On multi-exponentiation in cryptography

Roberto M. Avanzi (IEM Essen)

October 9, 2002

Abstract

We combine an algorithm for computing multiple exponentiations with different representations of the exponents. Some variants rely on the fact that inversion of group elements is fast. These algorithms are particularly suitable for computing double or triple exponentiations in rational point subgroups of elliptic or hyperelliptic curves and perform efficiently in memory constrained environments, especially with exponents in the range from 160 to 256 bits. These methods can also be used for computing *single* exponentiations in groups which admit an automorphism σ satisfying a monic equation of small degree over the integers, such as trace zero varieties. We outline a few applications of these algorithms. By construction, such methods provide good resistance against side channel attacks (at least Simple Power Attacks).

Contents

1 Introduction

A common operation in public-key cryptographic protocols is the computation of the product of powers of two [ANSI] or three [El] elements of a group, the first case being the more common and the paramount example being digital signatures. Furthermore, in some algebraic structures the computation of a single exponentiation can be reduced to such a product: If a cyclic group G admits an automorphism σ satisfying a monic equation over the integers of degree d then g^e can be computed as $g^{e_0} \cdot \sigma(g)^{e_1} \cdots \sigma^{d-1}(g)^{e_{d-1}}$ for suitable integers $e_0, \ldots, e_{d-1}$ which in many practical instances have size $O(e^{1/d})$ (see [GaLaVa, SiCiQu]). In this context, too, the cases $d = 2$ and $d = 3$ are of particular practical relevance because of trace zero varieties and XTR subgroups.

Clearly, such computations can be performed by computing the various powers separately and then multiplying the results together. However, if the intermediate results are not needed elsewhere, one can do much better. Shamir suggested [El] a simple yet effective trick for speeding up this kind of operations which scans the bits of the exponents simultaneously. We extend the trick by using sliding windows across the representations of all the exponents simultaneously: To our knowledge the first written report of this obvious extension is [YeLaLe]. It has been recalled in [Mö] where it is compared to a different multi-exponentiation algorithm called *interleaved exponentiation.* So far, this extension to Shamir's trick has been applied only to the usual binary representation of the exponents.

One of our main concerns is reducing the running time of multi-exponentiation while keeping the memory requirements as small as possible. This is crucial for example when implementing cryptographic protocols on smart cards.

Our results can be summarized as follows:

(1) We consider the algorithm from [YeLaLe] for performing double and triple exponentiations in groups where inversion is slow and the inverses of the base elements are not previously given. This algorithm is faster than other methods such as interleaved exponentiation for exponents up to at least 256 bits.

(2) We propose and analyze variants of the algorithm which are better suited to groups where inversion is cheap. To take advantage of this we rewrite the exponents using signed digit representations, which were first introduced in [Bo]: In particular we consider the *non-adjacent form* [Re, MoOl] and a new representation of pairs of integers due to Solinas [So3].

Let n be the bit-lenght of the exponents. Our best algorithm performs double exponentiations with $64 < n \leq 354$ by $9+3n/8$ multiplications on average and about n squarings (in fact, slightly less) and uses 12 precomputed values (including the bases). This is either slightly better in performance than interleaved exponentiation or of comparable time complexity while having smaller memory requirements. For $n \leq 64$ it reverts to Solinas' method requiring $1 + n/2$ multiplications on average, n squarings and using 4 precomputed values.

In particular, this shows the flexibility and adaptability of Solinas' recently proposed representation.

(3) To our knowledge Theorem 2.6 also complements existing literature in the case of single exponentiations.

(4) We compare our variants of the multi-exponentiation method from [YeLaLe] to the interleaved multi-exponentiation, thus extending Möller's analysis.

We now proceed with the description of the algorithm from [YeLaLe].

Let G be a commutative group of order $q \approx 2^n$ and d a (small) integer. Suppose we are given elements $g_1, \ldots, g_d \in G$ and integers $e_1, \ldots, e_d$ and want to compute $x := \prod_{i=1}^{d} g_i^{e_i}$.

Write

$$e_i = \sum_{j=0}^{n-1} e_{i,j}\, 2^j \tag{1}$$

with $e_{i,j} \in \{0, \pm 1\}$. The coefficients $e_{i,j}$ are called bits: *unsigned bits* if the value -1 is not allowed, *signed bits* otherwise. In this paper, as it is now customary, $\bar{1}$ means -1 in signed bit expansions of integers.

For the moment we assume that the chosen representation is the unsigned binary one. Shamir's trick is as follows: First precompute the 2^d values $\prod_{i=1}^d g_i^{\{0,1\}}$. Then put $x = \prod_{i=1}^d g_i^{e_{i,n-1}}$ by one table look-up. Finally, for $j = n-2, \ldots, 1, 0$, replace x by $x^2 \cdot \prod_{i=1}^d g_i^{e_{i,j}}$ by one squaring, one table look-up and one multiplication.

It is easily seen that Shamir's method requires $2^d - d - 1$ multiplications to prepare the table, n squarings and on average $(1 - 2^{-d})n$ multiplications, 2^{-d} being the probability that for a fixed j, $e_{i,j}$ is 0 for all $i = 1, 2, \ldots, d$. If the exponents are written in a signed binary representation, the table $\mathcal{E}$ can be formed from the products of the form $\prod_{i=1}^d g_i^{k_i}$ with $k_i \in \{0, \pm 1\}$. However, if the cost of an inversion in the group G is negligible, which is usually the main reason for adopting a signed binary representation, one only needs a half of those values, *i.e.* those where the first nonzero k_i equals 1. Then some products are replaced by divisions.

This method can be improved by means of sliding windows [Kn] in the same way as the square-and-multiply method. The resulting algorithm is as follows:

Algorithm 1.1 Multi-exponentiation with parallel sliding windows

Input: A window size w, integers $e_1, \ldots, e_d$ as in (1) and a set $\mathcal{E}$ of precomputed elements of the group G of the form $\prod_{i=1}^d g_i^{k_i}$ including $g_1, \ldots, g_d$ *(the set $\mathcal{E}$ depends on w and on the chosen representation for the integers e_i: see Remarks 1.2 (3–4) for examples)*

Output: $\prod_{i=1}^d g_i^{e_i}$

```
Step 1.  $t \leftarrow n$ and $x \leftarrow 1 \in G$
Step 2.  if ($e_{i,t-1} = 0$ for $i = 1, 2, \ldots, d$) then {
             (a)  $t \leftarrow t - 1$ and $x \leftarrow x^2$
         } else {
             (b)  if $t \geq w$ then $t \leftarrow t - w$ else { $w \leftarrow t$ and $t \leftarrow 0$ }
             (c)  for $i = 1, 2, \ldots, d$ do $f_i \leftarrow \sum_{j=0}^{w-1} e_{i,t+j} 2^j$
             (d)  Let $s$ be the largest integer $s \geq 0$ such that $2^s | f_i$ for all $i$
             (e)  for $i = 1, 2, \ldots, d$ do $f_i \leftarrow f_i / 2^s$
             (f)  $x \leftarrow \left(x^{2^{w-s}} \cdot \prod_{i=1}^d g_i^{f_i}\right)^{2^s}$
         }
Step 3.  if $t = 0$ then return $x$ else goto Step 2
```

Remarks 1.2 (1) *In the case $d = 1$ the above algorithm is the usual sliding window exponentiation algorithm.*

(2) *At the beginning of Step 2 (c) f_i is the integer represented by a string of w consecutive bits from the exponent e_i. Now s is the largest non-negative integer such that $e_{i,t+u} = 0$ for all i and all u with $0 \le u \le s$. The* normalisation *Step 2 (e) is performed such that at least one of the integers f_i is odd, in order to reduce the number of elements of $\mathcal{E}$ without impacting the total number of operations done in Step 2 (f).*

(3) *If the chosen representation for the exponents is the standard binary one, then the set $\mathcal{E}$ should consist of all elements of the form $\prod_{i=1}^d g_i^{k_i}$ such that $0 \le k_i < 2^w$ and at least one of the k_i is odd. Then Step 2 (f) is performed by means of one table look-up, one multiplication and w squarings.*

Further, note that, regardless of the representation chosen, in Step 2 (f) the first time it is $x = 1$, so one multiplication can be saved and only s squarings are needed.

(4) *The changes to Algorithm 1.1 required to work with the NAF are straightforward. We now assume that inversion in the group is very fast or for free. The largest integer representable by a w-bit number in NAF is $(10\ldots01)_2$ for odd w and $(10\ldots10)_2$ for even w, and it is easy to see that this number is $T_w = \frac{2^{w+2}-3-(-1)^w}{6}$. Hence, there are*

$$I_w = \frac{2^{w+2} - (-1)^w}{3}$$

integers in the interval $[-T_w, \ldots, T_w]$. To form the set $\mathcal{E}$ it suffices to precompute all elements of the form $\prod_{i=1}^d g_i^{k_i}$ such that $|k_i| \le T_w$ for $i = 1, 2, \ldots, d$, at least one of the k_i is odd and the first nonzero element in the sequence $k_1, k_2, \ldots, k_p$ is positive. *There are $(I_w^d - I_{w-1}^d)/2$ elements in $\mathcal{E}$. Finally, in Step 2 (f) if the first nonzero f_i is positive we compute $x = \left(x^{2^{w-s}} \cdot \prod_{i=1}^d g_i^{f_i}\right)^{2^s}$ otherwise we compute $x = \left(x^{2^{w-s}} / \prod_{i=1}^d g_i^{-f_i}\right)^{2^s}$.*

The above algorithm will be analysed in Section 2 in a few variants: this is the main part of this paper. In Section 3 the optimal parameters for Algorithm 1.1 will be discussed and the resulting time and space complexities will be compared against those of interleaved exponentiation. Next, some applications will be outlined. A remark about security of multi-exponentiation algorithms under side-channed attacks concludes the paper.

