Exponentiation Using a Large-Digit Representation and ECC Applications

Nicolas Méloni and M. Anwar Hasan
Department of Electrical and Computer Engineering
University of Waterloo

Abstract

Fast computation of exponentiation is a key issue for group based public key cryptosystems. In fact, it has drawn considerable attention of researchers for many years [1]. Serious break-throughs were made in the 30's by Brauer [5] and then by Yao [20] during the 70's, with improvements by Thurber, Knuth and more recently Möller [19, 11, 17]. All those methods are based on a somewhat similar approach: first recode the exponent using a certain range of digits, precompute the powers corresponding to the digit set and then use them in a square-and-multiply algorithm. In this work, we propose to merge the precomputation stage with the exponentiation itself with gain thanks to a large-digit representation.

Keywords: exponentiation, addition chains, non adjacent form, sliding windows, integer representation, elliptic curve, point scalar multiplication.

1 Introduction

Exponentiation consists of computing $x^n = x \times \cdots \times x$, where x is an element of a group G and n is an integer. It is one of the most common arithmetic operations. It is also the key operation in many group based cryptographic protocols, such as RSA [18] and ECC [12, 16], where group operations are modular multiplication and elliptic curve point addition, respectively. Speeding up this operation is crucial for the efficiency of those protocols. To do so, several algorithms have been proposed. They typically consist of recoding the exponent n in order to minimize the number of group operations. However, minimizing the number of operations does not always yield the fastest exponentiation algorithm for a given group. The relative costs of multiplications and squarings, coupled with the possibility of faster combined operation (square and multiply, tripling, quadrupling etc), can lead to costlier (in terms of number of group multiplications) but faster algorithms using specific representation such as the Zeckendorf representation [8], Double Base Number System [9], binary/ternary representation [6] etc.

In this work, we will focus on a general recoding algorithm, which means that we will try to reduce the total number of group multiplications without any consideration of other specific operations. In this generic case, the best technique is to use the Brauer algorithm [5] combined with different improvements due to Thurber [19], Knuth [11] and Möller [17], leading to the so called 2^k-ary and NAF_w (width w Non Adjacent Form) methods, with or without fractional windows. They are based on a similar approach. First, one computes the powers corresponding to the digits of the used representation and then one performs a square-and-multiply scheme. In this work, our contribution is to propose to merge the precomputation stage with the actual exponentiation thanks to the use of large digits. More precisely, the main idea is to use the elements computed during the first steps of the exponentiation as precomputed elements. Then, we propose two major improvements to this basic idea (the use of even digits and the choice of starting addition chain) in order to make it more efficient.

This article is organized as follow: We first briefly recall Brauer's algorithm and generic exponentiation techniques. Then we present the basic idea behind our large digits representation and several improvements to make it efficient. We present some comparisons with state of the art of different bit length exponents. Finally, we propose an efficient implementation of our work on elliptic curves in short Weierstraßform and give a comprehensive comparison with the most recent methods.

2 Exponentiation techniques

Let $n = (n_{t-1} \dots n_0)_2$ be an integer. It is always possible to compute x^n using less than $2\log(n) = 2t$ group multiplications, for which one can simply use the classical *square-and-multiply* algorithm. Let $y = x^{n_{t-1}\cdots n_{t-i}}$, if n_{t-i-1} is 0 then $y \leftarrow y \times y$ and if n_{t-i-1} is one $y \leftarrow (y \times y) \times x$. One just has to look at the binary representation of n to perform the exponentiation. Brauer's algorithm and its different improved versions generalize this approach. Instead of looking at the binary representation of n, one considers its 2^k-ary representation. As an example if $n = 314159 = (1001100101100101111)_2$ and $k = 4$ then

$$\begin{aligned} n &= (\underbrace{100}_{4}\,\underbrace{1100}_{12}\,\underbrace{1011}_{11}\,\underbrace{0010}_{2}\,\underbrace{1111}_{15})_2 \\ &= 4 \times 2^{16} + 12 \times 2^{12} + 11 \times 2^8 + 2 \times 2^4 + 15, \end{aligned}$$

and we can recode n as $(4\ 000\,12\ 000\,11\ 0002\ 000\,15)_2$. To perform an exponentiation using this representation, first one has to precompute and store $x, x^2, x^3, \dots, x^{14}, x^{15}$ and then perform a *square-and-multiply* like algorithm. Several improvements can reduce the total number of multiplications. Three commonly used improvements are as follows:

First, one does not have to use even digits. We can easily see that the operation $y \leftarrow y^2 \times x^{14}$ can be replaced by $y \leftarrow (y \times x^7)^2$.

Second, one can use *sliding windows*, where we allow k to vary, e.g. we do not take into account 0 digits between two non zero digits. As an example, considering the integer $(1110000011)_2$, we can consider it as $7 \times 2^7 + 3 = (70000003)_2$ instead of $3 \times 2^8 + 8 \times 2^4 + 3 = (300080003)_2$. In this case, each window begins with a 1, ends with a 1 and has a length of at most k bits.

Third, one can use negative digits, if the element inversion operation is easy in the considered group.

Algorithm 1 summarizes these improvements. Deleting line 5 allows us to obtain the unsigned version of Brauer's improved algorithm.

Algorithm 1 Computes the NAF_w representation of integer n

Input: An integer n and a window size w
Output: $n = (n_t n_{t-1} \dots n_1 n_0)_2$

```
i = 0
while n > 0 do
    if n mod 2 ≠ 0 then
        n_i = n mod 2^w
        if n_i > 2^{w-1} then n_i = n_i − 2^w
        end if
    else
        n_i = 0
    end if
    n = (n − n_i)/2
    i = i + 1
end while
return (n_{i-1} ... n_0)
```

3 Large Digit Representation

The NAF_w method requires, on average, only $\frac{1}{w+1}$ multiplications per exponent bit. However, at the same time it reduces the number of squarings by at most $(w-2)$. Moreover, using width $w+1$ instead of w requires us to double the number of precomputed powers, whereas the addition saving in the number of squarings is 1.

The method we propose here is an attempt to merge precomputations and actual exponentiation, and also, to take advantage of sufficiently large amount of storage available on certain platforms, such as personal computers. The main idea is to store the first computed powers corresponding to the most significant bits of n and use those powers as precomputed powers in a square-and-multiply scheme.

3.1 A simple example

Let n be equal to 314159. Its different NAF_w representations are:

$w = 2:$ 101 $0\,\overline{1}\,01$ $0\,\overline{1}\,0\,\overline{1}$ $010\,\overline{1}$ $000\,\overline{1}$
$w = 3:$ 100 0300 1003 0003 $000\,\overline{1}$
$w = 4:$ 5 $000\,\overline{3}$ $000\,\overline{5}$ 0003 $000\,\overline{1}$

The exponentiation is made of two stages: precomputations and then actual exponentiation. The precomputation stage consists of determining the digits of the representation. However, it is also possible to represent n as:

$n =$ 19 00 0110 0 19 01 0 19 0 19
$=$ 19 00 1010 000 19 $000\,\overline{1}$

In this case, we have first determined the integer corresponding to the five most significant bits of n (i.e. $19 = (10011)_2$) using a traditional double-and-add algorithm, and then used the intermediate integers as digits. This finally leads to the following addition chains:

$\underline{1}, 2, 4, 8, 9, 18, \underline{19}$, 38, 76, 152, 304,. . . , 157070, 314140, 314159 and

$\underline{1}, 2, 4, 8, 9, 18, \underline{19}$, 38, 76, 152, 153,. . . , 157080, 314160, 314159

There is no precomputation stage; we just reuse the underlined integers as digits. The purpose of the following subsection is to present an algorithm for computing the large-digit representation using the first intermediate integers as a set of digits.

