Appears in P. J. Lee, C. H. Lim (Eds.): *Information Security and Cryptology – ICISC 2002,* Springer-Verlag LNCS 2587, pp. 298–312, ISBN 3-540-00716-4, 2003.

Improved Techniques for Fast Exponentiation

Bodo Möller

Technische Universität Darmstadt, Fachbereich Informatik
moeller@cdc.informatik.tu-darmstadt.de

Abstract. We present improvements to algorithms for efficient exponentiation. The *fractional window* technique is a generalization of the sliding window and window NAF approach; it can be used to improve performance in devices with limited storage. *Window NAF splitting* is an efficient technique for exponentiation with precomputation for fixed bases in groups where inversion is easy (e.g. elliptic curves).

1 Introduction

Many schemes in public key cryptography require computing powers

$$g^e$$

(exponentiation) or power products

$$\prod_{1 \le j \le k} g_j^{e_j}$$

(multi-exponentiation) in a commutative semigroup G with neutral element 1_G, e.g. in the group $(\mathbb{Z}/n\mathbb{Z})^*$ or more generally in the multiplicative semigroup $(\mathbb{Z}/n\mathbb{Z})$ for some integer n, or in the group of rational points on an elliptic curve over a finite field. The exponents e, e_j are positive integers with a typical length of a few hundred or a few thousand bits.

Bases $g, g_j \in G$ sometimes are fixed between many computations. With fixed bases, it is often advantageous to perform a single time a possibly relatively expensive *precomputation* in order to prepare a table that can be used to speed up exponentiations involving those bases. (For multi-exponentiation, some of the bases may be fixed while others are variable: for example, verifying a DSA [11] or ECDSA [1] signature involves computing the product of two powers where one of the bases is part of *domain parameters* that can be shared between a large number of signers while the other base is specific to a single signer.)

In this paper, we look at efficient algorithms for exponentiation and multi-exponentiation based on either just multiplication in the given semigroup or optionally, in the case of a group, on multiplication and division. This amounts to constructing addition chains or addition-subtraction chains for the exponent e for exponentiation, and to constructing vector addition chains or vector addition-subtraction chains for the vector of exponents $(e_1, \ldots, e_k)$ for multi-exponentiation (see e.g. the survey [4]).

For purposes of performance analysis, we distinguish between squarings and general multiplications, as the former can often be implemented more efficiently. If we allow division, our performance analysis does not distinguish between divisions and multiplications; this is reasonable e.g. for point groups of elliptic curves, where inversion is almost immediate. If inversion is expensive, the group should be treated as a semigroup, i.e. inversion should be avoided.

Section 2 gives a framework for exponentiation algorithms. In section 3, we show how it can be adapted to multi-exponentiation by using interleaved exponentiation. In section 4, we describe within the framework the sliding window exponentiation method and the window NAF exponentiation method. We then present improvements to the state of the art: section 5 describes *fractional windows,* a technique that closes a gap in the sliding window and window NAF methods and is useful for devices with limited storage; section 6 describes *window NAF splitting,* a technique for exponentiation with precomputation for fixed bases in groups where inversion is easy. Then, in section 7, we discuss how the exponent representations employed by our techniques can be implemented with small memory overhead. Finally, section 8 gives our conclusions.

1.1 Notation

If c is a non-negative integer, $\mathrm{LSB}_m(c) = c \bmod 2^m$ is the integer formed by the m least significant bits of c, and $\mathrm{LSB}(c) = \mathrm{LSB}_1(c)$.

When writing digits, we use the convention that $\overline{b}$ denotes a digit of value $-b$ where b is understood to be a positive integer; for example, $10\overline{1}_2 = 2^2 - 2^0 = 3$.

2 A Framework for Exponentiation

Many algorithms for computing g^e for arbitrary large integers e fit into one of two variants of a common framework, which we describe in this section. Exponents e are represented in base 2 as

$$e = \sum_{0 \le i \le \ell} b_i \cdot 2^i,$$

using digits $b_i \in B \cup \{0\}$ where B is some set of integers with $1 \in B$. We call this a B-representation of e. Details of it are intrinsic to the specific exponentiation method. (Note that for given B, B-representations are usually not canonical.) The elements of B must be non-negative unless G is a group where inversion is possible in reasonable time. Given a B-representation, *left-to-right* or *right-to-left* methods can be used. Left-to-right methods look at the elements of b_i starting at b_ℓ and proceed down to b_0; right-to-left methods start at b_0 and proceed up to b_ℓ. Depending on how the values b_i can be obtained from an input value e, it may be easy to compute them on the fly instead of storing the B-representation beforehand. Left-to-right methods and right-to-left methods can be considered dual to each other (cf. the duality observation for representations of arbitrary addition chains as directed multi-graphs in [6, p. 481]); both involve two stages.