Acknowledgements. The author has been supported by the AREHCC project (Advanced Research in Elliptic and Hyperelliptic Curve Cryptography, see: `http://www.arehcc.com`) funded by the European Commission (Fifth Framework Program IST – 2001 – 32613).

This paper would not exist, at least not in its present form, without Professor Gerhard Frey's steady encouragement and support. The author is grateful to Tanja Lange who drew the author's attention to Solinas' work and proofread the manuscript. Many thanks also to Arjen Lenstra for kindly providing a reprint of [YeLaLe].

Some computations have been performed using the `maple` computer algebra system [ChGeGo$^+$].

2 Complexity analysis

In this section we compute the complexity of Algorithm 1.1 for different choices of the representation of the exponents and with some additional restrictions.

Definition 2.1 *A* column *is defined as a d-tuple of digits $e^{(t)} = (e_{1,t}, \ldots, e_{d,t})$ of the representation of integers (1) and the ordered sequence $e^{(n-1)}, e^{(n-2)}, \ldots, e^{(0)}$ of such columns is called* joint representation *of the d exponents $e_1, \ldots, e_d$.*

If $e^{(n-1)} \neq 0$ then the joint representation is said to be proper *and n is its* length*.*

The number of nonzero colums in the joint representation is called its Hamming weight*, and its* density *is the ratio of the Hamming weight to the length.*

For simplicity we require that the joint representation of the exponents $e_1, \ldots, e_d$ is proper. This implies that at the first iteration of Step (2), substeps (b)–(f) are always performed. To evaluate the number of squarings in the iterations one should not consider those which can be avoided in the first iteration, which are w minus the expected first value of s.

Algorithm 1.1 scans the joint representation of the d exponents $e_1, \ldots, e_d$ one column at a time, starting with the column formed by the most significant digits in the chosen representation. Step 2 is iterated until the joint representation has been scanned completely. At each iteration one column is read and the algorithm enters in one of two possible distinct states:

$\mathcal{S}_0$. A zero column is found, so the scanning advances by one column (Step 2 (a)).

$\mathcal{S}_1$. A nonzero column is found and the scanning advances by w columns (Steps 2 (b)–(f)).

The number of multiplications (excluding squarings) performed by the algorithm equals the number of times we are in the second state.

Let π be the probability that the column read in Step 2 is zero. After m iterations, the expected number of columns scanned by the scanning process is $(\pi + w(1-\pi))m$. Suppose that for some m this number is n. The number of multiplications performed by Algorithm 1.1 in Step 2 (d) is then $(1-\pi)m - 1$ (remember that the first multiplication can be replaced by an assignment) *i.e.*

$$n \cdot \frac{1-\pi}{\pi + w(1-\pi)} - 1. \qquad (2)$$

This is, with some adaptations, the approach followed in the next two subsections.

Before dealing with the particular instances of Algorithm 1.1 we are interested in, we make a further definition and a remark.

Definition 2.2 *Let $e = \sum_{j=0}^{n-1} e_j\, 2^j$ be an integer. We say that an algorithm scans (generates, rewrites...) the* bits *e_j* right-to-left *(resp.* left-to-right*) if it scans (generates, rewrites...) them from the least significant ones to the most significant ones, i.e. first e_0, then e_1, e_2, etc. (resp. from the most significant ones to the least significant ones, i.e. first e_{n-1}, then e_{n-2}, and so on).*

Similar definitions hold for algorithms which deal with the colums of a joint representation of several integers.

Remark 2.3 *Algorithm 1.1 processes the columns of the chosen joint representation of the exponents left-to-right. However most recoding algorithms for producing signed binary* *representations, such as the Reitwiesner's algorithm [Re] NAF and Solinas' own algorithm for the Joint Sparse Form, rewrite the exponents right-to-left. This is a general problem with window methods. In most situations recoding and (multi-)exponentiation cannot be interleaved, and the recoded representations must be stored explicitly.*

2.1 Unsigned binary inputs

Here we assume that the exponents are written in (unsigned) base 2, *i.e.* that $e_{i,j} \in \{0,1\}$.

As noted in Remark 1.2 (2) in this case the set $\mathcal{E}$ should consist of all elements of the form $\prod_{i=1}^{d} g_i^{k_i}$ such that $0 \leq k_i < 2^w$ and at least one of the k_i is odd. This set has cardinality $2^{wd} - 2^{(w-1)d}$. Half of the powers of the base elements g_i can be computed via squarings and all other elements via products.

The bits in each representation are assumed to be zero or one with equal probability and independent from the adjacent bits, so $\pi = 2^{-d}$.

To evaluate the number of squarings in the main loop of the algorithm we must determine the expected value of s at the first iteration. As all the bits are independent from each other, $s \geq u$ with $1 \leq u < w$ with probability 2^{-ud}. Hence the expected value of s is $\sum_{u=1}^{w-1} 2^{-ud} = \frac{1-2^{-d(w-1)}}{2^d-1}$.

We have thus proved the following result:

Theorem 2.4 *Suppose that in Algorithm 1.1 the unsigned binary representation is used for the exponents and that their joint representation has length n.*

Then the set $\mathcal{E}$ has cardinality $2^{wd} - 2^{(w-1)d}$ and requires $2^{wd} - 2^{(w-1)d} - d$ operations to be computed: of these at least $d(2^{w-1}-1)$ can be assumed to be squarings.

The expected number of multiplications in the algorithm is $n\frac{1}{w+(2^d-1)^{-1}} - 1$ and that of the squarings is $n - w + \frac{1-2^{-d(w-1)}}{2^d-1}$.

Remark 2.5 *In the case $w = d = 2$, the set $\mathcal{E}$ consists of the values $g_1^a g_2^b$ with $0 \leq a, b \leq 3$ and at least one of a, b odd. To determine them one has to compute and store g_1^2 and g_1^3, as well as g_2^2 and g_2^3. This requires 2 squarings and 2 multiplications. Computing the remaining 8 values requires 8 further multiplications.*

2.2 Using the NAF

A *non-adjacent (binary) form*, or representation (abbreviated as NAF) is a signed binary representation of an integer $e = \sum_{j=0}^{n-1} e_j 2^j$ with $e_j \in \{0, \pm 1\}$ and $e_j e_{j-1} = 0$. Each integer admits a NAF, which is uniquely determined and is the signed binary representation of minimal Hamming weight and of expected density 1/3 (see [MoOl] and [ArWh] for proofs of these facts).

Similar situations have been considered already but only for single NAF's (*i.e.* $d = 1$) and not joint representations. In the paper [EğKo], of which we use some arguments in this subsection, windows are not allowed to slide and only the probability that a certain fixed window is zero is considered there. The adoption of sliding windows leads to an algorithm of better complexity. Hence, even in the case $p = 1$ our results will complement existing literature.

Theorem 2.6 *Suppose that in Algorithm 1.1 the exponents are input in NAF, and that their joint representation is n bits long.*

The set $\mathcal{E}$ has cardinality $(I_w^d - I_{w-1}^d)/2$ where $I_w = \frac{2^{w+2}-(-1)^w}{3}$.

The number of squarings in the main loop of the algorithm is between $n-w$ and $n-1$, with an heuristically expected value $n - w + \left(\frac{4}{3}\right)^d \frac{1-2^{-d(w-1)}}{2^d-1}$. In the cases $d = 1, 2$ and 3 respectively, the expected number of multiplications is $n \cdot \frac{1-\pi^{(d)}}{w-(w-1)\pi^{(d)}} - 1$ where

$$\begin{aligned} \pi^{(1)} &= \frac{4\,(2^w-(-1)^w)}{7\cdot 2^w - 4\cdot(-1)^w}\,, \quad \pi^{(2)} = \frac{16\,(4^w-1)}{43\cdot 4^w + 24\cdot(-2)^w - 16} \quad \text{and} \\ \pi^{(3)} &= \frac{64\,(2^w+(-1)^w)(8^w-(-1)^w)}{253\cdot 16^w + 397\cdot(-8)^w + 324\cdot 4^w + 80\cdot(-2)^w - 64}. \end{aligned} \tag{3}$$

In particular for $d = 1$ the expected number of multiplications is $n \cdot \frac{1}{w+\frac{4}{3}(1-(-\frac{1}{2})^w)} - 1$.

Remark 2.7 *In the case $w = d = 2$, the set $\mathcal{E}$ consists of the values $g_1^a g_2^b$ with either $0 < a \leq 2$ and $-2 \leq b \leq 2$ where at least one of a, b odd, or $a = 0$ and $b = 1$. A chain for obtaining all the elements to be precomputed is*

$$\{\ g_1,\quad g_2,\quad g_1g_2,\quad g_1g_2^{-1},\quad g_1g_2^2,\quad g_1g_2^{-2},\quad g_1^2g_2,\quad g_1^2g_2^{-1}\ \}.$$

This requires 6 multiplications or multiplications with the inverse.

