ON MULTI-EXPONENTIATION IN CRYPTOGRAPHY

ROBERTO M. AVANZI* (IEM ESSEN)
Email: mocenigo@exp-math.uni-essen.de

October 8, 2002. Revised: October 28, 2002.

Abstract

We describe and analyze new combinations of multi-exponentiation algorithms with representations of the exponents. We deal mainly but not exclusively with the case where the inversion of group elements is fast: This is true for example for elliptic curves, groups of rational divisor classes of hyperelliptic curves, trace zero varieties and XTR.

These methods are most attractive with exponents in the range from 80 to 256 bits, and can also be used for computing *single* exponentiations in groups which admit an automorphism satisfying a monic equation of small degree over the integers.

The choice of suitable exponent representations allows us to match or improve the running time of the best multi-exponentiation techniques in the aforementioned range, while keeping the memory requirements as small as possible. Hence some of the methods presented here are particularly attractive for deployment in memory constrained environments such as smart cards. By construction, such methods provide good resistance against side channel attacks.

We also describe some applications of these algorithms.

Keywords and phrases: Cryptographic protocols, Exponentiation, Integer recoding, Scalar multiplication, Elliptic and Hyperelliptic curves, Trace zero varieties, XTR, Groups with automorphisms, Smart card applications

Contents

*The work described in this paper has been supported by the Commission of the European Communities through the IST Programme under Contract IST-2001-32613 (see http://www.arehcc.com).

1 Introduction

Some public-key cryptographic protocols such as the verification of digital signatures require the computation of the product of powers of two [1], three [12] or more elements of a group. Furthermore, in some algebraic structures the computation of a single exponentiation can be reduced to such a product: If a commutative 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 [15, 27]). In this context, too, the cases $d = 2$ and $d = 3$ are of particular practical relevance because of trace zero varieties and XTR (see Section 5).

Such computations can be performed by computing the various powers separately and then multiplying them together. If one of the bases is known in advance and one can afford to store a lot of precomputed values, one can obtain very good performance from this idea [10]. However, if the intermediate results are not needed elsewhere, one can do much better in the general case.

Shamir's trick can be extended in a straightforward way by using sliding windows: To our knowledge this was first reported in [33]. So far, this generalisation of Shamir's trick has been applied only to the usual binary representation of the exponents, and only in this basic form Möller has compared it to other methods such as *interleaved exponentiation* [23]. We combine the idea of [33] with different exponent recodings and compare these variants with interleaved multiexponentiation, thus extending Möller's analysis. One of our main concerns is to keep the memory requirements as small as possible, which is important for smart card implementations. Our results can be summarized as follows:

(1) We propose and analyze variants of the algorithm from [33] which are better suited to groups where inversion is cheap. This is done by considering signed digit representations – first introduced in [6] – of the exponents. In particular we consider the *non-adjacent form* [26, 24] (see Theorem 3.6 and Remark 3.7) and a new representation of pairs of integers due to Solinas [30] (see Theorem 3.12).

Our best algorithms perform double exponentiations with similar performance as the best previous methods while having much smaller memory requirements: See Tables 5 and 6 (the number of operations there does not include the number of squarings, which is essentially the same in all methods which we consider) and Remark 4.6 (2). For example for a bit length n *with* $124 < n \leq 354$ *a double exponentiation is done by* $9 + 3n/8$ *multiplications on average and about* n *squarings, using only* 12 *precomputed values: the precomputations can be reduced further paying only a minimal performance penalty.*

(2) To our knowledge Theorem 3.6 complements existing literature on single exponentiations.

(3) We show in Section 5 how all these methods can be used to speed up current and proposed cryptosystems.

Power analysis of cryptosystems has aroused a lot of attention in the last years. It attempts to guess secret keys by monitoring power signals of cryptographic devices. Since distinct operations have different power consumptions, the use of simple square-and-multiply exponentiation methods can allow a potential attacker to recover the binary representation of the exponent. If, for example, digits ± 1 are used like in the NAF, the attacker can gain less information. Window methods are even better and the adoption of multi-exponentiation algorithms such as those presented in this paper hide the secret information very efficiently.

In the next section we will introduce the general form of the algorithm from [33], which will be analysed in detail in Section 3: this forms the main part of this paper. In Section 4 the optimal parameters will be discussed and the resulting time and space complexities will be compared against those of interleaved exponentiation. Next, some applications will be outlined.

Acknowledgements. 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 [33].

Some computations have been performed using the `maple` computer algebra system [8].

2 The algorithm

We now proceed with the derivation and the description of the algorithm from [33]. 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 (*bit* means *binary digit*): *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.

The most obvious way of performing the desired computation, as already mentioned, consists in computing the powers separately and multiplying them together. The second most obvious way is perhaps the following one, which saves some squarings.