2.1 Left-to-Right Methods

For *left-to-right* methods, first, in the *precomputation stage,* powers g^b for all $b \in B$ are computed and stored; if division in G is permissible and $|b| \in B$ for each $b \in B$, then it suffices to precompute g^b for those $b \in B$ that are positive. We refer to this collection of precomputed powers g^b as the *precomputed table.* How to implement the precomputation stage efficiently depends on the specific choice of B. In certain semigroups, in order to accelerate the evaluation stage, precomputed elements can be represented in a special way such that multiplications with these elements take less time (for example, precomputed points on an elliptic curve may be converted from projective into affine representation [3]). Note that if both the base element g and the digit set B are fixed, then the precomputation stage need not be repeated for multiple exponentiations if the precomputed table is kept in memory. In cases without such fixed precomputation, B is usually a set consisting of small integers such that the precomputation stage requires only a moderate amount of time. If

$$B = \{1, 3, \ldots, \beta\} \quad \text{or} \quad B = \{\pm 1, \pm 3, \ldots, \pm\beta\}$$

with $\beta \geq 3$ odd, the precomputation stage can be implemented with one squaring and $(\beta - 1)/2$ multiplications as follows: first compute g^2; then iteratively compute $g^3 = g \cdot g^2$, $\ldots$, $g^\beta = g^{\beta-2} \cdot g^2$. This applies to all encoding techniques we will present in later sections.

In the *evaluation stage* (or *left-to-right stage*) of a left-to-right method, given the precomputed table and the representation of e as digits b_i, the following algorithm is executed to compute the desired power from the precomputed elements g^b.

$A \leftarrow 1_G$
for $i = \ell$ down to 0 **do**
  $A \leftarrow A^2$
  if $b_i \neq 0$ **then**
    $A \leftarrow A \cdot g^{b_i}$
return A

If division is permissible, the following modified algorithm can be used:

$A \leftarrow 1_G$
for $i = \ell$ down to 0 **do**
  $A \leftarrow A^2$
  if $b_i \neq 0$ **then**
    if $b_i > 0$ **then**
      $A \leftarrow A \cdot g^{b_i}$
    else
      $A \leftarrow A/g^{|b_i|}$
return A

Note that in these algorithms squarings can be omitted while A is 1_G; similarly, the first multiplication or division can be replaced by an assignment or an assignment followed by inversion of A.

2.2 Right-to-Left Methods

For *right-to-left* methods, no precomputed elements are used. Instead, first the *right-to-left stage* yields values in a number of accumulators A_b, one for each positive element $b \in B$. If division is permissible, B may contain negative digits; we require that $|b| \in B$ for each $b \in B$. Second, the *result stage* combines the accumulator values to obtain the final result. The following algorithm description comprises both stages, but the result stage is condensed into just the "return" line: how to implement it efficiently depends on the specific choice of B. For brevity, we show just the algorithm with division (if B does not contain negative digits, the "else"-branch will never be taken and can be left out).

$\{$*right-to-left stage*$\}$
for $b \in B$ **do**
  if $b > 0$ **then**
    $A_b \leftarrow 1_G$
$A \leftarrow g$
for $i = 0$ to ℓ **do**
  if $b_i \neq 0$ **then**
    if $b_i > 0$ **then**
      $A_{b_i} \leftarrow A_{b_i} \cdot A$
    else
      $A_{|b_i|} \leftarrow A_{|b_i|}/A$
  $A \leftarrow A^2$

$\{$*result stage*$\}$
return $\prod_{\substack{b \in B \\ b>0}} {A_b}^b$

The squaring operation may be omitted in the final iteration as the resulting value of A will not be used. For each A_b, the first multiplication or division can be replaced by an assignment or an assignment followed by inversion (implementations can use flags to keep track which of the A_b still contain the values 1_G).

If

$$B = \{1, 3, \ldots, \beta\} \quad \text{or} \quad B = \{\pm 1, \pm 3, \ldots, \pm\beta\}$$

with β odd (as in all encoding techniques we will present in later sections), the result stage can be implemented as follows ([19], [6, exercise 4.6.3-9]):

for $b = \beta$ to 3 step -2 **do**
  $A_{b-3} \leftarrow A_{b-3} \cdot A_b$
  $A_1 \leftarrow A_1 \cdot A_b^2$
return A_1

This algorithm requires $(\beta - 1)/2$ squarings and $\beta - 1$ multiplications.