The remainder of this subsection is devoted to the proof of Theorem 2.6.

Definition 2.8 *A joint representation of integers in NAF will be called a* joint NAF.

Definition 2.9 *Let $\mathbf{e} = (e_1, \ldots, e_d)$ be a d-tuple of n-bit integers so that (1) is proper. The* bit-reversing $\widehat{\mathbf{e}}$ *of $\mathbf{e}$ is the d-tuple formed by the numbers $\widehat{e}_i = \sum_{j=0}^{n-1} e_{i,(n-1)-j}\, 2^j$.*

In order to avoid ambiguity, we only define bit-reversing for proper joint representations. The mapping which associates to a proper joint NAF its bit-reversing induces a bijection between the set of proper joint NAF's of d integers of n bits and the set of joint NAF's (not necessarily proper) of d integers of n bits, at least one of the integers being odd. *Hence the expected number of windows made by Algorithm 1.1 on n-bit proper joint NAF's of d integers equals the expected number of windows formed by a sliding window algorithm which scans from right to left joint NAF's of d integers of n bits, at least one odd.* The parity condition amounts to the fact that at the first iteration a nonzero column is found, exactly as in the original algorithm.

Consequently we will consider an algorithm which forms sliding windows on joint NAF's from left to right, and we will model it as a Markov chain: At each iteration one column is read and the algorithm enters in one of $d+1$ possible distinct states, defined by the number of nonzero entries in the colums:

$\mathcal{S}_0'$. A zero column is found, so the scanning advances by one column.

$\mathcal{S}_k'$ (for $1 \leq k \leq d$). A column is found with exactly k nonzero entries and the scanning advances by w columns. (∗)

To determine the transition probability from state $\mathcal{S}'_\ell$ to state $\mathcal{S}'_k$ we need a few preliminary results.

We begin with a review of Reitwiesner's algorithm for recoding the unsigned binary representation of a number $e = \sum_{j=0}^{n-1} b_j 2^j$ into a NAF $\sum_{j=0}^{n} e_j 2^j$. For $j = 0, 1, \ldots, n-1$, the digit e_j of the NAF is a function of the values of b_{j+1}, b_j and of the j-th carry c_j, which is equal to one if the NAF of the truncated number $e = \sum_{i=0}^{j-1} b_i 2^i$ is one bit longer than its unsigned binary representation. At the beginning $c_0 = 0$. The recoding is then done as shown in Table 1 – where we also write the admissible following state according to the value of e_{i+2} and the corresponding output – and at the end $e_n = c_{n-1}$. If $e_n \neq 0$ then the NAF is longer than the original representation. Since in the unsigned binary

State	Input		Output		Next State (and e_{i+1})	
	$(b_{i+1}\, b_i)_2$	c_i	e_i	c_{i+1}	if $b_{i+2}=0$	if $b_{i+2}=1$
s_0	$(0\,0)$	0	0	0	s_0 (0)	s_4 (0)
s_1	$(0\,0)$	1	1	0	s_0 (0)	s_4 (0)
s_2	$(0\,1)$	0	1	0	s_0 (0)	s_4 (0)
s_3	$(0\,1)$	1	0	1	s_1 (1)	s_5 ($\bar{1}$)
s_4	$(1\,0)$	0	0	0	s_2 (1)	s_6 ($\bar{1}$)
s_5	$(1\,0)$	1	$\bar{1}$	1	s_3 (0)	s_7 (0)
s_6	$(1\,1)$	0	$\bar{1}$	1	s_3 (0)	s_7 (0)
s_7	$(1\,1)$	1	0	1	s_3 (0)	s_7 (0)

Table 1: States of Reitwiesner's Algorithm

representation each bit assumes a value of zero or one with equal probability and there is no dependency between any two bits, it is clear that all admissible transitions from a state s_ℓ to a state s_k occur with probability $\frac{1}{2}$. It is straightforward to write down the corresponding transition probability matrix P. The resulting limiting probabilities for the states $s_0, \ldots, s_7$ are thus [EğKo] given by the vector

$$\mathbf{v} = \left[\frac{1}{6}, \frac{1}{12}, \frac{1}{12}, \frac{1}{6}, \frac{1}{6}, \frac{1}{12}, \frac{1}{12}, \frac{1}{6}\right]$$

whose components add up to 1 and which satisfies $P \cdot \mathbf{v}^\perp = \mathbf{v}^\perp$. (Here the symbol $\perp$ denotes matrix transposition.) From this it is immediate, upon summing the probabilities for states s_1, s_2, s_5 and s_6, to obtain the known result that the expected Hamming weight of a NAF is $\frac{1}{3}$. The fact which is more relevant to us here is that states s_0, s_3, s_4 and s_7, which all output a zero, occur with equal probabilities, and that in two cases another zero will be output by the next state, whereas in the other two a nonzero bit will be output. We have thus proved the following lemma.

Lemma 2.10 *The probability that in a NAF the digit immediately to the left of a* 0 *is another* 0 *is* $\frac{1}{2}$ *and that it is* 1 *or* -1 *is in each case* $\frac{1}{4}$.

We now generalize this by determining the probabilities that a bit $e_{j,i+w}$ which is w places to the left of $e_{j,i}$ is zero or one, depending on the value of $e_{j,i}$ and w.

Lemma 2.11 *If $e_{j,i} = 0$, then $e_{j,i+w} = 0$ with probability $\pi_{w,0}$ and $e_{j,i+w} \neq 0$ with probability $\pi_{w,*}$, where*

$$\pi_{w,0} = \frac{2^{w+1} + (-1)^w}{3 \cdot 2^w} \quad \textit{and} \quad \pi_{w,*} = 1 - \pi_{w,0} = \frac{1}{2}\pi_{w-1,0} = \frac{2^w - (-1)^w}{3 \cdot 2^w}. \tag{4}$$

Since a nonzero bit is always followed by a zero, we also have that if $e_{j,i} \neq 0$, then $e_{j,i+w} = 0$ with probability $\pi_{w-1,0}$ and $e_{j,i+w} \neq 0$ with probability $\pi_{w-1,*}$.

Proof. Clearly $\pi_{w,0} + \pi_{w,*} = 1$. By Lemma 2.10 we have $\pi_{1,0} = \pi_{1,*} = \frac{1}{2}$ and

$$\begin{cases} \pi_{i+1,0} = \pi_{i,*} + \frac{1}{2}\pi_{i,0} = 1 - \frac{1}{2}\pi_{i,0} \\ \pi_{i+1,*} = \frac{1}{2}\pi_{i,0}. \end{cases}$$

Now (4) follows easily by induction. □

We are now in the position to model the right-to-left scanning process as a Markov chain with the states $\mathcal{S}'_0, \ldots, \mathcal{S}'_d$ defined above in $(*)$. Denote by $\tau_{\ell,k}$ the transition probability from state $\mathcal{S}'_\ell$ to state $\mathcal{S}'_k$.

Suppose that a zero column is read. Then no window is being formed and at the next iteration the scanning algorithm will read the next column to the left. The probability $\tau_{0,k}$ that this column contains exactly k nonzero entries is $\binom{d}{k}\frac{1}{2^k}$.

On the other hand suppose that a column $\mathbf{c}$ with exactly $\ell \neq 0$ nonzero entries has been read. The bit-reversing of the numbers represented by this column and the next $w-1$ columns at its left are the exponents $f_1, \ldots, f_d$ in Step 2 (c). The next column checked by the right-to-left scanning process, say $\mathbf{c}'$, will be then that which is exactly w places to the left of $\mathbf{c}$. Now $\tau_{\ell,k}$ is the probability that $\mathbf{c}'$ has exactly k nonzero entries (where $0 \le k \le d$). For some integer r, in exactly r of the positions occupied by the ℓ nonzero digits in $\mathbf{c}$ there will be nonzero bits in the respective positions in $\mathbf{c}'$, and in the positions of the remaining $\ell - r$ nonzero bits in $\mathbf{c}$ there will be zeros in $\mathbf{c}'$. Therefore, to exactly $k - r$ of the zero bits in $\mathbf{c}$ will correspond nonzero bits in $\mathbf{c}'$, and to the other $d - \ell - (k - r)$ zeros of $\mathbf{c}$ will correspond zeros in $\mathbf{c}'$. Finally

$$\begin{aligned}
\tau_{\ell,k} &= \sum_{\substack{r\,:\,0 \le r \le \ell \\ 0 \le k-r \le d-\ell}} \binom{\ell}{r}\binom{d-\ell}{k-r} \pi_{w-1,*}^{r}\pi_{w-1,0}^{\ell-r}\pi_{w,*}^{k-r}\pi_{w,0}^{d-\ell-(k-r)} \\
&= \sum_{\substack{r\,:\,0 \le r \le \ell \\ k+\ell-d \le r \le k}} \binom{\ell}{r}\binom{d-\ell}{k-r} \left(1 - 2\pi_{w,*}\right)^{r} 2^{\ell-r}\pi_{w,*}^{\ell-r}\pi_{w,*}^{k-r}\left(1 - \pi_{w,*}\right)^{d-\ell-(k-r)} \\
&= \sum_{r=\max\{0,k+\ell-d\}}^{\min\{\ell,k\}} \binom{\ell}{r}\binom{d-\ell}{k-r} 2^{\ell-r}\pi_{w,*}^{\ell+k-2r}\left(1 - \pi_{w,*}\right)^{(d-\ell-k)+r}\left(1 - 2\pi_{w,*}\right)^{r}.
\end{aligned}$$