Algorithm 2.1 Simple multi-exponentiation

INPUT: Group elements $g_1, \ldots, g_d$ and corresponding exponents $e_1, \ldots, e_d$ written as in (1) in base 2 (*i.e.* with $e_{i,j} \in \{0, 1\}$)
OUTPUT: $\prod_{i=1}^{d} g_i^{e_i}$

Step 1. $x \leftarrow 1 \in G$

Step 2. **for** $j = n-1 \ldots 0$ **do** $\{$

$x \leftarrow x^2$ [Skip at first iteration]

for $i = 1 \ldots d$ **do** $\{$ **if** $e_{i,j} = 1$ **then** $x \leftarrow x \cdot g_i$ $\}$ $\}$

Step 3. `return` x

Shamir's trick improves Algorithm 2.1 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. 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 $\prod_{i=1}^{d} g_i^{k_i}$ with $k_i \in \{0, \pm 1\}$. 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 [19] in the same way as the square-and-multiply method. We describe the resulting algorithm.

Algorithm 2.2 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 2.3 (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) (i) $x \leftarrow x^{2^{w-s}}$; (ii) $x \leftarrow x \cdot \prod_{i=1}^{d} g_i^{f_i}$ and (iii) $x \leftarrow x^{2^s}$ $\}$

Step 3. **if** $t = 0$ **then** `return` x **else** `goto` **Step 2**

Remarks 2.3 (1) *In the case $d = 1$ the above algorithm is the usual sliding window exponentiation algorithm. If $w = 1$ then it is just Shamir's trick.*

(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 $0 \leq u \leq s$. 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) *In Step 2 (f) the first time it is $x = 1$, so one multiplication can be saved and only s squarings are needed.*

(4) *If the exponents are written in base 2, then $\mathcal{E}$ consists 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. Then Step 2 (f) is done with one table look-up, one multiplication and w squarings.*

(5) *The changes to Algorithm 2.2 required to work with the NAF are straightforward. A detailed discussion of this case is found in Subsection 3.2. One important consequence of the fact that inversion is free is that the size of $\mathcal{E}$ can be reduced: In fact in Step 2 (f,ii) one can compute either $x \leftarrow x \cdot \prod_{i=1}^{d} g_i^{f_i}$ or $x \leftarrow x / \prod_{i=1}^{d} g_i^{-f_i}$ so only a half of the full table is required.*

3 Complexity analysis

In this section we are concerned only with Algorithm 2.2 and its complexity.

Definition 3.1 *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. Thus at the first iteration of Step (2), substeps (b)–(f) are always performed. To evaluate the number of squarings one should not consider those which can be avoided in the first iteration, which are w minus the expected first value of s.

Algorithm 2.2 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 read 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 2.2 in Step 2 (d) is then $(1-\pi)m - 1$ (remember that the first multiplication can be replaced by an assignment) *i.e.*

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

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

Definition 3.2 *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 3.3 *Algorithm 2.2 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 Reitwiesner's algorithm [26] and Solinas' own algorithm for the Joint Sparse Form, rewrite the exponents right-to-left. In such situations recoding and (multi-)exponentiation cannot be interleaved, and the recoded representations must be stored explicitly. This is a general problem with window methods.*

3.1 Unsigned binary inputs

Here the exponents are written in base 2, *i.e.* $e_{i,j} \in \{0,1\}$. The set $\mathcal{E}$ consists 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. It has cardinality $2^{wd} - 2^{(w-1)d}$. Half of the powers of the base elements g_i can be computed by squarings and all other elements by 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, $s \ge u$ with $1 \le u < w$ 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 the following result:

Theorem 3.4 *Suppose that in Algorithm 2.2 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 3.5 *In the case $w = d = 2$, the set $\mathcal{E}$ consists of the values $g_1^a g_2^b$ with $0 \le a, b \le 3$ 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.*

3.2 Using the NAF

A *non-adjacent form* (abbreviated as NAF) is a signed binary representation of an integer $e = \sum_{j=0}^{n-1} b_j 2^j$ with $b_j \in \{0, \pm 1\}$ and $b_j b_{j-1} = 0$. Each integer admits a NAF, which is uniquely determined. It is the signed binary representation of minimal Hamming weight and it has expected density $1/3$ (see [24] and [2] for proofs).

Reading NAFs through non-sliding windows has been considered already but only for a single NAF (*i.e.* $d = 1$) and not in the case of joint representations: See the paper [13], of which we use some arguments in this subsection. We consider here sliding windows, which lead to lower complexity. Hence, even in the case $d = 1$ our results will complement existing literature. This must not be confused with the wNAF, cfr. §4.2.4.

Theorem 3.6 *Suppose that in Algorithm 2.2 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 $n \cdot \frac{1}{w+\frac{4}{3}\left(1-\left(-\frac{1}{2}\right)^w\right)} - 1$.

Remark 3.7 *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 computing $\mathcal{E}$ by 6 multiplications or multiplications with the inverse is*

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

The remainder of this subsection is devoted to the proof of Theorem 3.6.