3 Multi-Exponentiation by Interleaved Exponentiation

Let a B_j-representation

$$e_j = \sum_{0 \le i \le \ell_j} b_{j,i} \cdot 2^i$$

be given for each of the exponents in a power product

$$\prod_{1 \le j \le k} g_j^{e_j},$$

where each B_j is a digit set as in section 2. Then the multi-exponentiation can be performed by *interleaving* the left-to-right algorithms for the individual exponentiations $g_j^{e_j}$ [10]. For each j, precomputed elements g_j^b are needed as in section 2.1.

Let ℓ be the maximum of the ℓ_j. We may assume that $\ell = \ell_1 = \ldots = \ell_k$ (pad with leading zeros where necessary). If division is permissible, interleaved exponentiation to compute $\prod_{1 \le j \le k} g_j^{e_j}$ can be performed as follows:

$A \leftarrow 1_G$
for $i = \ell$ down to 0 **do**
  $A \leftarrow A^2$
  for $j = 1$ to k **do**
    if $b_{j,i} \neq 0$ **then**
      if $b_{j,i} > 0$ **then**
        $A \leftarrow A \cdot g_j^{b_{j,i}}$
      else
        $A \leftarrow A / g_j^{|b_{j,i}|}$
return A

As in section 2.1, initial squarings can be omitted while A is 1_G, and the first multiplication or division can be replaced by an assignment possibly followed by inversion. The algorithm variant without division is obvious.

4 Sliding Window Exponentiation and Window NAF Exponentiation

A well-known method for exponentiation in semigroups is the *sliding window* technique (cf. [18, p. 912] and [4, section 3]). The encoding is based on a parameter w, a small positive integer called the *window size.* The digit set is $B = \{1, 3, \ldots, 2^w - 1\}$. Encodings using these digits can be computed on the fly by scanning the ordinary binary representation of the exponent either in left-to-right or in right-to-left direction: in the respective direction, repeatedly look out for the first non-zero bit and then examine the sequence of w bits starting at this bit position; one of the odd digits in B suffices to cover these w bits. For example, given $e = 88 = 1011000_2$, left-to-right scanning using window size $w = 3$ yields

$$\underline{101}\,\underline{000}_2 \quad \rightarrow \quad 51000_2,$$

and right-to-left scanning also using window size $w = 3$ yields

$$\underline{\quad 1}\,\underline{011}\,000_2 \quad \rightarrow \quad 1003000_2.$$

The average density of non-zero digits in the resulting representation

$$e = \sum_{0 \le i \le \ell} b_i \cdot 2^i$$

is $1/(w+1)$ for $e \rightarrow \infty$. The length is at most that of the binary representation, i.e. a maximum index ℓ suffices to represent any $\ell + 1$-bit exponent.

Including negative digits into B allows decreasing the average density: a $\{\pm 1\}$-representation such that no two adjacent digits are non-zero ("property M" from [13]) is called a *non-adjacent form* or *NAF*. More generally, let

$$B = \big\{ \pm 1, \pm 3, \ldots, \pm(2^w - 1) \big\};$$

then the following algorithm (from [17]) generates a B-representation of e such that at most one of any $w + 1$ consecutive digits is non-zero. There is a unique representation with this property, the *width-*$(w+1)$ *NAF* of e. We use the term *window NAF* (wNAF) if w is understood. This idea is also known as the *signed window* approach; $w + 1$ can be considered the window size.

$c \leftarrow e$
$i \leftarrow 0$
while $c > 0$ **do**
  if $\mathrm{LSB}(c) = 1$ **then**
    $b \leftarrow \mathrm{LSB}_{w+1}(c)$
    if $b \ge 2^w$ **then**
      $b \leftarrow b - 2^{w+1}$
    $c \leftarrow c - b$
  else
    $b \leftarrow 0$
  $b_i \leftarrow b$; $i \leftarrow i + 1$
  $c \leftarrow c/2$
return $b_{i-1}, \ldots, b_0$

Width-$(w+1)$ NAFs have an average density of $1/(w+2)$ for $e \rightarrow \infty$ ([15], [16], [9], [17]). Compared with the binary representation, the length can grow by one at most, so a maximum index ℓ is sufficient to represent any ℓ-bit exponent.

For left-to-right exponentiation using the sliding window or window NAF technique, the precomputation stage has to compute g^b for $b \in \{1, 3, \ldots, 2^w - 1\}$, which for $w > 1$ can be achieved with one squaring and $2^{w-1} - 1$ multiplications (see section 2.1).

For right-to-left exponentiation using the sliding window or window NAF technique, the result stage has to compute

$$\prod_{b \in \{1,3,\ldots,2^w-1\}} {A_b}^b$$

given accumulator values A_b resulting from the right-to-left stage. This can be done in $2^{w-1} - 1$ squarings and $2^w - 2$ multiplications (see section 2.2).

