			400 bits			500 bits		
	$\#\mathcal{P}$	ℓ	a_1	b_1	ℓ	a_1	b_1	ℓ	a_1	b_1	ℓ	a_1	b_1
$2\mathrm{NAF}_2$	0	66.7	200	0	100	300	0	133.3	400	0	166.7	500	0
Binary/ternary	0	46.1	90.7	68.1	69.2	136.4	102.2	91.9	182.6	136.3	114.4	228.0	170.7
DB-chain	0	45.6	118.7	50.4	68.2	178.7	75.5	91.3	239.0	100.6	113.7	298.6	126.2
$3\mathrm{NAF}_2$	1	50	200	0	75	300	0	100	400	0	125	500	0
$(1,0)$-DB-chain	1	46.8	118.9	50.2	70.5	179.1	75.1	94.5	239.3	100.3	117.7	298.8	125.9
$(0,1)$-DB-chain	1	42.9	118.7	50.4	63.8	178.7	75.5	85.4	239.0	100.6	106.4	298.6	126.2
$\mathcal{S}_1$-DB chain	1	36.8	118.1	49.9	55.0	178.0	75.0	72.9	238.2	100.1	91.0	297.8	125.7
$2\mathrm{NAF}_3$	2	50.4	0	126	75.6	0	189	100.8	0	252	126	0	315
$(1,1)$-DB-chain	2	39.4	118.9	50.2	58.5	179.1	75.1	77.9	239.3	100.3	96.6	298.8	125.9
$\mathcal{S}_2$-DB chain	2	32.9	117.8	49.8	49.2	177.8	74.9	65.3	238	100.0	81.5	297.7	125.6
$4\mathrm{NAF}_2$	3	40	200	0	60	300	0	80	400	0	100	500	0
$\mathcal{S}_3$-DB chain	3	30.7	117.5	49.7	45.7	177.5	74.8	60.6	237.8	99.8	75.6	297.3	125.4
$(2,2)$-DB-chain	4	36.8	119.2	49.8	54.7	179.3	74.8	72.6	239.4	100.1	90.5	299.0	125.7
$\mathcal{S}_4$-DB chain	4	28.9	117.3	49.6	43.2	177.3	74.7	57.6	237.6	99.8	71.5	297.1	125.4
$(3,3)$-DB-chain	6	35.3	119.3	49.5	52.2	179.4	74.6	69.2	239.5	99.6	86.1	299.2	125.2
$\mathcal{S}_6$-DB chain	6	27.3	117.4	49.4	40.6	177.3	74.5	54.0	237.6	99.6	67.1	297	125.3
$3\mathrm{NAF}_3$	8	36	0	126	54	0	189	72	0	252	90	0	315
$(4,4)$-DB-chain	8	34.2	119.3	49.3	50.5	179.5	74.2	67.0	239.6	99.3	83.5	299.3	125
$\mathcal{S}_8$-DB chain	8	25.9	117.2	49.3	38.5	177.1	74.4	51.2	237.4	99.5	63.6	296.9	125.2