First note that 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: It is easy to see that this number is $T_w = (2^{w+2} - 3 - (-1)^w)/6$. Hence, there are $I_w = (2^{w+2} - (-1)^w)/3$ integers in the interval $[-T_w, \ldots, T_w]$. Now $\mathcal{E}$ consists of *all elements of the form $\prod_{i=1}^d g_i^{k_i}$ such that $|k_i| \leq T_w$ 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.* In this way, if in Step 2 (f,ii) the first nonzero f_i is positive we compute $x \leftarrow x \cdot \prod_{i=1}^d g_i^{f_i}$ otherwise we compute $x \leftarrow x / \prod_{i=1}^d g_i^{-f_i}$. Hence we need only $(I_w^d - I_{w-1}^d)/2$ elements in $\mathcal{E}$.

Definition 3.8 *A joint representation of integers in NAF will be called a* joint NAF.

Definition 3.9 *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 $\widehat{e}_i = \sum_{j=0}^{n-1} e_{i,(n-1)-j}\, 2^j$.*

To avoid ambiguity, we define bit-reversing only 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 2.2 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 right to left, 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 $1 \le k \le d$). 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

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

unsigned binary 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 [13] 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 3.10 *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 3.11 *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*

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

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 3.10 we have $\pi_{1,0} = \pi_{1,*} = \frac{1}{2}$ and

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

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 states $\mathcal{S}'_0, \ldots, \mathcal{S}'_d$ defined 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 $0 \leq k \leq d$). 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

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

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` [8] code. With he same software it is immediate to verify that for $d = 1, 2$ and 3 the matrix U_d is 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 3.6.

To estimate the value of s at the first iteration of the main loop, we proceed heuristically. [13, 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 2.2. We then proceed as in the proof of Theorem 3.4, the only difference consisting in the multiplicative factor $\left(\frac{4}{3}\right)^d$. □

3.3 Using the JSF

The Joint Sparse Form has been introduced by Solinas [30] to make Shamir's trick more effective for elliptic curves. It applies however to all groups where inversion is for free. It has been defined only for *pairs* of integers: accordingly we will restrict ourselves to the case $d = 2$ here. 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. For more precise statements see Subsection 4.1, in particular the row for $w = 2$ in Table 2 and Remark 4.1.

In this subsection we prove the following theorem.

Theorem 3.12 *Suppose that in Algorithm 2.2 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 $\frac{3}{8}n - 1$*, 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* $-2 \leq b \leq 2$*;* (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 *and requiring* 10 *multiplications or divisions is*

$$\begin{gathered} \{\, g_1, \quad g_2, \quad g_1 g_2, \quad g_1 g_2^{-1}, \quad g_1 g_2^2, \quad g_1 g_2^{-2}, \\ g_1^2 g_2, \quad g_1^2 g_2^{-1}, \quad g_1^2 g_2^3, \quad g_1^2 g_2^{-3}, \quad g_1^3 g_2^2, \quad g_1^3 g_2^{-2}. \,\} \end{gathered} \tag{5}$$

We assume that the reader is acquainted with the results in Solinas' cited technical report, from which we recall however a few important facts. The joint Hamming weight of the JSF of two integers is minimal among all (un)signed joint binary representations of the same pair of integers. Its average density 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 2.2 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 2.2 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 2.2 scans a joint representation left-to-right in order to form windows, but *consecutive nonzero columns always belong to one window regardless of the direction in which we are scanning the joint representation.* 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.

Therefore to estimate the number of nonzero windows (which corresponds to the number of multiplications performed by Algorithm 2.2 plus one) we scan our input right-to-left. 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 $\pi_0 = \frac{1}{8}$, $\pi_1 = \frac{3}{8}$ and $\pi_2 = \frac{1}{2}$. Hence the expected number of multiplications performed by Algorithm 2.2 is $\frac{3}{8}n - 1$ with n-bit inputs.

According to the defining properties of the JSF, the admissible nonzero colums $\binom{e_{1,j}}{e_{2,j}}$ and windows $\left(\begin{smallmatrix} e_{1,j} & e_{1,j-1} \\ e_{2,j} & e_{2,j-1} \end{smallmatrix}\right)$ 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 $\mathcal{E}$. □

4 Comparisons

4.1 Algorithm 2.2: 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 contains 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 [23]) 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 $w = 3$.

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		JSF	
	$\#\mathcal{E}$ and # Ops		$\#\mathcal{E}$ and # Ops		$\#\mathcal{E}$ and # 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 $d = 2$

w	Unsigned $\#\mathcal{E}$ and # Ops		NAF $\#\mathcal{E}$ and # 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 4.1 *Assume $d = 2$ and consider Table 2. 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 the parameter $w = 2$ is optimal for the exponents sizes which interests us.*

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

With the JSF the parameter $w = 1$ is optimal for $n \leq 64$. Furthermore, using the JSF with $w = 2$ is better than using the NAF with either $w = 2$ or 3 when $124 < n \leq 354$: in the range which concerns us most however using the NAF can be marginally slower but requires fewer precomputations.

Remark 4.2 *In the case $d = 3$ (see Table 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.