Put

$$T_d = (\tau_{\ell,k})_{\ell,k=0}^{d} = \begin{pmatrix} 1/2^d & \tau_{1,0} & \tau_{2,0} & \cdots & \tau_{d,0} \\ \binom{d}{1}/2^d & \tau_{1,1} & \tau_{2,1} & \cdots & \tau_{d,1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \binom{d}{d-1}/2^d & \tau_{1,d-1} & \tau_{2,d-1} & \cdots & \tau_{d,d-1} \\ 1/2^d & \tau_{1,d} & \tau_{2,d} & \cdots & \tau_{d,d} \end{pmatrix}.$$

The limiting probabilities $\sigma_0, \ldots, \sigma_d$ of the algorithm being in state $\mathcal{S}'_0, \ldots, \mathcal{S}'_d$ respectively satisfy $\sum_{k=1}^{d} \sigma_k = 1$ and $T_d \cdot (\sigma_0 \, \cdots \, \sigma_d)' = (\sigma_0 \, \cdots \, \sigma_d)'$. Hence, upon putting

$$U_d = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ d/2^d & \tau_{1,1}-1 & \tau_{2,1} & \cdots & \tau_{d,1} \\ \binom{d}{2}/2^d & \tau_{1,2} & \tau_{2,2}-1 & \cdots & \tau_{d,2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ d/2^d & \tau_{1,d-1} & \tau_{2,d-1} & \cdots & \tau_{d,d-1} \\ 1/2^d & \tau_{1,d} & \tau_{2,d} & \cdots & \tau_{d,d}-1 \end{pmatrix},$$

we have $U_d \cdot (\sigma_0 \, \cdots \, \sigma_d)^{\perp} = (1, 0, \ldots, 0)^{\perp}$. Hence, provided that U_d is invertible, $(\sigma_0 \, \cdots \, \sigma_d)^{\perp} = U_d^{-1} \cdot (1, 0, \ldots, 0)^{\perp}$ and in particular σ_0 is the value in the top left corner of U_d^{-1}.

We are interested in U_d only in the cases $d = 1, 2$ and 3. Upon putting $\alpha = 2^w$ and $\beta = (-1)^w$ we obtain

$$U_1 = \begin{pmatrix} 1 & 1 \\ \frac{1}{2} & \frac{\alpha+2\beta}{3\alpha} - 1 \end{pmatrix}, \qquad U_2 = \begin{pmatrix} 1 & 1 & 1 \\ \frac{1}{2} & \frac{4\alpha^2+\alpha\beta+4\beta^2}{9\alpha^2} - 1 & \frac{4(\alpha-\beta)(\alpha+2\beta)}{9\alpha^2} \\ \frac{1}{4} & \frac{(\alpha-\beta)(\alpha+2\beta)}{9\alpha^2} & \frac{(\alpha+2\beta)^2}{9\alpha^2} - 1 \end{pmatrix} \quad \text{and}$$

$$U_3 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ \frac{3}{8} & \frac{(2\alpha+\beta)(2\alpha^2-\alpha\beta+2\beta^2)}{9\alpha^3} - 1 & \frac{4(\alpha^3-\beta^3)}{9\alpha^3} & \frac{4(\alpha-\beta)^2(\alpha+2\beta)}{9\alpha^3} \\ \frac{3}{8} & \frac{2(\alpha^3-\beta^3)}{9\alpha^3} & \frac{(\alpha+2\beta)(2\alpha^2-\alpha\beta+2\beta^2)}{9\alpha^3} - 1 & \frac{2(\alpha-\beta)(\alpha+2\beta)^2}{9\alpha^3} \\ \frac{1}{8} & \frac{(\alpha-\beta)^2(\alpha+2\beta)}{27\alpha^3} & \frac{(\alpha-\beta)(\alpha+2\beta)^2}{27\alpha^3} & \frac{(\alpha+2\beta)^3}{27\alpha^3} - 1 \end{pmatrix}.$$

The above matrices have been written down using simple `maple` [ChGeGo$^+$] code. Within the same software environment it is immediate to verify that for $d = 1, 2$ and 3 the matrix U_d is indeed invertible and to compute σ_0, *i.e.* the value of π in the introductory part of this section. We thus obtain the values $\pi = \pi^{(d)}$ given in equation (3), Theorem 2.6.

To estimate the value of s at the first iteration of the main loop, we proceed heuristically. [EğKo, Theorem 1] states that the probability that a length u bit section of a number in NAF is zero is $\frac{4}{3}\left(\frac{1}{2}\right)^u$. For $u = 1, \ldots, w-1$ we apply this result to the u least significant bits used to form each of the integers $f_1, \ldots, f_d$ in Step 2 (c) at the first iteration of the loop of Algorithm 1.1. We then proceed as in the proof of Theorem 2.4, the only difference consisting in the multiplicative factor $\left(\frac{4}{3}\right)^d$. $\square$

2.3 Using the JSF

The Joint Sparse Form has been introduced by Solinas [So3] to make Shamir's trick more effective for elliptic curves. It applies however to all groups where inversion is essentially for free. It has been defined only for *pairs* of integers: accordingly we will restrict ourselves to the case $d = 2$ in this subsection.

We shall also assume that $w = 2$: this assumption fits naturally with the defining properties of the JSF, and by a good stoke of luck this brings the highest improvement over the methods studied before for exponents in the range in which we are interested. In fact using Algorithm 1.1 with the JSF and $w = 2$ results in a method which is better than those described earlier even for bit lengths of the exponent for which the optimal window length for said methods is larger than 2 (see Subsection 3.1 for more precise statements).

In this subsection we prove the following theorem.

Theorem 2.12 *Suppose that in Algorithm 1.1 Solinas' JSF is used for the exponents, and $w = d = 2$. Assume further that the JSF of the exponents has length n.*

The expected number of multiplications in the main loop of the algorithm is $\frac{3}{8}n$, and the heuristically expected number of squarings is $n - 2 + \frac{1}{2} = n - \frac{3}{2}$.

The set $\mathcal{E}$ consists of the 10 *elements $g_1^a g_2^b$ with:* (i) *$a = 0$ and $b = 1$;* (ii) *$a = 1$ and $-2 \le b \le 2$;* (iii) *$a = 2$ and $b \in \{\pm 1, \pm 3\}$ and* (iv) *$a = 3$ and $b = \pm 2$. A chain for precomputing all the* 10 *required values other than g_1 and g_2 is*

$$\begin{array}{c} \{\ g_1,\quad g_2,\quad g_1g_2,\quad g_1g_2^{-1},\quad g_1g_2^2,\quad g_1g_2^{-2}, \\ g_1^2g_2,\quad g_1^2g_2^{-1},\quad g_1^2g_2^3,\quad g_1^2g_2^{-3},\quad g_1^3g_2^2,\quad g_1^3g_2^{-2}\ \} \end{array} \tag{5}$$

requiring 10 *multiplications or divisions.*

We assume that the reader is acquainted with the results in Solinas' cited technical report, from which we recall however a few important facts. One of the most important properties of the JSF is that its *joint Hamming weight i.e.* the number of nonzero columns in a joint representation of two integers, is minimal among all (un)signed joint binary representations of the same pair of integers. The average density of the JSF is 1/2 – which gives the heuristical estimate of the squarings in the main loop – whereas that of the joint unsigned binary representation and of the joint NAF is 3/4 and 5/9 respectively. It is natural then to expect that using the JSF in Algorithm 1.1 would lead to an improvement over the complexities of the other two cases even if $w > 1$.

The JSF is uniquely determined by the following properties:

(JSF-1) Of any three consecutive columns, at least one is zero.

(JSF-2) Adjacent nonzero bits have the same sign. In other words, $e_{i,j+1}e_{i,j} = 0$ or 1.

(JSF-3) If $e_{i,j+1}e_{i,j} \neq 0$ then $e_{3-i,j+1} \neq 0$ and $e_{3-i,j} = 0$.

Solinas provides proofs for existence and uniqueness of the JSF, as well as an algorithm for determining it. His algorithm generates the JSF right-to-left. Analysing it Solinas considers three states which he simply calls A, B and C. In state C this algorithm outputs a zero column. In states A or B it outputs nonzero columns. The transition probabilities between these states are explicitly given: we return to this later.

Property **(JSF-1)** suggests that the representation is particularly suitable for an implementation of Algorithm 1.1 with a window width $w = 2$. As already announced we restrict ourselves to this case in the sequel. Further, this choice also simplifies the complexity analysis, by the following observation: Algorithm 1.1 scans a joint representation left-to-right in order to form windows, but we are lucky that a similar algorithm which scans the input right-to-left generates the same windows. This is easy to see, as by property **(JSF-1)** there can be at most two consecutive nonzero columns, which must be preceded and followed by zero columns or by the boundaries of the representation. *Thus consecutive nonzero columns always belong to one window regardless of the direction in which we are scanning the joint representation.*

Hence to estimate the number of nonzero windows (which corresponds to the number of multiplications performed by Algorithm 1.1) we scan our input right-to-left so that we can mimic the procedure we followed in Subsection 2.2. In Solinas' algorithm State A is always followed by State B, State B by State C, and there are the following transition probabilities: $\mathcal{P}(C \mapsto A) = 1/4$, $\mathcal{P}(C \mapsto B) = 1/2$ and $\mathcal{P}(C \mapsto C) = 1/4$. We thus consider a Markov chain with *three* states, which correspond to those in Solinas' algorithm, as follows:

$\mathcal{S}_0^*$. A nonzero column is output by State A of Solinas' algorithm: this column will be the second column in a "square" window when read left-to-right, as the next state in Solinas' algorithm is always State B.