4.1 Modified Window NAFs

The efficiency of exponentiation given a B-representation depends on the number of non-zero digits and the length of the representation (i.e. the minimum index I such that $b_i = 0$ for $i \geq I$). Window NAFs may have increased length compared with the ordinary binary representation: e.g., the (width-2) NAF for $3 = 11_2$ is $10\overline{1}_2$, and the NAF for $7 = 111_2$ is $100\overline{1}_2$.

Such length expansion can easily be avoided in about half of the cases and thus exponentiation made more efficient by weakening the non-adjacency property (cf. [2]). A *modified window NAF* is a B-representation obtained from a window NAF as follows: if the $w+2$ most significant digits (ignoring any leading zeros) have the form

$$1\underbrace{0\,0\ldots0}_{w \text{ zeros}}\overline{b},$$

then substitute

$$0\,1\underbrace{0\ldots0}_{w-1 \text{ zeros}}\beta$$

where $\beta = 2^w - b$. In the above example, we obtain that the modified (width-2) NAF for 3 is 11_2. However, the modified NAF for 7 is still $100\overline{1}$: in this case, length expansion cannot be avoided without increasing the number of non-zero digits.

5 Fractional Windows

In small devices, the choice of w for exponentiation using the sliding window or window NAF technique described in section 4 may be dictated by memory limitations. The exponentiation algorithms given in section 2 need storage for $1 + 2^{w-1}$ elements of G, and thus memory may be wasted: e.g., if sufficient storage is available for up four elements, only three elements can actually be used ($w = 2$).

In this section, we show how the efficiency of exponentiation can be improved by using *fractional windows,* a generalization of the sliding window and window NAF techniques. We describe this new encoding technique first for the case that negative digits are allowed *(signed fractional windows).* We then describe a simpler variant for the case that only non-negative digits are permissible *(unsigned fractional windows).*

5.1 Signed Fractional Windows

Let $w \geq 2$ be an integer and m an odd integer such that $1 \leq m \leq 2^w - 3$. The digit set for the *signed fractional window representation* with these parameters is

$$B = \{\pm 1, \pm 3, \ldots, \pm(2^w + m)\}.$$

Let the mapping $digit\colon \{0, 1, \ldots, 2^{w+2}\} \to B \cup \{0\}$ be defined as follows:

- If x is even, then $digit(x) = 0$;
- otherwise if $0 < x \leq 2^w + m$, then $digit(x) = x$;
- otherwise if $2^w + m < x < 3 \cdot 2^w - m$, then $digit(x) = x - 2^{w+1}$;
- otherwise we have $3 \cdot 2^w - m \leq x < 2^{w+2}$ and let $digit(x) = x - 2^{w+2}$.

Observe that if x is odd, then $x - digit(x) \in \{0, 2^{w+1}, 2^{w+2}\}$. The following algorithm encodes e into signed fractional window representation:

$d \leftarrow \mathrm{LSB}_{w+2}(e)$
$c \leftarrow \lfloor e/2^{w+2} \rfloor$
$i \leftarrow 0$
while $d \neq 0 \ \vee \ c \neq 0$ **do**
  $b \leftarrow digit(d)$
  $b_i \leftarrow b$; $i \leftarrow i + 1$
  $d \leftarrow d - b$
  $d \leftarrow \mathrm{LSB}(c) \cdot 2^{w+1} + d/2$
  $c \leftarrow \lfloor c/2 \rfloor$
return $b_{i-1}, \ldots, b_0$

This algorithm is a direct variant of the window NAF generation algorithm shown in section 4, but based on the new mapping *digit*. Here we have expressed the algorithm in a way that shows that the loop is essentially a finite state machine (with $2^{w+1} + 1$ states for storing, after b has been subtracted from the previous value of d, the even number d with $0 \leq d \leq 2^{w+2}$); new bits taken from c are considered input symbols and the generated digits b_i are considered output symbols.

The average density achieved by the signed fractional window representation with parameters w and m is

$$\frac{1}{w + \frac{m+1}{2^w} + 2}$$

for $e \to \infty$. (Assume that an endless sequence of random bits is the input to the finite state machine described above: whenever it outputs a non-zero digit, the intermediate value $d \bmod 2^{w+2}$ consists of $w + 1$ independent random bits plus the least significant bit, which is necessarily set. Thus with probability $p = \frac{1}{2} - \frac{m+1}{2^{w+1}}$, we have $d - digit(d) = 2^{w+1}$, which implies that the next non-zero output digit will follow after exactly w intermediate zeros; and with probability $1 - p$, we have $d - digit(d) \in \{0, 2^{w+2}\}$, which implies that the next non-zero output digit will follow after $w + 2$ intermediate zeros on average. Thus the total average for the number of intermediate zeros is $p \cdot w + (1-p) \cdot (w+2) = w + \frac{m+1}{2^w} + 1$, which yields the above expression for the density.) Comparing this with the $1/(w + 2)$ density for width-$(w + 1)$ NAFs, we see that the effective window size has been increased by $(m + 1)/2^w$, which is why we speak of "fractional windows".