Remark 4.3 *As already mentioned, the algorithms of Reitwiesner and Solinas recode the exponents right-to-left, so extra storage must be reserved for the recoded representations. There exists an alternative to the NAF with the same Hamming weight and which can be computed from left to right [17] by a simple algorithm. However this representation dispenses with the non-adjacency property, which has a very negative impact on memory usage. For instance 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 instead of 8 or 12. Therefore both the total memory usage of Algorithm 2.2 combined with this recoding and its running time would be worse than the variants we analysed. For this reason it should not be considered.*

4.2 Interleaved exponentiation and exponent representations

A multi-exponentiation algorithm called *interleaved exponentiation* has been described by Möller [23]. It is better understood in terms of exponent recording, and it is nothing but Algorithm 2.1 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 generalization of the left-to-right exponentiation algorithm computes $x := \prod_{i=1}^{d} g_i^{e_i}$.

Algorithm 4.4 Left-to-right interleaved multi-exponentiation

INPUT: Group elements $g_1, \ldots, g_d$ 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** $j = n-1 \ldots 0$ **do** {

(a) $x \leftarrow x^2$ [Skip at first iteration]

for $i = 1 \ldots d$ **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, balancing a low density of the representations with 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). We see now four candidates for the representations.

4.2.1 Radix-*r* representation

A first possibility is offered by radix-r recoding where $r = 2^w$ is a power of 2. We embed this representation into a redundant base 2: If $e = \sum_{i=0}^{n-1} b_i 2^i$ we put

$$b'_k = \begin{cases} \sum_{\ell=0}^{w-1} b_{wk+\ell} 2^\ell & \text{if } k \equiv 0 \pmod{w} \\ 0 & \text{otherwise,} \end{cases}$$

for $0 \leq k \leq n-1$ and assuming $b_i = 0$ for $i \geq n$, then consider $e = \sum_{k=0}^{\lceil n/w \rceil - 1} b'_{kw} 2^{kw} = \sum_{i=0}^{n-1} b'_i 2^i$. The last representation is that which is actually used to represent the exponents in Algorithm 4.4. This is very easy to implement, for exponents are scanned online w bits at a time and all blocks of multiplications are done only every w squarings. The density of a radix-r representation is $\frac{r-1}{r}$ and so Algorithm 4.4 requires about $\frac{dn}{w} \frac{2^w-1}{2^w}$ multiplications and n squarings.

4.2.2 The generalized non-adjacent form

A better alternative, assuming free inversion, can be the *generalized non-adjacent form*, or GNAF, which is a *signed* radix-r recoding. A radix-r GNAF of the integer e is a representation $e = \sum_{i=0}^{n-1} b_i r^i$ with $-r < b_i < r$ and satisfying the following two properties

(GNAF-1) $|b_i + b_{i+1}| < r$ for all i.

(GNAF-2) If $b_i\, b_{i+1} < 0$ then $|b_i| < |b_{i+1}|$.

This form coincides with the definition of the NAF when $r = 2$. Moreover, as for the NAF, it can be proven that this form is unique and has minimal Hamming weight among signed radix-r representations [7]. Here we consider only the case that r is a power of 2 and we embed the representation into a redundant base 2 one as in the previous paragraph.

Using the GNAF 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 a multi-exponentiation algorithm requiring $\frac{dn}{w}\frac{2^w-1}{2^w+1}$ multiplications and about n squarings to compute the product of d powers with n-bit exponents.

The GNAF is recoded right-to-left, and thus cannot be used online.

There exists a left-to-right recoding with the same weight as the GNAF [18].

4.2.3 Width-*w* left-to-right sliding windows

One can let a sliding window of size w scan right-to-left the binary representation of an integer, skipping zeros and reading the value contained in the window only if a bit equal to 1 is found. This gives a recoding $e = \sum_{j=0}^{n-1} b_j\, 2^j$ where the coefficients b_j are either zero or odd, satisfy $0 \leq b_j < 2^w$ and of any w consecutive of them at most one is nonzero. It is very well known that this representation has density $1/(w+1)$.

We want however a *left-to-right sliding window algorithm*. In the most naïve way it produces a recoding where the coefficients b_j are either zero or and satisfy $2^{w-1} \leq b_j < 2^w$, except perhaps for the least significant nonzero coefficient. Clearly this representation has the same expected density $1/(w+1)$. If however we do not include always w bits in the window but *only as many bit as possible as long as they are at most w and not only the most significant one but also the least significant one is* 1, we get a recoding where the b_j are zero or odd and satisfy $0 \leq b_j < 2^w$ and with the same weight as before. Note that two nonzero coefficients need no longer to be separated by at least $w-1$ zero coefficients. We call this recoding the *width w sliding window recoding*, or wSWR for short. It can be used online but some care is required in the implementation.

The wSWR is better than radix-2^w as long as $w > 1$ (and equal if $w = 1$) and also better than the GNAF if $w \geq 3$. Moreover only about a half of the elements need to be precomputed compared to those needed for the radix-2^w and GNAF for the same w.

The result is that the wSWR is preferable over using the radix-2^w form or the GNAF for the optimal value of w for a given exponent size (which is either 4 or 5 for $160 \leq n \leq 256$).