Table 5. Parameters for various scalar multiplication methods on generic curves

Size		200 bits		300 bits		400 bits		500 bits	
	$\#\mathcal{P}$	N_{M}	Gain	N_{M}	Gain	N_{M}	Gain	N_{M}	Gain
$2\mathrm{NAF}_2$	0	2442.9	—	3669.6	—	4896.3	—	6122.9	—
Binary/ternary	0	2275.0	6.87%	3422.4	6.74%	4569.0	6.68%	5712.5	6.70%
DB-chain	0	2253.8	7.74%	3388.5	7.66%	4531.8	7.44%	5666.5	7.45%
$3\mathrm{NAF}_2$	1	2269.6	—	3409.6	—	4549.6	—	5689.6	—
$(1,0)$-DB-chain	1	2265.8	0.17%	3410.3	−1.98%	4562.3	−1.72%	5707.4	−1.69%
$(0,1)$-D B-chain	1	2226.5	1.90%	3343.2	1.95%	4471.0	1.73%	5590.4	1.74%
$\mathcal{S}_1$-DB chain	1	2150.4	5.25%	3238.1	5.03%	4326.3	4.91%	5418.1	4.77%
$2\mathrm{NAF}_3$	2	2384.8	—	3579.3	—	4773.8	—	5968.2	—
$(1,1)$-DB-chain	2	2188.6	8.23%	3285.5	8.21%	4390.0	8.04%	5487.7	8.05%
$\mathcal{S}_2$-DB chain	2	2106.5	11.67%	3174.1	11.32%	4243.6	11.11%	5314.8	10.95%
$4\mathrm{NAF}_2$	3	2165.6	—	3253.6	—	4341.6	—	5429.6	—
$\mathcal{S}_3$- DB chain	3	2078.1	4.04%	3132.8	3.71%	4189.8	3.50%	5248.5	3.34%
$(2,2)$-DB-chain	4	2158.2	—	3242.6	—	4333.1	—	5421.6	—
$\mathcal{S}_4$-DB chain	4	2056.7	—	3105.0	—	4156.1	—	5204.0	—
$(3,3)$-DB-chain	6	2139.4	—	3215.0	—	4291.7	—	5371.9	—
$\mathcal{S}_6$-DB chain	6	2036.3	—	3074.3	—	4115.4	—	5155.1	—
$3\mathrm{NAF}_3$	8	2236.2	—	3355.8	—	4475.4	—	5595.0	—
$(4,4)$-DB-chain	8	2125.4	4.95%	3192.2	4.88%	4264.1	4.72%	5340.5	4.55%
$\mathcal{S}_8$-DB chain	8	2019.3	9.70%	3049.8	9.12%	4084.3	8.74%	5116.8	8.55%

Table 6. Complexity of various extended DBNS methods for generic curves and gain with respect to window NAF methods having the same number of precomputations

Size		200 bits			300 bits			400 bits			500 bits		
	$\#\mathcal{P}$	ℓ	a_1	b_1	ℓ	a_1	b_1	ℓ	a_1	b_1	ℓ	a_1	b_1
2NAF$_2$	0	66.7	200	0	100	300	0	133.3	400	0	166.7	500	0
Binary/ternary	0	46.1	90.7	68.1	69.2	136.4	102.2	91.9	182.6	136.3	114.4	228.0	170.7
DB-chain	0	45.0	98.5	63.2	67.1	148.7	94.4	89.4	198.5	126.2	110.5	248.9	157.5
3NAF$_2$	1	50	200	0	75	300	0	100	400	0	125	500	0
$(1,0)$-DB-chain	1	42.8	98.8	62.9	63.8	149.0	94.1	84.7	198.8	125.9	106.1	249.2	157.1
$(0,1)$-DB-chain	1	49.0	98.5	63.2	73.4	148.7	94.4	98.8	198.5	126.2	123.3	248.9	157.5
$\mathcal{S}_1$-DB chain	1	36.6	97.7	62.8	54.4	148	93.9	72.2	197.8	125.8	89.8	248.1	157.0
2NAF$_3$	2	50.4	0	126	75.6	0	189	100.8	0	252	126	0	315
$(1,1)$-DB-c hain	2	40.6	98.8	62.9	60.4	149	94.1	80.8	198.8	125.9	100.9	249.2	157.1
$\mathcal{S}_2$-DB chain	2	32.5	97.6	62.6	48.1	147.7	93.9	64.0	197.6	125.6	79.6	247.9	156.9
4NAF$_2$	3	40	200	0	60	300	0	80	400	0	100	500	0
$\mathcal{S}_3$-DB chain	3	30.3	97.1	62.5	44.8	147.6	93.7	59.5	197.2	125.4	74.2	247.5	156.9
$(2,2)$-DB-chain	4	37.3	99.0	62.6	55.5	149.2	93.8	74	199	125.7	92.3	249.3	156.9
$\mathcal{S}_4$-DB chain	4	28.7	97.1	62.4	42.5	147.5	93.6	56.4	197.1	125.4	70.2	247.4	156.7
$(3,3)$-DB-chain	6	36.2	99.1	62.2	53.8	149.3	93.6	71.8	199.2	125.2	89.6	249.4	156.7
$\mathcal{S}_6$-DB chain	6	27.0	97.1	62.2	39.9	147.4	93.4	52.7	197.1	125.2	65.6	247.4	156.5
5NAF$_2$	7	33.3	200	0	50	300	0	66.7	400	0	83.3	500	0
3NAF$_3$	8	36	0	126	54	0	189	72	0	252	90	0	315
$(4,4)$-DB-chain	8	34.9	99.2	62	52.1	149.4	93.1	69.5	199.3	125.0	86.8	249.5	156.3
$\mathcal{S}_8$-DB chain	8	25.6	96.8	62.2	37.8	147.3	93.3	50.0	196.9	125.1	62.1	247.2	156.5