As in section 4.1, length expansion can be avoided in many cases by modifying the representation. The *modified signed fractional window representation* is obtained as follows: if the $w+2$ most significant digits are of the form

$$1\underbrace{0\,0\ldots0}_{w \text{ zeros}}\overline{b},$$

then substitute

$$0\,1\underbrace{0\ldots0}_{w-1 \text{ zeros}}\beta$$

where $\beta = 2^w - b$; if the $w+3$ most significant digits are of the form

$$1\underbrace{0\,0\ldots0}_{w+1 \text{ zeros}}\overline{b}$$

with $b > 2^w$, then substitute

$$0\,1\underbrace{0\ldots0}_{w \text{ zeros}}\beta$$

where $\beta = 2^{w+1} - b$; and if the $w+3$ most significant digits are of the form

$$1\underbrace{0\,0\,0\ldots0}_{w+1 \text{ zeros}}\overline{b}$$

with $b < 2^w$, then substitute

$$0\,0\,3\underbrace{0\ldots0}_{w-1 \text{ zeros}}\beta$$

where $\beta = 2^w - b$.

Precomputation for left-to-right exponentiation can be done in one squaring and $2^{w-1} + (m-1)/2$ multiplications (see section 2.1), and the result stage for right-to-left exponentiation can be implemented in $2^{w-1} + (m-1)/2$ squarings and $2^w + m - 1$ multiplications (see section 2.2).

Table 1 shows expected performance figures for left-to-right exponentiation using the signed fractional window method in comparison with the usual window NAF method for 160-bit scalars; a typical application is elliptic curve cryptography. The signed fractional window method with $w = 2$, $m = 1$ achieves an evaluation stage speed-up of about 2.3 % compared with the window NAF method with $w = 2$, assuming that squarings take as much time as general multiplications. (When projective coordinates are used for representing points on elliptic curves, squarings are in fact usually faster, which will increase the relative speed-up.) Table 2 is for right-to-left exponentiation; it takes into account the optimizations to the right-to-left stage noted in section 2.2. The table shows that at this exponent bit length, for $w = 3$ fractional windows bring hardly any advantage for right-to-left exponentiation due to the relatively high computational cost of the result stage.

Table 1. Left-to-right exponentiation with window NAFs or signed fractional windows, $\ell = 160$

	$w = 2$		$w = 3$				$w = 4$
	wNAF	s. fract. $m = 1$	wNAF	s. fract. $m = 1$	s. fract. $m = 3$	s. fract. $m = 5$	wNAF
precomputation stage:							
table entries	2	3	4	5	6	7	8
squarings	1	1	1	1	1	1	1
multiplications	1	2	3	4	5	6	7
evaluation stage:							
squarings	≤ 160	≤ 160	≤ 160	≤ 160	≤ 160	≤ 160	≤ 160
multiplications	≈ 40.0	≈ 35.6	≈ 32.0	≈ 30.5	≈ 29.1	≈ 27.8	≈ 26.7

Table 2. Right-to-left exponentiation with window NAFs or signed fractional windows, $\ell = 160$

	$w = 2$		$w = 3$				$w = 4$
	wNAF	s. fract. $m = 1$	wNAF	s. fract. $m = 1$	s. fract. $m = 3$	s. fract. $m = 5$	wNAF
right-to-left stage:							
squarings	≤ 160	≤ 160	≤ 160	≤ 160	≤ 160	≤ 160	≤ 160
multiplications	≈ 39.0	≈ 33.6	≈ 29.0	≈ 26.5	≈ 24.1	≈ 21.8	≈ 19.7
result stage:							
input variables	2	3	4	5	6	7	8
squarings	1	2	3	4	5	6	7
multiplications	2	4	6	8	10	12	14

5.2 Unsigned Fractional Windows

The *unsigned fractional window representation* uses digit set

$$B = \{1, 3, \ldots, 2^w + m\}$$

and can be obtained by a variant of the technique from section 5.1. Here, let the mapping $digit\colon \{0, 1, \ldots, 2^{w+1}\} \to B \cup \{0\}$ be defined as follows:

- If x is even, then $digit(x) = 0$;
- otherwise if $0 < x \leq 2^w + m$, then $digit(x) = x$;
- otherwise let $digit(x) = x - 2^w$.