4.2.4 The flexible window exponentiation algorithm and the *w*NAF

Cohen's *flexible window* exponentiation algorithm [10, 9] which was also proposed independently by Solinas [28, 29] consists in the application of Algorithm 4.4 with $d = 1$ to the *wNAF* of the exponent. The wNAF of the integer e is a representation $e = \sum_{j=0}^{n-1} b_j\, 2^j$ where the integer coefficients b_j satisfy the following two conditions:

(*w*NAF-1) Either $b_j = 0$ or b_j is odd and $|b_j| \leq 2^w$.

(*w*NAF-2) Of any $w+1$ consecutive coefficients $b_{j+w}, \ldots, b_j$ at most one is nonzero.

It is also called *width-$(w+1)$ NAF* and it must not be confused with the GNAF. The special case $w = 1$ is the usual NAF. 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. The density of the representation is $1/(w+2)$. This immediately leads to an exponentiation algorithm requiring about $n/(w+2)$ multiplications for an n-bit exponent. The wNAF recoding algorithm works right-to-left, therefore it cannot be used online.

Remark 4.5 *In general the recoding of choice for Algorithm 4.4 is the wSWR if inversion is expensive, and the wNAF otherwise, because of their better densities and smaller precomputation tables*

	$d=2$						$d=3$					
n	Algorithm 2.2 (base-2) $\#\mathcal{E}$ and # Ops (w)			Algorithm 4.4 (wSWR) $\#\mathcal{E}$ and # Ops (w)			Algorithm 2.2 (base-2) $\#\mathcal{E}$ and # Ops (w)			Algorithm 4.4 (wSWR) $\#\mathcal{E}$ 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)
96	12	50.14	(w=2)	16	53.40	(w=3)	7	87	(w=1)	12	83	(w=3)
128	12	63.86	(w=2)	16	66.20	(w=3)	56	111.86	(w=2)	12	107	(w=3)
160	12	77.57	(w=2)	16	79	(w=3)	56	126.67	(w=2)	12	131	(w=3)
192	12	91.28	(w=2)	16	91.80	(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

n	Algorithm 2.2 (NAF) $\#\mathcal{E}$ and # Ops (w)			Algorithm 2.2 (JSF) $\#\mathcal{E}$ and # Ops (w)			Algorithm 4.4 (wNAF) $\#\mathcal{E}$ and # Ops (w)		
56	8	27.81	(w=2)	4	29	(w=1)	8	29.40	(w=3)
64	8	31.07	(w=2)	4	33	(w=1)	8	32.60	(w=3)
80	8	37.59	(w=2)	12	39	(w=2)	8	39	(w=3)
96	8	44.11	(w=2)	12	45	(w=2)	8	45.40	(w=3)
128	8	57.15	(w=2)	12	57	(w=2)	16	57.66	(w=4)
160	8	70.19	(w=2)	12	69	(w=2)	16	68.33	(w=4)
192	8	83.22	(w=2)	12	81	(w=2)	16	79	(w=4)
240	8	102.78	(w=2)	12	99	(w=2)	16	95	(w=4)
256	8	109.30	(w=2)	12	105	(w=2)	16	100.33	(w=4)

Table 5: Complexity of multi-exponentiation using signed representations, $d = 2$

with respect to the simpler radix-r form and GNAF. The wSWR can be a good backup choice for Algorithm 4.4 even if inversion is free if memory usage is critical. For the corresponding penalty hit see the tables in the next subsection.

4.3 Comparing the two algorithms

If inversion in the group is not for free, we recode online the exponents as wSWR's in Algorithm 4.4. One requires then d squarings and $d(2^{w-1}-1)$ multiplications in the precomputation stage and $\frac{dn}{w+1}-1$ multiplications and about $n-1$ squarings in the main loop (note that the first multiplication is just a variable assignment). In Table 4 we add the total number of operations in the precomputation stage to the number of multiplications in the main loop of the algorithms. (This is the same performance parameter used before.) In each case the value of w which minimizes the running time is chosen.

If inversion in the group is cheap, we write the exponents as wNAF's. Algorithm 4.4 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. Tables 5 and 6 collect the complexity data for these algorithms which exploit signed representations in the cases $d = 2$ and $d = 3$ respectively.

n	Algorithm 2.2 (NAF) #$\mathcal{E}$ and # Ops (w)			Algorithm 4.4 (wNAF) #$\mathcal{E}$ and # Ops (w)		
56	13	48.40	(w=1)	12	44.60	(w=3)
64	13	54.03	(w=1)	12	49.40	(w=3)
80	13	65.29	(w=1)	12	59	(w=3)
96	13	76.55	(w=1)	12	68.60	(w=3)
128	13	99.07	(w=1)	24	87	(w=4)
160	49	115.57	(w=2)	24	103	(w=4)
192	49	129.69	(w=2)	24	119	(w=4)
240	49	150.86	(w=2)	24	143	(w=4)
256	49	157.91	(w=2)	24	151	(w=4)