$\mathcal{S}_1^*$. A nonzero column is output by State B of Solinas' algorithm: this column will be the first column in a window when read left-to-right, whereas the second column is non-zero if we are coming from state A or zero if we come from State C.

$\mathcal{S}_2^*$. A zero column is output by State C of Solinas' algorithm.

The number of times we enter in $\mathcal{S}_1^*$ corresponds to the number of windows formed and thus to the number of multiplications performed by our algorithm. The transition probability matrix is

$$T = \left(\mathcal{P}(\mathcal{S}_i^* \mapsto \mathcal{S}_j^*)\right)_{i,j=0}^2 = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1/4 & 1/2 & 1/4 \end{pmatrix}$$

which yields limiting probabilities

$$\pi_0 = \frac{1}{8}, \quad \pi_1 = \frac{3}{8} \quad \text{and} \quad \pi_2 = \frac{1}{2}.$$

Hence the expected number of multiplications performed by Algorithm 1.1 is $\frac{3}{8}n = 0.375n$ with n-bit inputs. This is better than the two variants described earlier, which for $w = d = 2$ attain $\frac{3}{7}n$ and $\frac{11}{27}n$ multiplications respectively.

According to the defining properties of the JSF, the admissible nonzero colums $\binom{e_{1,j}}{e_{2,j}}$ and windows $\begin{pmatrix} e_{1,j} & e_{1,j-1} \\ e_{2,j} & e_{2,j-1} \end{pmatrix}$ with both columns non zero that, up to sign, can be found are

$$\begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ \pm 1 \end{bmatrix}, \begin{bmatrix} 0 & 1 \\ \pm 1 & 0 \end{bmatrix},$$

$$\begin{bmatrix} 1 & 0 \\ 0 & \pm 1 \end{bmatrix}, \begin{bmatrix} 1 & 0 \\ \epsilon & \epsilon \end{bmatrix} \text{ with } \epsilon = \pm 1, \text{ and } \begin{bmatrix} 1 & 1 \\ \pm 1 & 0 \end{bmatrix},$$

thus proving the statements about $\mathcal{E}$. □

3 Comparisons and applications

The algorithm which we explained above is particularly suitable for the computation of double exponentiations in memory constrained environments, such as smart cards or mobile phones, as it requires a relatively small amount of precomputations. In some protocols such as the DSA or the ECDSA one needs to compute products $g_1^{e_1} g_2^{e_2}$ where g_1 is fixed but g_2 varies. In this case the amount of operations is further reduced, as the smaller powers of g_1 can be simply stored within the system.

3.1 Optimal parameters for $d = 2$ and 3

First of all, it is important to know for which values of the parameter w the algorithms run fastest, given the bit length n of the inputs and the number d of the exponents. For simplicity we ignore the number of squarings performed in the main loop and we consider it only for $d = 2$ and 3.

Suppose first $d = 2$. Table 2 summarizes the cardinality of $\mathcal{E}$ and the sum of the number of operations needed to build it with the expected number of multiplications in the main loop of the algorithm. This performance parameter (similar to that used for instance in [Mö]) is a natural way of comparing exponentiation algorithms. In fact, it is easy to adapt these values to the relative costs of squarings by adding $c_s\, n$, where c_s is the cost of a squaring relative to that of a multiplication.

In the column for the JSF there is of course no entry for $w = 3$.

w	Unsigned		NAF		JSF	
	$\#\mathcal{E}$	# Ops	$\#\mathcal{E}$	# Ops	$\#\mathcal{E}$	# Ops
1	3	$\frac{3}{4}n$	4	$1 + \frac{5}{9}n$	4	$1 + \frac{1}{2}n$
2	12	$9 + \frac{3}{7}n$	8	$5 + \frac{11}{27}n$	12	$9 + \frac{3}{8}n$
3	48	$45 + \frac{3}{10}n$	48	$45 + \frac{32}{117}n$		

Table 2: Cardinality of $\mathcal{E}$ and number of operations for $d = 2$

Remark 3.1 *Using the unsigned binary representation, the optimal choice of w is $w = 1$ for $n \leq 28$, and $w = 2$ for $28 \leq n \leq 280$. In particular, for the range of exponents which interests us most the parameter $w = 2$ is optimal.*

With the NAF the thresholds are $n = 27$ and $n = \frac{14040}{47} = 298.72$ respectively.

The JSF brings us a surprise. The parameter $w = 1$ is optimal for $n \leq 64$, a remarkably high value. Furthermore, using the JSF with $w = 2$ is better than using the NAF with either $w = 2$ or 3 for $n \leq 354$. In this range, using the JSF yields the better algorithm already for $n \geq 4$.

Table 3 collects the analogous data for $d = 3$: Note that the JSF, being defined only for $d = 2$, is not represented.

w	Unsigned		NAF	
	$\#\mathcal{E}$	# Ops	$\#\mathcal{E}$	# Ops
1	7	$3 + \frac{7}{8}n$	13	$9 + \frac{19}{27}n$
2	56	$52 + \frac{7}{15}n$	49	$45 + \frac{131}{297}n$
3	448	$444 + \frac{7}{22}n$	603	$599 + \frac{1082}{3645}n$

Table 3: Cardinality of $\mathcal{E}$ and number of operations for $d = 3$

Remark 3.2 *In the case $d = 3$ the thresholds are higher, as intuition suggests. Using the unsigned binary representation, the optimal choice of w is $w = 1$ for $n \leq 120$, and $w = 2$ for $121 \leq n \leq 2640$. In the NAF case, $w = 1$ is optimal for $n \leq 137$ and $w = 2$ for $138 \leq n \leq 3841$.*

If $w = 1$, the NAF leads to better performance as long as $n > 35$, if $w = 2$ the NAF will always yield a better algorithm. However, if $w = 3$, the much larger constant term in the complexity when using the NAF has a price: for $n \leq 7264$ it is better to use the unsigned binary representation.

3.2 Comparison with interleaved exponentiation

Recently a multi-exponentiation algorithm called *interleaved exponentiation* has been described by Möller [Mö]. The algorithm is better understood in terms of exponent recording, so that it becomes clear that it is nothing but the naive left-to-right multi-exponentiation algorithm applied to a different representation of the exponents. Suppose that the exponents $e_1, \ldots, e_d$ are written as

$$e_i = \sum_{j=0}^{n-1} e_{i,j}\, 2^j \tag{6}$$

where the coefficients $e_{i,j}$ are allowed to vary in a set larger than $\{0, \pm 1\}$. Then the following algorithm computes $x := \prod_{i=1}^{d} g_i^{e_i}$.

Algorithm 3.3 Left-to-right interleaved multi-exponentiation

Input: Group elements $g_1, \dots, g_d$ of which some powers have been precomputed and exponents $e_i = \sum_{j=0}^{n-1} e_{i,j}\, 2^j$

Output: $\prod_{i=1}^{d} g_i^{e_i}$

```
Step 1.  $x \leftarrow 1 \in G$
Step 2.  for $j = n-1 \dots 0$ do {
         (a)  $x \leftarrow x^2$                          [Skip at first iteration]
              for $i = 1 \dots d$ do {
         (b)      if $e_{i,j} \neq 0$ then $x \leftarrow x \cdot g_i^{e_{i,j}}$ } }
Step 3.  return $x$
```

This algorithm becomes efficient if a careful choice of the recoding of the exponents is done, which must balance the need for a low density of the representations against the work done in the precomputation stage: this should allow Step 2 (b) to be done always with a table access and a single multiplication (or multiplication with the inverse).

A left-to-right sliding window algorithm with a window width w corresponds to a recoding $e = \sum_{j=0}^{n-1} e_j\, 2^j$ where the coefficients e_j satisfy $0 \leq e_j < 2^w$, are either zero or odd, and of any consecutive w of them only one is nonzero. It is very well known that this representation has density $1/(w+1)$. We call this recoding the *width w sliding window recoding*, or wSWR for short. The same density is achieved by a similar method using a window sliding right-to-left. We note that such recodings operate on the binary representations of the exponents, so recoding from left to right can be done *online*, *i.e. during* the exponentiation proper, without the need to store the recoded representation.

Cohen's *flexible window* exponentiation algorithm [CoMiOn1, Co] which was also proposed independently by Solinas [So1, So2] consists in the application of Algorithm 3.3 with $d = 1$ to the *wNAF* of the exponent. The wNAF of the integer e is a representation $e = \sum_{j=0}^{n-1} e_j\, 2^j$ where the integer coefficients e_j satisfy the following two conditions:

(wNAF-1) Either $e_j = 0$ or e_j is odd and $|e_j| \leq 2^w$.

(wNAF-2) Of any $w+1$ consecutive coefficients $e_{j+w}, \dots, e_j$ at most one is nonzero.

It is also called *width-$(w+1)$ NAF* and it must not be confused with the *generalized NAF* or *GNAF* [ClLi], which is a signed recoding of a radix-r representation of integers for arbitrary r.