3.2 Creating dictionary

When using a 2^k-ary method, one looks at the k rightmost bits of n and subtracts the integer corresponding to these k bits. This assumes that one has first precomputed a dictionary $(1, 3, \ldots, 2^k - 1)$. For example, if $k = 4$ and $n = 109 = (110\underline{1101})_2$, one will use the precomputed integer 13=$(1101)_2$, rewriting n as $(1\ 1\ 0\ 0\ 0\ 0\ 13)_2$. However, one can use 29=$(11101)_2$ instead of 13 and rewrite 109 as $(1\ 0\ 1\ 0\ 0\ 0\ 29)_2$. In fact, one can use any integer whose 4 rightmost bits are $(1101)_2$. Based on this remark, we construct a dictionary $\mathcal{D}_S^w$ based on a set of l integers $S = \{s_1, \ldots, s_l\}$ and a width w. Our goal is to find representatives for the digits $\{1, 2, 3, 4, \ldots, 2^w - 2, 2^w - 1\}$ in the set S. For each integer of this set, we look at its least significant j bits, for $j = 1, \ldots, w$. This results into w integers, which are not all necessarily distinct. If they correspond to digits that do not already have any representative, then the current integer, say s_i, becomes a representative for those digits. In the end, the rth element of the dictionary is the representative s_i of digit r. The creation of the dictionary is formally described in Algorithm 2.

Algorithm 2 Creating dictionary $\mathcal{D}_S^w$

Input: A set $S = \{s_1, \ldots, s_l\}$ of integers and a width w
Output: $\mathcal{D}_S^w = (d_1, \ldots, d_{2^w-1})$

```
D_S^w = (0, ..., 0)
for i = 1 to l do
    for j = 1 to w do
        r = s_i mod 2^j
        if d_r = 0 then d_r = s_i
        end if
    end for
end for
return (d_1, ..., d_{2^w-1})
```

Example 3.1 *Using the set of integers $S = \{1, 2, 4, 8, 9, 18, 19\}$ and a width equal to 4, when $i = 6$ in Algorithm 2, the dictionary is equal to $(1, 2, 0, 4, 0, 0, 0, 8, 9, 0, 0, 0, 0, 0, 0)$. Next, we consider the 7th element of S, that is to say $19 = (10011)_2$ and look at its residues modulo 2^j:*

- $19 \mod 2 = 1$ *and* $d_1 = 1 \neq 0$,
- $19 \mod 2^2 = 3$, *and* $d_3 = 0$ *so* $d_3 \leftarrow 19$
- $19 \mod 2^3 = 3$ *and* $d_3 = 19 \neq 0$,
- $19 \mod 2^4 = 3$ *and* $d_3 = 19 \neq 0$.

So now, 19 will be used as the third element of the dictionary, and the final dictionary is $\mathcal{D}_S^w = (1, 2, 19, 4, 0, 0, 0, 8, 9, 0, 0, 0, 0, 0, 0)$.

Remark 3.2 *The order of the elements of the set S directly influence the result of the algorithm (which means that it should be considered more like a list than a set). From the previous example, if we exchange 9 and 1, then 9 will be the representative of 1 instead of 1. As we prefer to have the smallest representative as possible, from now on we assume that the set S is sorted. However, it suffices to modify the condition: "If $d_r = 0$" by "If $d_r = 0$ or $s_i < d_r$" to always obtain the expected result.*

Algorithm 3 Computing a Large-Digit Representation using dictionary $\mathcal{D}_S^w$

Input: An integer n, a window width w and a dictionary $D_S^w = (d_1, \ldots, d_{2^w-1})$
Output: $n = (n_t n_{t-1} \ldots n_1 n_0)_2$ such that $n = \sum_{i=0}^{t} n_i 2^i$

```
i = 0
while n > 0 do
    if n mod 2 ≠ 0 then
        for j = 0 to w − 1 do
            r = n mod 2^{w−j}
            if d_r ≠ 0 and d_r ≤ n then n_i = d_r; break
            end if
        end for
    else
        n_i = 0
    end if
    n = (n − n_i)/2
    i = i + 1
end while
return n_{i−1} ... n_1 n_0
```

3.3 Finding a Large-Digit Representation

Now that we are able to build a dictionary, it is easy to find a Large-Digit Representation (LDR) of an integer n based on D_S^w.