Table 6: Complexity of multi-exponentiation using signed representations, $d = 3$

Remarks 4.6 (1) *Only implementation can decide which of the algorithms is fastest for a particular purpose if the number of operations is similar. In these cases memory usage can be the decisive factor. In the case of double exponentiations with unsigned representations Algorithm 2.2 seems the best choice, either yielding better performance than Algorithm 4.4 or yielding similar performance while requiring less memory.*

(2) *Algorithm 2.2 with the NAF or with the JSF seems be preferable to Algorithm 4.4 for double exponentiations with exponents from* 160 *to* 256 *bits in memory constrained environments. In particular the use of the NAF reduces considerably the number of required precomputations, saving RAM and, in the case of one fixed base, also ROM. The use of the NAF is undisputably preferable also for smaller bit sizes, i.e.* 80 *to* 128 *bits, and this gives the method of choice for implementing single exponentiations in groups with an automorphism of degree* 2*, such as trace zero varieties or XTR subgroups as described in Section 5.*

(3) *For triple exponentiations Algorithm 4.4 seems always preferable with unsigned representations (using the wSWR) and $n \geq 128$ or with signed representations (using the wNAF).*

5 Applications

In this section we show a few applications of the above multi-exponentiation algorithms.

5.1 Elliptic and hyperelliptic curves

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

The natural application of Algorithm 2.2 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. Hence $d = 2$ and for the current applications (where exponent sizes are between 160 and 256 bits) we have already seen that the optimal value of the parameter is $w = 2$.

In the ecc case we observe that mixed coordinate systems can be used exactly as in [11], as we have sequences of repeated doublings (always at least two of them) alternated with single additions of points from a precomputed table. We compute directly the double scalar product, whereas Cohen et al. compute the two scalar products separately and then they add the results: For the fixed base scalar multiplication they use essentially a comb method and for the variable base scalar product the flexible window algorithm (see § 4.2.4). For brevity we call the resulting method the CMO method.

We work out the costs in the case of an elliptic curve defined over a prime field of about 2^n elements, where $n = 160$ and 240. We shall denote by M, S and I the timings of a multiplication, of a squaring and of an inversion respectively in the base field of our curve.

In [11] *five* coordinate systems for elliptic curves are described, namely affine ($\mathcal{A}$) projective ($\mathcal{P}$), jacobian ($\mathcal{J}$), Chudnovsky jacobian ($\mathcal{J}^c$) and modified jacobian ($\mathcal{J}^m$) coordinates. Adding points in coordinate systems $\mathcal{C}_1$ and $\mathcal{C}_2$ yielding a result in system $\mathcal{C}_3$ is denoted by $\mathcal{C}_1 + \mathcal{C}_2 = \mathcal{C}_3$, doubling is notated as $2\mathcal{C}_1 = \mathcal{C}_2$. If the coordinate systems are the same one writes simply $\mathcal{C}_1 + \mathcal{C}_1$ and $2\mathcal{C}_1$. Timings are denoted by $t(\cdots)$. The gist is that all these operations have different costs (explicitly given in the cited paper), so a different system can be selected for each operation. Three coodinate systems are employed: the first system ($\mathcal{C}_1$) is used for all the doublings but the final one before an addition with a precomputed point, the second one ($\mathcal{C}_2$) for the result of a final doubling, and the third one ($\mathcal{C}_3$) for the precomputed points. Therefore, if Algorithm 2.2 requires N_+ additions and N_2 doublings, its total cost is

$$(N_2 - N_+)t(2\mathcal{C}_1) + N_+\big(t(2\mathcal{C}_1 = \mathcal{C}_2) + t(\mathcal{C}_2 + \mathcal{C}_3 = \mathcal{C}_1)\big) + t_p$$

where t_p is the cost of the precomputations. Since doublings in $\mathcal{C}_1$ are the most frequent operation, $\mathcal{C}_1$ should be the system with fastest doubling, *i.e.* $\mathcal{C}_1 = \mathcal{J}^m$ with cost $t(2\mathcal{J}^m) = 4M + 4S$. The result of a final doubling is done in $\mathcal{C}_2 = \mathcal{J}$, and $t(2\mathcal{J}^m = \mathcal{J}) = 3M + 4S$.

Now we consider two possible choices for $\mathcal{C}$: $\mathcal{A}$ and $\mathcal{J}^c$.