Every integer admits a wNAF which is uniquely determined. In the cited papers by Solinas and by Cohen et al. there are algorithms for computing it and the density of the representation is $1/(w+2)$ The special case $w = 1$ is the usual NAF. This immediately leads to an exponentiation algorithm requiring about $n/(w+2)$ multiplications for an n-bit exponent. The recoding algorithms work right-to-left and cannot be used online.

Assume that the exponents $e_1, \dots, e_d$ have approximately the same bit-lenght n:

	$d=2$				$d=3$			
n	Algorithm 1.1		Algorithm 3.3		Algorithm 1.1		Algorithm 3.3	
	$\#\mathcal{E}$ and	# Ops (w)	$\#\mathcal{E}$ and	# Ops (w)	$\#\mathcal{E}$ and	# Ops (w)	$\#\mathcal{E}$ and	# Ops (w)
56	12	33 (w=2)	8	35 (w=3)	7	52 (w=1)	12	53 (w=3)
64	12	36.43 (w=2)	8	39 (w=3)	7	59 (w=1)	12	59 (w=3)
80	12	42.43 (w=2)	8	47 (w=3)	7	73 (w=1)	12	71 (w=3)
86	12	45.86 (w=2)	8	50 (w=3)	7	78.25 (w=1)	12	75.50 (w=3)
96	12	50.14 (w=2)	8	55 (w=3)	7	87 (w=1)	12	83 (w=3)
128	12	63.86 (w=2)	8	71 (w=3)	56	111.86 (w=2)	12	107 (w=3)
160	12	77.57 (w=2)	8	87 (w=3)	56	126.67 (w=2)	12	131 (w=3)
192	12	91.28 (w=2)	8	103 (w=3)	56	141.60 (w=2)	24	138.20 (w=4)
240	12	111.86 (w=2)	16	111 (w=4)	56	164 (w=2)	24	167 (w=4)
256	12	118.71 (w=2)	16	117.40 (w=4)	56	171.47 (w=2)	24	176.60 (w=4)

Table 4: Complexity of multi-exponentiation using unsigned representations

	$d=2$				$d=3$			
n	Algorithm 1.1 (JSF)		Algorithm 3.3 (wNAF)		Algorithm 1.1 (NAF)		Algorithm 3.3 (wNAF)	
	$\#\mathcal{E}$ and	# Ops (w)	$\#\mathcal{E}$ and	# Ops (w)	$\#\mathcal{E}$ and	# Ops (w)	$\#\mathcal{E}$ and	# Ops (w)
56	4	29 (w=1)	8	29.40 (w=3)	13	48.40 (w=1)	12	44.60 (w=3)
64	4	33 (w=1)	8	32.60 (w=3)	13	54.03 (w=1)	12	49.40 (w=3)
80	12	39 (w=2)	8	39 (w=3)	13	65.29 (w=1)	12	59 (w=3)
86	12	41.25 (w=2)	8	41.40 (w=3)	13	69.52 (w=1)	12	62.60 (w=3)
96	12	45 (w=2)	8	45.40 (w=3)	13	76.55 (w=1)	12	68.60 (w=3)
128	12	57 (w=2)	16	57.66 (w=4)	13	99.07 (w=1)	24	87 (w=4)
160	12	69 (w=2)	16	68.33 (w=4)	49	115.57 (w=2)	24	103 (w=4)
192	12	81 (w=2)	16	79 (w=4)	49	129.69 (w=2)	24	119 (w=4)
240	12	99 (w=2)	16	95 (w=4)	49	150.86 (w=2)	24	143 (w=4)
256	12	105 (w=2)	16	100.33 (w=4)	49	157.91 (w=2)	24	151 (w=4)

Table 5: Complexity of multi-exponentiation using signed representations

If inversion in the group is not for free, we recode online the exponents as wSWR's. Algorithm 3.3 requires then d squarings and $d(2^{w-1}-1)$ multiplications in the precomputation stage and $\frac{dn}{w+1}-1$ multiplications and about n squarings in the main loop (note that the first multiplication is just a variable assignment). In Table 4 we add the number of operations in the precomputation stage to the number of multiplications in the main loop of the algorithms.

If inversion in the group is cheap, we write the exponents as wNAF's. Algorithm 3.3 needs d squarings and $d(2^{w-1}-1)$ multiplications for the precomputations and $\frac{dn}{w+2}-1$ multiplications and about n squarings in the main loop. Table 5 collects the complexity data for these algorithms which exploit signed representations.

Remarks 3.4 (1) *We observe that rewriting for the wNAF with $w>1$ may impact memory requirements and/or performance more than recoding for the NAF and possibly also for the JSF. This results from the internal representation of the wNAF. Either a* *wNAF is stored as a sequence of (nonzero) coefficient-exponent pairs (at most $\lceil n/w \rceil$ pairs) making the details of Algorithm 3.3 much more complicated, or as a length n vector of w bit-numbers.*

On the other hand a signed binary representation requires just two bits per coefficient and there is a very simple way of writing down the NAF in software. Let $e = \sum_{j=0}^{n-1} b_j 2^j$ be an integer in base 2. Let $r := 3e = \sum_{j=0}^{n+1} r_j 2^j$ the base 2 representation of $3e$. Then $(r-e)/2 = \sum_{j=0}^{n} (r_{j+1} - b_{j+1}) 2^j$ (assume $b_n = b_{n+1} = 0$) is a signed binary representation of e and the coefficients $e_j := r_{j+1} - b_{j+1} \in \{0, \pm 1\}$ also satisfy $e_j e_{j+1} = 0$. It can be easily proved [ClLi] that this method is essentially the same as Reitwiesner's algorithm.

There exists an alternative to the NAF with the same Hamming weight and which can be computed from left to right [JoYe1] by a simple algorithm. However this representation dispenses with the non-adjacency property: Hence, for $w = d = 2$, the set of precomputations $\mathcal{E}$ consists of the values $g_1^a g_2^b$ with either $0 < a \leq 3$ and $-3 \leq b \leq 3$, at least one of a, b odd or $a = 0$ and $b = 1$ or 3, a total of 20 values. For this reason and by virtue of the fact that the density of the corresponding joint representation cannot be better than that of the JSF, we do not consider its usage here.

(2) *Another possibility is given by using a radix-r representation in Algorithm 3.3, where r is a small power of 2, say $r = 2^w$. This is very simple, exponents are scanned left-to-right r bits at a time, thus online, and all blocks of multiplications happen only every w squarings. On the other hand its performance is clearly poorer than using the wSWR. A better alternative would be to use the GNAF, which is a signed radix-r recoding [ClLi]: With it the density of the nonzero digits decreases from $\frac{r-1}{r}$ of the radix-r representation to $\frac{r-1}{r+1}$, hence it leads to an multi-exponentiation algorithm requiring $dn\frac{2^w-1}{w(2^w+1)}$ multiplications and about n squarings to compute the product of d powers with n-bit exponents. This is too worse than the wSWR for $w > 2$ and the wNAF for $w > 1$. However there is an alternative recoding achieving the same densities and which operates left-to-right [JoYe2].*

(3) *Only implementation can decide which of the algorithms is best for each purpose if the number of operations is similar. It seems to us, however, that Algorithm 1.1 with the JSF should be preferable to Algorithm 3.3 for double exponentiations with exponents from 160 to 256 bits in memory constrained environments. For triple exponentiations Algorithm 3.3 seems always preferable with unsigned representations (using the wSWR) and $n \geq 128$ or with signed representations (using the wNAF).*

3.3 Applications

In this subsection we show just a few possible applications of the above multi-exponentiation algorithms.

3.3.1 Elliptic and hyperelliptic curves

Here, as well as in the next paragraph, we shall use additive terminology (and shall speak, for example, of a scalar product $r \cdot P$ instead of an exponentiation P^r).

The first obvious application of the Algorithms described in this paper is to electronic signature schemes based on the discrete logarithm problem in the rational point group of an elliptic curve (ecc) or of the Jacobian variety of an hyperelliptic curve (hec) over a finite field.

In the ecc case we observe that mixed coordinate systems can be used [CoMiOn2], because our algorithms are based on alternating sequences of several squarings and single multiplications exactly as the methods proposed by Cohen et al. The difference is that we compute here directly the double scalar product, whereas Cohen et al. compute the two scalar products separately. For the fixed base scalar multiplicatio they use essentially a comb method and for the variable base scalar product the flexible window algorithm. The two results are then added together.

It is a simple exercise to verify that the number of finite field operation required by Algorithm 1.1 with the JSF and their method is extremely close for $n = 160, 192$ and 224 (a difference of at most 1%!). However the method by Cohen et al. requires a lot of precomputed powers of the fixed base to be stored in ROM with the system (62 values in the case $n = 160$) whereas the requirements by our method are minimal.

3.3.2 Trace zero varieties

Trace zero varieties are abelian varieties constructed essentially by Weil Descent from other varieties, such as elliptic curves [Na, Fr] or Jacobians of hyperelliptic curves [La1, La2].

Construction and security parameters. We start with an elliptic curve (resp. hyperelliptic curve of genus g) defined over a prime field $\mathbb{F}_p$ where p^2 (resp. p^{2g}) has the order of magnitude of the desired group size. We also assume that the characteristic polynomial of the Frobenius endomorphism is known. Next, we consider the group of rational points of the elliptic curve (resp. ideal class group) over the finite field extension $\mathbb{F}_{p^3}$ and consider the elements defined by the property that its elements D are of trace zero, *i.e.* they satisfy $(\sigma^2 + \sigma + 1)(D) = 0$. In general for a genus g curve considered over $\mathbb{F}_{p^d}$ the elements of trace zero form a subgroup as they are the kernel of a homomorphism. Therefore they form an abelian subvariety of dimension $g(d-1)$, which is called the *trace-zero variety.* We shall denote it by G in the sequel and call G_0 the subgroup of large prime order ℓ in which we actually implement the cryptographic primitives.