If x is odd, then $x - digit(x) \in \{0, 2^w\}$. The following algorithm encodes e into unsigned fractional window representation:

$d \leftarrow \mathrm{LSB}_{w+1}(e)$
$c \leftarrow \lfloor e/2^{w+1} \rfloor$
$i \leftarrow 0$
while $d \neq 0 \ \vee \ c \neq 0$ **do**

$b \leftarrow digit(d)$
$b_i \leftarrow b$; $i \leftarrow i + 1$
$d \leftarrow d - b$
$d \leftarrow \mathrm{LSB}(c) \cdot 2^w + d/2$
$c \leftarrow \lfloor c/2 \rfloor$
return $b_{i-1}, \ldots, b_0$

Similarly to the signed case, it can be seen that the average density of the unsigned fractional window representation is

$$\frac{1}{w + \frac{m+1}{2^w} + 1}$$

for $e \to \infty$. The precomputation or result stage is as before.

Table 3 shows expected performance figures for left-to-right exponentiation using the unsigned fractional window method in comparison with the usual sliding window method for 1024-bit scalars; a typical application is exponentiation in the multiplicative semigroup $(\mathbb{Z}/n\mathbb{Z})$ for an integer n. If squarings take as much time as general multiplications, the unsigned fractional window method with $w = 2$, $m = 1$ is approximately 3.7 % faster than the sliding window method with $w = 2$. Table 4 shows the figures for right-to-left exponentiation, taking into account the optimizations to the right-to-left stage noted in section 2.2.

5.3 Example: Application to Multi-Exponentiation

Assume we have to compute a power product $g_1^{e_1} g_2^{e_2}$ with random ℓ-bit exponents e_1, e_2 in a group where inversion is easy, and that we have storage for five precomputed elements. For using interleaved exponentiation as described in section 3, we can represent e_1 as a width-3 NAF and e_2 in signed fractional window representation with $w = 2$, $m = 1$. This means we use precomputed elements $g_1, g_1^3, g_2, g_2^3, g_2^5$. The evaluation stage needs at most ℓ squarings and approximately $\left(\frac{1}{4} + \frac{1}{4+1/2}\right)\ell = \frac{17}{36}\ell$ multiplications on average, compared with

Table 3. Left-to-right exponentiation with sliding windows or unsigned fractional windows, $\ell = 1023$

	$w = 2$		$w = 3$				$w = 4$
	slid. w.	u. fract. $m = 1$	slid. w.	u. fract. $m = 1$	u. fract. $m = 3$	u. fract. $m = 5$	slid. w.
precomputation stage:							
table entries	2	3	4	5	6	7	8
squarings	1	1	1	1	1	1	1
multiplications	1	2	3	4	5	6	7
evaluation stage:							
squarings	≤ 1023	≤ 1023	≤ 1023	≤ 1023	≤ 1023	≤ 1023	≤ 1023
multiplications	≈ 341.0	≈ 292.3	≈ 255.8	≈ 240.7	≈ 227.3	≈ 215.4	≈ 204.6

Table 4. Right-to-left exponentiation with sliding windows or unsigned fractional windows, $\ell = 1023$

	$w = 2$		$w = 3$				$w = 4$
	slid. w.	u. fract. $m = 1$	slid. w.	u. fract. $m = 1$	u. fract. $m = 3$	u. fract. $m = 5$	slid. w.
right-to-left stage:							
squarings	≤ 1023	≤ 1023	≤ 1023	≤ 1023	≤ 1023	≤ 1023	≤ 1023
multiplications	≈ 340.0	≈ 290.3	≈ 252.8	≈ 236.7	≈ 222.3	≈ 209.4	≈ 197.6
result stage:							
input variables	2	3	4	5	6	7	8
squarings	1	2	3	4	5	6	7
multiplications	2	4	6	8	10	12	14

$\frac{1}{2}\ell$ multiplications for interleaved exponentiation with width-3 NAFs for both exponents (precomputed elements g_1, g_1^3, g_2, g_2^3).

(A similar scenario is considered in [14], using a different multi-exponentiation algorithm; for groups where inversion is easy, that technique using the same amount of storage as needed in our above example runs slightly slower according to the heuristical results in [14, table 12].)

6 Window NAF Splitting

One approach for efficient exponentiation with precomputation for fixed bases, given an upper bound $\ell+1$ for exponent bit lengths and a positive integer parameter v, is to turn exponentiations into multi-exponentiations by using *exponent splitting* as follows [12]:

$$g^e = \prod_{0 \leq i < \lceil (\ell+1)/v \rceil} (g^{2^{iv}})^{e[iv+v-1 \ldots iv]}$$

Here $e[j \ldots j']$ denotes the integer whose binary representation is the concatenation of bits j down to j' of e (i.e. $\lfloor e/2^{j'} \rfloor \bmod 2^{j-j'+1}$).