If $\mathcal{C}_3 = \mathcal{A}$, all additions with a precomputed point are as fast as possible: $t(\mathcal{J} + \mathcal{A} = \mathcal{J}^m) = 9M + 5S$. The total running-time is

$$N_2(4M + 4S) + N_+(8M + 5S) + t_p(\mathcal{A}).$$

We proceed to estimate $t_p = t_p(\mathcal{A})$. Since g_1 is the fixed point of the system, we assume that it is given in $\mathcal{A}$, whereas g_2, belonging to the signature to be verified, is in $\mathcal{P}$. In the NAF (resp. JSF) case we have 4 (resp. 6) additions of type $\mathcal{A} + \mathcal{J} = \mathcal{P}$, and 2 (resp. 4) of type $\mathcal{P} + \mathcal{P}$. Then we convert the results to affine coordinates, which can be done naïvely by inverting 7 (resp. 11) field elements and performing 14 (resp. 22) multiplications. Using Montgomery's trick, one can invert m numbers by one inversion and $3(m-1)$ multiplications. So in the NAF case the total precomputation cost is $t_p(\mathcal{A}, \mathrm{NAF}) = 4\,t(\mathcal{A} + \mathcal{P} = \mathcal{P}) + 2\,t(\mathcal{P} + \mathcal{P}) + (14 + 3 \cdot 6)\,M + I = 4\,(8M + 2S) + 2\,(12M + 2S) + 32M + I = 88M + 12S + I$ whereas in the JSF case it is $t_p(\mathcal{A}, \mathrm{JSF}) = 148M + 20S + I$.

If $\mathcal{C}_3 = \mathcal{J}^c$, we have $t(\mathcal{J} + \mathcal{J}^c = \mathcal{J}^m) = 12M + 5S$ and the total running-time is

$$N_2(4M + 4S) + N_+(11M + 5S) + t_p(\mathcal{J}^c).$$

To estimate $t_p(\mathcal{J}^c)$ we first have to convert g_2 from $\mathcal{P}$ to $\mathcal{J}^c$, requiring $1M + 1S$. Then in the NAF (resp. JSF) case we have 4 (resp. 6) additions of type $\mathcal{A} + \mathcal{J}^c = \mathcal{J}^c$, and 2 (resp. 4) of type $\mathcal{J}^c + \mathcal{J}^c$. The total precomputation costs are $t_p(\mathcal{J}^c, \mathrm{NAF}) = (1M + 1S) + 4\,(8M + 3S) + 2\,(11M + 3S) = 55M + 19S$ and $t_p(\mathcal{J}^c, \mathrm{JSF}) = (1M + 1S) + 6\,(8M + 3S) + 4\,(11M + 3S) = 93M + 31S$.

Algorithm 2.2 with the NAF has $N_2 = n - \frac{10}{9}$ and $N_+ = \frac{11}{27}n - 1$ (cfr. Theorem 3.6). With the JSF $N_2 = n - \frac{3}{2}$ and $N_+ = \frac{3}{8}n - 1$.

In what follows we shall assume $S \approx 0.8M$, which is confirmed by experience.

The scalar multiplication of the variable point in the CMO method for $n = 160$ and $n = 240$ has costs $1488.4M + 4I$ and $2228 + 4I$ if the coordinates $(\mathcal{J}^m, \mathcal{J}, \mathcal{A})$ are used, and $1610.2M$ and $2400M$

respectively with the coordinates $(\mathcal{J}^m, \mathcal{J}, \mathcal{J}^c)$. The cost of the scalar multiplication of the fixed point is $454.4M$ and $620.8M$ in the cases $n = 160$ and 240 respectively. We have computed these values by adapting their comb algorithm also to $n = 240$ (in which case it requires a precomputed table of 93 elements), assuming that the precomputed table is in $\mathcal{A}$, and computing the results in $\mathcal{J}$. For $n = 160$ there are 15 doublings ($2\mathcal{J}$) and 31 additions ($\mathcal{J} + \mathcal{A} = \mathcal{J}$). If $n = 240$ the doublings remain 15 and the additions increase to 47. Note that our cost for $n = 160$ is better than the value (474) stated in [10]. A further addition in jacobian coordinates yielding a projective result is needed, with cost $12M + 2S \approx 13.6M$.

Upon putting all pieces together, we get the results of Table 7.

n	Coordinate systems	Algorithm 2.2 (NAF)	Algorithm 2.2 (JSF)	CMO method
160	$(\mathcal{J}^m, \mathcal{J}, \mathcal{A})$	$2011.82M + I$	$2013.2M + I$	$1956.4M + 4I$
	$(\mathcal{J}^m, \mathcal{J}, \mathcal{J}^c)$	$2177.03M$	$2144M$	$2078.2M$
240	$(\mathcal{J}^m, \mathcal{J}, \mathcal{A})$	$2978.93M + I$	$2949.2 + IM$	$2862.4M + 4I$
	$(\mathcal{J}^m, \mathcal{J}, \mathcal{J}^c)$	$3256.86M$	$3170M$	$3034.4M$

Table 7: Comparison of double scalar multiplication methods on `ecc`

Whereas on modern CPUs $I \approx 20M$ for $n = 160$ and $I \approx 40M$ for $n = 240$, on smart card architectures these values are much worse, even 50 for $n = 160$ and 100 for $n = 240$, or higher. One sees at once that the different methods have comparable performance. One advantage of our method is that it does not require a table of 62 to 93 fixed precomputed points to be stored in ROM, as in the CMO method, where the cost of computing those points has not *been taken into account. The size of such tables varies from 2480 bytes for $n = 160$ to 5580 bytes for $n = 240$. In the CMO method the fixed base scalar multiplication needs to be reengineered for each exponent range for best performance. This is not necessary with our method. The conclusion is that our method can be used much more efficiently in systems which do not assume a fixed point (this is optimal if fast system configuration is an issue).*

We note that mixed coordinate systems also exist for hyperelliptic curves of genus 2 [22].