Table 7. Parameters for various scalar multiplication methods on special curves

Size		200 bits		300 bits		400 bits		500 bits	
	$\#\mathcal{P}$	N_{M}	Gain	N_{M}	Gain	N_{M}	Gain	N_{M}	Gain
2NAF$_2$	0	2282.9	—	3429.6	—	4576.3	—	5722.9	—
Binary/ternary	0	1930.2	15.45%	2904.3	15.32%	3877.8	15.26%	4847.2	15.30%
DB-chain	0	1927.8	15.55%	2897.1	15.53%	3870.4	15.43%	4830.3	15.60%
3NAF$_2$	1	2109.6	—	3169.6	—	4229.6	—	5289.6	—
$(1,0)$-DB-chain	1	1904.0	9.75%	2861.9	9.71%	3821.6	9.65%	4783.6	9.57%
$(0,1)$-DB-chain	1	1968.7	6.68%	2962.2	6.54%	3968.9	6.16%	4963.1	6.17%
$\mathcal{S}_1$-DB chain	1	1829.8	13.26%	2754.1	13.11%	3681.6	12.96%	4604.3	12.96%
2NAF$_3$	2	1880.8	—	2823.3	—	3765.8	—	4708.2	—
$(1,1)$-DB-chain	2	1881.5	-1.96%	2826.6	-1.88%	3781.0	-1.60%	4729.2	-1.55%
$\mathcal{S}_2$-DB chain	2	1784.1	5.14%	2685.9	4.87%	3592.8	4.59%	4494.9	4.53%
4NAF$_2$	3	2005.6	—	3013.6	—	4021.6	—	5029.6	—
$\mathcal{S}_3$- DB chain	3	1757.1	12.39%	2647.9	12.13%	3541.0	11.95%	4434.8	11.83%
$(2,2)$-DB-chain	4	1845.4	—	2773.8	—	3708.4	—	4638.5	—
$\mathcal{S}_4$-DB chain	4	1738.3	—	2621.5	—	3506.8	—	4391.0	—
$(3,3)$-DB-chain	6	1830.8	—	2753.9	—	3682.0	—	4608.4	—
$\mathcal{S}_6$-DB chain	6	1718.3	—	2592.1	—	3467.1	—	4341.5	—
3NAF$_3$	8	1732.2	—	2599.8	—	3467.4	—	4335.0	—
$(4,4)$-DB-chain	8	1815.0	-5.22%	2731.9	-6.92%	3656.5	-6.55%	4575.9	-6.44%
$\mathcal{S}_8$-DB chain	8	1701.9	1.75%	2568.6	1.20%	3436.1	0.90%	4302.7	0.75%

Table 8. Complexity of various extended DBNS methods for special curves and gain with respect to window NAF methods having the same number of precomputations