For groups where inversion is easy, [10] proposes to use this approach with window NAF based interleaved exponentiation: that is, each of the length-v exponent parts is encoded as a window NAF as described in section 4, and then an interleaved exponentiation using these windows NAFs is performed as described in section 3. With width-$(w+1)$ NAFs, this computation should take about v squarings and $\ell/(w+2)$ multiplications using $\lceil (\ell+1)/v \rceil \cdot 2^{w-1}$ precomputed elements. However, if v is very small, the expected number of multiplications will be noticeably higher because the estimate that the density of window NAFs is approximately $1/(w+2)$ becomes accurate only if the encoded number is sufficiently long. (Window NAFs usually waste part of one window; the more individual integers must be encoded into window NAFs, the more is wasted in total.)

An improved technique that avoids this drawback is *window NAF splitting.* Instead of splitting the binary representation of exponent e into partial exponents of length v and determining window NAFs for these, we first determine the window NAF of e and then split this new representation into parts of length v. The computation continues as above, using the interleaved exponentiation algorithm shown in section 3. To avoid length expansion if possible, this technique should be used with modified window NAFs (section 4.1) The leftmost part can be made large than the others if one more part would have to be added otherwise; e.g. for integers up to 160 bits with $v = 8$:

$$\underbrace{\boxed{b_{160}\, b_{159} \cdots b_{152}}}_{9 \text{ digits}} \underbrace{\boxed{b_{151} \cdots b_{144}}}_{8 \text{ digits}} \cdots \underbrace{\boxed{b_7 \cdots b_0}}_{8 \text{ digits}}$$

Most of the time, the additional digit of the leftmost part will be zero since length expansion is relatively rare (for modified window NAFs of positive integers up to a length of ℓ bits with $w = 4$, only about one out of five cases has a non-zero digit at maximum index ℓ).

With window NAF splitting, exponentiations for ℓ-bit exponents can be performed in $v - 1$ squarings and on average about $\ell/(w + 2)$ multiplications, using $\lceil(\ell + 1)/v\rceil \cdot 2^{w-1}$ precomputed elements. If the leftmost part gets an extra digit as described above, $\lceil\ell/v\rceil \cdot 2^{w-1}$ precomputed elements are sufficient, and the number of squarings goes up to v for some cases.

This method can compete with Lim and Lee's algorithm for exponentiation with precomputation described in [8] and [7] even when much space is available for precomputed elements (whereas exponent splitting with window NAF based interleaving exponentiation is better than the Lim-Lee algorithm only for comparatively small precomputed tables). For example, if $\ell = 160$, then with $v = 8$ and $w = 4$ (160 precomputed elements if we allow an extra digit in the leftmost window NAF part), our exponentiation method with window NAF splitting needs about 7.2 squarings and 26.7 multiplications. The Lim-Lee algorithm can perform such 160-bit exponentiations in 13 squarings and about 26.6 multiplications using 128 precomputed elements, or in 11 squarings and about 22.8 multiplications using 256 precomputed elements.

It is possible to use window NAF splitting with a *flexible window size:* While generating digits using the algorithm described in section 4, parameter w can be changed. This should be done only at the beginning of a new part of the window NAF (i.e., when the number of digits generated so far is a multiple of v). For example, if in the $\ell = 160$ setting we are using $v = 8$ and allowing an extra digit in the leftmost part, the (modified) window NAF will be split into 20 parts; we can start with $w = 5$ for the first 12 of these, then switch to $w = 4$ for the remaining 8. Then we need $12 \cdot 2^4 + 8 \cdot 2^3 = 256$ precomputed elements and can perform exponentiations in about 7.2 squarings and $\frac{12 \cdot 8}{5+2} + \frac{8 \cdot 8}{4+2} \approx 24.4$ multiplications, which is usually (depending on the relative performance of squarings and general multiplications) better than the performance of the Lim-Lee algorithm with 256 precomputed elements.

7 Compact Encodings

When storing a window NAF or fractional window representation where a single digit may take $w+1$ bits of memory (this is the case for width-$(w+1)$ NAFs if we take into account that the digit may be zero, and it is the case for signed fractional window representations), then it is not necessary to store digits separately in $w+1$ bits each. If memory is scarce, it is possible to exploit the properties of the representation to obtain a more compact encoding into bit strings (cf. [5]).