5.2 Trace zero varieties

Trace zero varieties are abelian varieties constructed essentially by Weil Descent from other varieties, such as elliptic curves [25, 14] or Jacobians of hyperelliptic curves [20, 21].

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. As usual we want to choose it so that it has a cofactor in G as small as possible, *i.e.* $\ell \approx p^{2g}$.

It has been noted that for $g(d-1) \leq 4$ the best attacks known to work on trace zero varieties have complexity $O(\sqrt{G_0})$ [25, 21].

In what follows we consider only the case where $d = 3$ *for simplicity.*

As we require the same level of security offered by, say, elliptic curves over fields of 160 bits, we have $\ell \approx 2^{160}$ also for trace zero varieties and the field $\mathbb{F}_p$ must satisfy $p \approx 2^{80/g}$.

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 [5, 4]), and by the presence of the automorphism σ of small degree.

The latter fact enables one to speed-up even *single* exponentiations. Instead of using single scalars to compute $r \cdot D$ for an ideal class D, one considers a pair (r_0, r_1) of scalars bounded by some quantity which is $O(p^g)$, and computes the double scalar product $r_0 \cdot D + r_1 \cdot \sigma(D)$. For r_0 and r_1 suitably bounded (see [25, 21]) all such double scalar products are distinct. Shamir's trick can be used and the result is that the number of doublings needed in cryptographic operations is roughly halved with respect to generic elliptic and hyperelliptic curves. Further savings can be achieved by the use of Algorithms 2.2 and 4.4, 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 bounding, if possible, the coefficients by kp^g 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 we observe that the results in [27, §2 and §5] actually apply to any automorphism of the group with given degree 2 minimal polynomial. In particular they apply to σ on the trace-zero variety with equation $\sigma^2 + \sigma + 1 = 0$ and a bound $O(p^g)$ on r_i, u_i holds.

To perform the quadruple exponentiation we suggest the use of Algorithm 4.4 and the wNAF with $d = 4$. Since two of the base divisors are images of D and E under σ, and the cost of the Frobenius is on average approximately 1/25-th of the cost of an addition or of a doubling (this value is obtained heuristically by considering the cases $g = 1$ and 2), we first precompute the necessary multiples of D and E, then we apply σ to the resulting sets. To determine the optimal value of w we have thus to minimize the number of operations, which is

$$n + 2\left(1 + \frac{1}{25}\right)(2^{w-1} - [1 = w]) + \frac{4n}{w+2} - 1$$

where $[\text{expr}]$ evaluates to 1 is expr is true, to 0 otherwise. For $n = \log_2(p^g) \approx 80$ the minimum 148.97 is achieved for $w = 4$. If $w = 3$ the amount of operations is 151.32. For a minimal trade-off, one can also store only the multiples of D and E and apply the frobenius on-the-fly when multiples of $\sigma(D)$ or $\sigma(E)$ are needed.

Let us consider signature verification using a trace zero variety arising from a genus g curve over a finite field of about $2^{80/g}$ *elements (so* $n = 80$*): It may be done with about* 150 *group operations. For comparison, using the ECDSA or hyperelliptic curve variants thereof of comparable security requires* *a minimum of* 229 *group operations (see Table 5 with* $n = 160$ *and* $d = 2$*). Furthermore, one should note that group operations on the trace zero variety are faster than on a elliptic or hyperelliptic curve of comparable size.*

5.3 XTR

The XTR cryptosystem was initially proposed by Lenstra and Verheul [32] 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. Recently, Lenstra and Stam [31] 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 extension. 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 [3]). 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 is $\mathbb{F}_{p^3}$ and 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.

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 about trace zero varieties 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.

References

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

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

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

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

[5] R.E. Blahut, *Fast algorithms for digital signal processing*. Addison-Wesley, Reading, MA, 1985.

[6] A.D. Booth, *A signed binary multiplication technique*. The Quarterly Journal of Mechanics and Applied Mathematics **4** (1951), 236–240.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

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

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

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

[15] 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.

[16] 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.

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

[18] 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.

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

[20] T. Lange, *Efficient Arithmetic on Hyperelliptic Curves*. Ph.D. Thesis, Universität Essen, 2001.

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

[22] T. Lange, *Weighted Coordinates on Genus 2 Hyperelliptic Curves*. Preprint.
See: `http://eprint.iacr.org/`

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

[24] F. Morain and J. Olivos, *Speeding up the computations on an elliptic curve using addition-subtraction chains*. RAIRO Inform. Theory **24** (1990), 531–543.

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

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

[27] 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.

[28] 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.

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

[30] 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`

[31] 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.

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

[33] S.-M. Yen, C.-S. Laih and A.K. Lenstra, *Multi-exponentiation*. IEE proceedings: computers and digital techniques vol. 141, No. 6, november 1994.