For cryptographic applications we must take into account the existence of efficient index calculus attacks for solving the discrete logarithm problem in abelian varieties of dimension at least 4: Gaudry actually describes his low genus variant [Ga] only for the Jacobian of hyperelliptic curves, but it should apply almost invaried to arbitrary varieties with given group law and dimension at least 4. To keep us on the secure side we assume that it is so. For dimension 4 Gaudry's algorithm has complexity $O(p^2)$ but Robert Harley observed [Ga] that the complexity is $O(p^{\frac{8}{5}})$ under the assumption that the factorization of polynomials can be done in polynomial time, which is true in practice. On the other hand the best attack known up to date to the discrete logarithm problem for genus up to 3 is given by Pollard's rho method with complexity $O(\sqrt{\ell}) = O(p^2)$. Its implied constants are better than in index calculus, but we should not ignore speed-ups arising from efficiently computable endomorphism: The Frobenius can be computed in our situation with relative ease, but not for free, and has order 3, so a speed-up of the index calculus by a factor 9 is expected, whereas Pollard's rho probably does not gain the factor $\sqrt{3}$.

We require security comparable to that of an elliptic curve over a finite field of about 2^{160} elements. The rough equation $p^{8/5} \approx \sqrt{2^{160}}$ implies $p \approx 2^{50}$ and thus $\ell \approx 2^{200}$ for the trace zero variety: This is probably paranoid, whereas $\ell \approx 2^{160}$ appears to be too small. We shall then presume that a trace zero variety with $\ell \approx 2^{180}$ points over $\mathbb{F}_p$ and $p \approx 2^{45}$ satisfies our security requirements.

For the same reasons varieties of dimension larger than 4 should be avoided. So $g(d-1) \leq 4$ limits considerably the choices for g and d. As observed by Lange [La2] an extension of degree $d = 3$ is "large enough to keep a large part of the group order". In what follows we consider only the case 2 for simplicity.

Performance advantages in cryptographic applications. The main performance advantages of trace-zero varieties come from the fast arithmetic in the extension field (where explicit closed formulae can be given for multiplication and squaring: if furthermore the polynomial defining the extension field is chosen carefully one can even use short convolutions [Bl, AvMi]) and by the presence of the automorphism σ of small degree.

The latter fact enables one to speed-up even *single* exponentiations by means of multi-exponentiation algorithms. Instead of using single scalars to compute $r \cdot D$ for a point or ideal class D, Lange considers a pair (r_0, r_1) of scalars bounded by some quantity which is $O(p^2)$, and computes the double scalar product $r_0 \cdot D + r_1 \cdot \sigma(D)$. For r_0 and r_1 suitably bounded (see [Na, La2]) all such double scalar products are distinct.

One observes at once that a variant of Shamir's trick can be used and the result is that the number of doublings (squarings in the multiplicative terminology) needed in cryptographic operations is roughly halved. Further savings can be achieved by the use of Algorithms 1.1 and 3.3, depending on the parameters.

All the usual cryptographic protocols can be adapted to this new setting, in particular those for key exchange and electronic signatures.

The Frobenius operates on G, and thus on G_0, like the scalar multiple by a constant s with $s^2 + s + 1 \equiv 0 \mod \ell$. For the verification of signatures, in place of the scalar product $r \cdot D + u \cdot E$ one is temped to write $r \equiv r_0 + r_1 s$ and $u \equiv u_0 + u_1 s \bmod \ell$ and to consider the *quadruple* product $r_0 \cdot D + r_1 \cdot \sigma(D) + u_0 \cdot E + u_1 \cdot \sigma(E)$. The problem is that the coefficients are not automatically bounded by kp^2 where k is a small constant. In the example above we did not have this problem because we started with a pair (r_0, r_1), however for the verification of digital signatures one needs to start with the given value r. To keep the coefficient reasonably bounded can be cumbersome, but without entering into details we just assume for now that our example is good and that the bound is $O(p^2)$. In this case we suggest the use of Algorithm 3.3 and the wNAF. The number of operations in the precomputation stage is 0 if $w = 1$, otherwise it is $4 \cdot 2^{w-1}$, plus an application of σ (to the point E – we assume D is the fixed point of the system). In the main loop there are about $n = \log_2(p^2)$ squarings and $\frac{4\,n}{w+2} - 1$ multiplications. To determine the optimal value of w we have thus to minimize the quantity $\phi(n, w) := 2^{w-1} + \frac{n}{w+2}$ if $w \geq 2$ and $\phi(n, 1) := n/3$. Now $\phi(n, w)$ is the number of multiplications which are associated to each n-bit wNAF-recoded exponent in a left-to-right (multi-)exponentiation, whereas the number of squarings and the fact that one can save one multiplication right at the beginning does not depend on the number of exponents.

The total number of group operations is then about $n + 4\,\phi(n,w) - 1$. If $w = 2$ we need to store 6 values, 14 if $w = 3$. If we choose $w = 1$ the complexity may be worse but we need *no precomputed values apart from D, $\sigma(D)$, E and $\sigma(E)$.*

For worst case curves two of the coefficients, namely r_1 and u_1, are bound by $p^{7/2}$ and the other two by p^2 according to [La2]. In Algorithm 3.3 the exponents need not be recoded in the same way. We choose to recode r_0 and u_0 as wNAF's and r_1 and u_1 as w'NAF's for two possibly different window widths w and w'. The average number of group operations in this case is $\frac{7}{4}n + 2\,\phi(n,w) + 2\,\phi\big(\frac{7}{4}n, w'\big) - 1$.

Let us consider signature verification using a trace zero variety arising from a genus 2 *curve over a finite field of about* 2^{45} *elements (here* $n = 90$*): It may be done, on well behaved curves, with about* 177 *group operations, obtained for* $w = 3$*, and some divisions with remainder of* 180 *bit integers. The minimum expected number of group operations for worst case curves grows to* 269*, obtained with* $w = 3$ *and* $w' = 4$*. For comparison, using the ECDSA or hyperelliptic curve variants thereof of comparable security requires a minimum of* 229 *group operations (see Table 4 with* $n = 160$ *and* $d = 2$*).*

3.3.3 XTR

The XTR cryptosystem was initially proposed by Lenstra and Verheul [VeLe] and makes use of the subgroup G of order $p^2 - p + 1$ of the multiplicative group of the cyclotomic extension $\mathbb{F}_{p^6}/\mathbb{F}_p$. Let g be an element of $\mathbb{F}_{p^6}^{\times}$ of order $q > 6$ dividing $p^2 - p + 1$. Since q does not divide $p^s - 1$ for $s = 1, 2, 3$ the subgroup generated by g cannot be embedded in the multiplicative group of any proper subfield of $\mathbb{F}_{p^6}$. Hence it appears that solving the discrete logarithm problem in $\langle g \rangle$ is at least as difficult as solving it in the large field. In the XTR cryptosystem elements from the field $\mathbb{F}_{p^6}$ are replaced by their traces over $\mathbb{F}_{p^2}$ and Lenstra et al. show how one can work only with these – actually with triples of traces – instead of using the original elements from the bigger field. This leads to very efficient arithmetic even though it is definitely not straightforward to port the usual exponentiation algorithms to this new setting. All this makes XTR interesting (both in the usual meaning of the word and according to the Chinese curse...).

Recently, Lenstra and Stam [StLe] observed that one can also compute directly in an efficient manner in the field $\mathbb{F}_{p^6}$ by using a suitable representation of the intermediate extensions. This allows the implementor to use all possible (multi-)exponentiation methods without change.

Independently, Frey suggested a similar idea which we sketch here (the following text is taken, abridged, from [AvLa]). Let σ be the Frobenius map $x \mapsto x^p$. One observes at once that for $z \in G$ the Frobenius satisfies $z^{\sigma^2 - \sigma + 1} = 1$ and that G is the intersection of the two trace zero varieties relative to *both* intermediate extensions, so that the elements satisfy $\sigma^3 + 1 = 0$ and also $\sigma^4 + \sigma^2 + 1 = 0$. The first relation immediately gives a simple inversion formula: $z^{-1} = \sigma^3(z)$. The field $\mathbb{F}_{p^6}$ is then constructed as the composite of two extensions of $\mathbb{F}_p$: the first, $\mathbb{F}_{p^3}$ is given by an irreducible binomial $X^3 + a$ over $\mathbb{F}_p$; the second is $\mathbb{F}_{p^2} = \mathbb{F}_p(\sqrt{\delta})$ where $\delta \in \mathbb{F}_p \setminus (\mathbb{F}_p)^2$. Ideally $|\delta|$ should be small (for instance $\delta = -1$: to allow this one needs $-1 \in \mathbb{F}_p \setminus (\mathbb{F}_p)^2$ and therefore $p \equiv 3 \bmod 4$). Also $\delta = 2$

is a good option. The constant a should be small, too, to make polynomial reduction inexpensive.