We can encode a zero digit as a single zero bit, and a non-zero digit as a one bit followed by a representation of the respective digit, which together takes $w+1$ bits in the case of window NAFs and $w+2$ bits in the case of signed fractional window representations. After each non-zero digit, there will be w zero digits (unless conversion into a modified window NAF has taken place), and these can be omitted from the encoding. Thus, compared with the usual binary representation of the number, in the case of window NAFs we only have growth by a small constant; in the case of signed fractional window representations (and similarly in the case of unsigned fractional window representations), we additionally have growth by one bit for each non-zero digit of the representation.

This bit string encoding can easily be adapted to the case that the bit string will be read in the reverse of the direction in which it was written (for example, non-zero digits should be encoded as a representation of the respective digit followed by a one bit rather than the other way around).

8 Conclusions

We have closed a gap in the sliding window and window NAF methods for efficient exponentiation: our fractional window techniques can improve the performance by a couple of percents in devices with limited memory by making use of memory that would have to remain unused with the previously known methods.

With window NAF splitting, we have shown an efficient technique for exponentiation with precomputation in groups where inversion is easy, which provides a convenient alternative to the patented Lim-Lee method.

References

1. American National Standards Institute (ANSI). Public key cryptography for the financial services industry: The elliptic curve digital signature algorithm (ECDSA). ANSI X9.62, 1998.
2. Bosma, W. Signed bits and fast exponentiation. Department of Mathematics, University of Nijmegen, Report No. 9935, 1999.
3. Cohen, H., Ono, T., and Miyaji, A. 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.

4. Gordon, D. M. A survey of fast exponentiation methods. *Journal of Algorithms 27* (1998), 129–146.
5. Joye, M., and Tymen, C. Compact encoding of non-adjacent forms with applications to elliptic curve cryptography. In *Public Key Cryptography – PKC 2001* (2001), K. Kim, Ed., vol. 1992 of *Lecture Notes in Computer Science*, pp. 353–364.
6. Knuth, D. E. *The Art of Computer Programming – Vol. 2: Seminumerical Algorithms (3rd ed.).* Addison-Wesley, 1998.
7. Lee, P.-j., and Lim, C.-h. Method for exponentiation in a public-key cryptosystem. United States Patent 5,999,627, 1999.
8. Lim, C. H., and Lee, P. J. More flexible exponentiation with precomputation. In *Advances in Cryptology – CRYPTO '94* (1994), Y. G. Desmedt, Ed., vol. 839 of *Lecture Notes in Computer Science*, pp. 95–107.
9. Miyaji, A., Ono, T., and Cohen, H. Efficient elliptic curve exponentiation. In *International Conference on Information and Communications Security – ICICS '97* (1997), Y. Han, T. Okamoto, and S. Qing, Eds., vol. 1334 of *Lecture Notes in Computer Science*, pp. 282–290.
10. Möller, B. Algorithms for multi-exponentiation. In *Selected Areas in Cryptography – SAC 2001* (2001), S. Vaudenay and A. M. Youssef, Eds., vol. 2259 of *Lecture Notes in Computer Science*, pp. 165–180.
11. National Institute of Standards and Technology (NIST). Digital Signature Standard (DSS). FIPS PUB 186-2, 2000.
12. Pippenger, N. On the evaluation of powers and related problems (preliminary version). In *17th Annual Symposium on Foundations of Computer Science* (1976), IEEE Computer Society, pp. 258–263.
13. Reitwiesner, G. W. Binary arithmetic. *Advances in Computers 1* (1960), 231–308.
14. Sakai, Y., and Sakurai, K. Algorithms for efficient simultaneous elliptic scalar multiplication with reduced joint Hamming weight representation of scalars. In *Information Security – ISC 2002* (2002), A. H. Chan and V. Gligor, Eds., vol. 2433 of *Lecture Notes in Computer Science*, pp. 484–499.
15. Schroeppel, R., Orman, H., O'Malley, S., and Spatscheck, O. Fast key exchange with elliptic curve systems. In *Advances in Cryptology – CRYPTO '95* (1995), D. Coppersmith, Ed., vol. 963 of *Lecture Notes in Computer Science*, pp. 43–56.
16. Solinas, J. A. An improved algorithm for arithmetic on a family of elliptic curves. In *Advances in Cryptology – CRYPTO '97* (1997), B. S. Kaliski, Jr., Ed., vol. 1294 of *Lecture Notes in Computer Science*, pp. 357–371.
17. Solinas, J. A. Efficient arithmetic on Koblitz curves. *Designs, Codes and Cryptography 19* (2000), 195–249.
18. Thurber, E. G. On addition chains $l(mn) \leq l(n) - b$ and lower bounds for $c(r)$. *Duke Mathematical Journal 40* (1973), 907–913.
19. Yao, A. C.-C. On the evaluation of powers. *SIAM Journal on Computing 5* (1976), 100–103.