For $z \in G$ write $z = x + y\sqrt{\delta}$ where $x, y \in \mathbb{F}_{p^3}$. The map σ^3 generates the group $\mathrm{Gal}(\mathbb{F}_{p^6}/\mathbb{F}_{p^3})$ of order 2, hence $\sigma^3(\sqrt{\delta}) = -\sqrt{\delta}$ and $z^{-1} = x - y\sqrt{\delta}$ is essentially for free.

One can then apply the considerations made in the previous paragraph about trace-zero varieties also to XTR subgroups. In particular, single and double exponentiations found in cryptographic protocols can be transformed into double and quadruple exponentiations with exponents of halved bit length.

3.4 A security bit

Solinas' original method may also lead to a reduction in security of the cryptosystem under side channel attacks, as it can reveal information about the sequence of squarings and multiplications exactly as the left-to-right binary exponentiation method. We restrict our attention here to Simple Power Attacks (SPA). In fact, the positions of the zero columns are given by the sequences of at least two squarings, and they are on average $n/2$. There are several possibilities for the nonzero columns even though the types of two consecutive nonzero columns are limited, so it might not be trivial to exploit this information.

One can also try to apply a side channel attack to a cryptosystem where exponentiation is done by means of Algorithm 1.1: Here a sequence of two squarings does not necessarily mean that a zero column is found, as this is done also before multiplying by the precomputed value corresponding to two nonzero adjacent columns.

We also suggest a way to make life more difficult to an attacker who wants to break a cryptosystem based on Solinas' idea by means of side channel attacks. The idea consists in computing also $g_1^{\pm 2}$ and $g_2^{\pm 2}$. In groups with fast or free inversion this is done just by two squarings. Then one replaces randomly chosen squares in the representation as follows

$$\pm\begin{bmatrix}1 & 0\\ 0 & 0\end{bmatrix} \mapsto \pm\begin{bmatrix}0 & 2\\ 0 & 0\end{bmatrix}, \qquad \pm\begin{bmatrix}0 & 0\\ 1 & 0\end{bmatrix} \mapsto \pm\begin{bmatrix}0 & 0\\ 0 & 2\end{bmatrix}$$

which amounts to including ± 2 in the set of digits. In this way, the sequence of squarings cannot give information about the bits in the original representations. One might think that the trick just described increases the number of precomputations which need to be done. In principle this is true, but in practice g_1 is the fixed base point of the cryptosystem so that $g_1^{\pm 2}$ can also be stored within the system and the implementor can decide to use only the first substitution systematically, without employing the second substitution and without having to make random choices: These choices are in a sense "automatically done" by the representations of the integers.

However, our method proposed in Section 1 attains a better complexity anyway so we believe that it is preferable.

References

[ANSI] ANSI X9.62, *Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)*. 1999.

[ArWh] S. Arno and F.S. Wheeler, *Signed digit representations of minimal Hamming weight.* IEEE Transactions on Computers **42** (1993), 1007–1010.

[AvLa] R. Avanzi and T. Lange, *Überlegungen zu XTR.* Unpublished manuscript.

[AvMi] R. Avanzi and P. Mihăilescu, *Generic efficient arithmetic algorithms for Processor Adequate Finite Fields.* A manuscript.

[Bl] R.E. Blahut, *Fast algorithms for digital signal processing.* Addison-Wesley publishing company, 1985.

[Bo] A.D. Booth, *A signed binary multiplication technique.* Quart. Journ. Mech. and Applied Math. **4** (1951), 236–240.

[ClLi] W.E. Clark and J.J. Liang, *On arithmetic weight for a general radix representation of integers.* IEEE Transactions on Information Theory **IT-19** (1973), 823–826.

[ChGeGo$^+$] B.W. Char, K.O. Geddes, G.H. Gonnet, B.L. Leong, M.B. Monagan and S.M. Watt, *Maple V Language Reference Manual.* Springer, 1991.

[Co] H. Cohen, *Analysis of the flexible window powering algorithm.* Preprint. Available from: `http://www.math.u-bordeaux.fr/~cohen/`

[CoMiOn1] H. Cohen, A. Miyaji and T. Ono, *Efficient elliptic curve exponentiation.* In *Proceedings ICICS'97*, Lecture Notes in Computer Science, vol. 1334, Springer-Verlag, 1997, pp. 282–290.

[CoMiOn2] H. Cohen, A. Miyaji and T. Ono, *Efficient elliptic curve exponentiation using mixed coordinates.* In *Advances in Cryptology – ASIACRYPT '98 (1998)*, K. Ohta and D. Pei, Eds., vol. 1514 of Lecture Notes in Computer Science, pp. 51–65.

[El] T. ElGamal, *A public-key cryptosystem and a signature scheme based on discrete logarithms.* IEEE Transactions on Information Theory IT-31 (1985), 469-472.

[EğKo] Ö. Eğecioğlu and Ç. K. Koç. *Exponentiation using canonical recoding.* Theoretical Computer Science, 129(2):407–417, 1994.

[Fr] G. Frey, *Applications of arithmetical geometry to cryptographic constructions.* In *Finite fields and applications (Augsburg, 1999)*, pages 128–161. Springer, Berlin, 2001.

[GaLaVa] R.P. Gallant, R.J. Lambert, S.A. Vanstone, *Faster Point Multiplication on Elliptic Curves with Efficient Endomorphisms* In *Advances in Cryptology – CRYPTO 2001 Proceedings*, pp. 190–200, Springer Verlag, 2001.

[Ga] P. Gaudry, *An algorithm for solving the discrete log problem on hyperelliptic curves.* In *Advances in Cryptology, Eurocrypt '2000*, vol. 1807 of Lecture Notes in Computer Science, pp. 19–34. Springer-Verlag, 2000.

[JoYe1] M. Joye and S.-M. Yen, *Optimal left-to-right binary signed-digit recoding.* IEEE Transactions on Computers **(49) 7**, 740–748 (2000).

[JoYe2] M. Joye and S.-M. Yen, *New Minimal Modified Radix-r Representation* In *Public Key Cryptography – 5th International Workshop on Practice and Theory in Public Key Cryptosystems, PKC 2002, Paris, France, February 2002*, D. Naccache and P. Paillier Eds., Lecture Notes in Computer Science, vol 2274, pp. 375–384.

[Kn] D. E. Knuth, *The art of computer programming. Vol. 2, Seminumerical algorithms*, third ed., Addison-Wesley Publishing Co., Reading, Mass., 1997, Addison-Wesley Series in Computer Science and Information Processing.

[La1] T. Lange, *Efficient Arithmetic on Hyperelliptic Curves.* PhD thesis, Universität Essen, 2001.

[La2] T. Lange, *Trace-Zero Subvariety for Cryptosystems.* Preprint.

[Mö] B. Möller, *Algorithms for Multi-exponentiation* in S. Vaudenay, A.M. Youssef (Eds.): *Selected Areas in Cryptography - SAC 2001.* Springer-Verlag LNCS 2259, pp. 165-180.

[MoOl] F. Morain and J. Olivos, *Speeding up the computations on an elliptic curve using addition-subtraction chains.* RAIRO Inform. Theory **24** (1990), 531–543. Available also as Chapter 4 of F. Morain's Ph.D. Thesis. See F. Morain's home page at: `http://ultralix.polytechnique.fr/~morain/index.html`

[Na] N. Naumann, *Weil-Restriktion abelscher Varietäten.* Master's thesis, Universität Essen, 1999.

[Re] G. W. Reitwiesner. *Binary arithmetic.* Advances in Computers **1**, 231–308, 1960.

[SiCiQu] F. Sica, M. Ciet and J.-J. Quisquater, *Analysis of the Gallant-Lambert-Vanstone Method based on Efficient Endomorphisms: Elliptic and Hyperelliptic Curves.* In *Proceedings of Selected Areas of Cryptography 2002 (SAC 2002), St. John's, Newfoundland (Canada), August 2002.* To appear.

[So1] J.A. Solinas, *An improved algorithm for arithmetic on a family of elliptic curves.* In *Advances in Cryptology – CRYPTO '97 (1997)*, B. S. Kaliski, Jr., Ed., Lecture Notes in Computer Science vol. 1294, pp. 357–371.

[So2] J.A. Solinas, *Efficient arithmetic on Koblitz curves.* Designs, Codes and Cryptography **19** (2000), 195–249.

[So3] J.A. Solinas, *Low-Weight Binary Representations for Pairs of Integers.* Centre for Applied Cryptographic Research, University of Waterloo, Combinatorics and Optimization Research Report **CORR 2001-41**, 2001. Available from:
`http://www.cacr.math.uwaterloo.ca/techreports/2001/corr2001-41.ps`

[StLe] M. Stam and A.K. Lenstra, *Efficient subgroup exponentiation in quadratic and sixth degree extensions.* In *Proceedings of Workshop on Cryptographic Hardware and Embedded Systems CHES 2002, August 13 - 15, 2002*, to be published by Springer–Verlag.

[VeLe] E.R. Verheul and A.K. Lenstra. *The XTR public key system.* In *Advances in Cryptography – Crypto'00*, M. Bellare, ed., vol. 1880 of Lecture Notes in Computer Science, pages 1–19. Springer-Verlag, 2000.

[YeLaLe] S.-M. Yen, C.-S. Laih and A.K. Lenstra, *Multi-exponentiation.* IEE Proc. Comput. Digit. Tech., Vol. 141, No. 6, november 1994.