The main difference between this algorithm and Algorithm 1 is the `for` loop. Indeed, in our case, every digit does not necessary has a representative in set S. So we first look in the dictionary for a representative for $n \mod 2^w$. If there is none we reduce the window width (i.e. w) and try again.

Example 3.3 *Using the dictionary* $(1, 2, 19, 4, 0, 0, 0, 8, 9, 0, 0, 0, 0, 0, 0)$*, Algorithm 3 finds a width 4 large-digit representation of* $n = 2863$ *as follows:*

- $n \mod 2^4 = 15$ *and* $d_{15} = 0$*,*
- $n \mod 2^3 = 7$ *and* $d_7 = 0$*,*
- $n \mod 2^2 = 3$ *and* $d_3 = 19 \Rightarrow n_0 = 19$ *and* $n = (n - 19)/2 = 1422$*,*
- $n \mod 2 = 0 \Rightarrow n_1 = 0$ *and* $n = n/2 = 711$
- ...
- *and finally* $n = (1100\,19\,010\,19\,0\,19)_2$

3.4 Performing an exponentiation using LDR

Let G be a group and x an element of G. One can compute x^{314159} in the following way:

- first decompose $314159 = (1001100101100101111)_2$ as $(10011)_2 \times 2^{14} + (101100101111)_2 = 19 * 2^{14} + 2863$

- compute x^{19} using the binary chain $(1, 2, 4, 8, 9, 18, 19)$ and store the values $(x, x^2, x^4, x^8, \ldots, x^{19})$
- $2863 = (1100\,19\,010\,19\,0\,19)_2$
- $x^{38} = (x^{19})^2 \rightarrow x^{76} = (x^{38})^2 \rightarrow x^{152} = (x^{76})^2 \rightarrow x^{305} = (x^{152})^2 \times x \rightarrow \ldots$
- $\cdots \rightarrow x^{157070} = (x^{8535})^2 \rightarrow x^{314159} = (x^{157070})^2 \times x^{19}$.

The complete exponentiation involves 18 squarings and 8 multiplications. Using the 2^k-ary method requires also 18 squarings and 8 multiplications if k is chosen carefully ($k = 3$). In this case, 314159 can be represented as $(1000300100300005007)_2$ and one first has to precompute x, x^2, x^3, x^5, x^7 to perform the exponentiation.

4 Improvements

From the previous section, we can see that despite the fact that we use a larger number of precomputed digits, we do not decrease the overall number of multiplications. This is because of two reasons: we do not use even numbers in our integer set and the addition chain used to compute the leftmost bits of n (19 in our example) does not provide a *good* set of integers.

Keeping these remarks in mind, below we discuss two improvements to our new LDR and how to use negative digits.

4.1 Using even numbers

Let us suppose that our integer set S is equal to $\{1, 3, 14\}$ and we want to perform a step of Algorithm 3 to the integer $n = (101110)_2$ using a width of 4. Algorithm 3 will first divide n by 2, then look for an integer for digit 7, fail and find 3 as an integer corresponding to digit 3. The final representation is then $(101030)_2$. However, it is easy to see that 14 could have been used instead of 3, saving one addition in the end ($n = (1\ 0\ 0\ 0\ 0\ 14)_2$).

To take into account even digits in our representation we can modify the way we compute the dictionary and then adapt the algorithm. The main idea is to consider that 14=$(1110)_2$ can be used as a digit for 7 as long as the preceding bit is a zero. The modified algorithm is given below where the output is a two-row dictionary. The first row is the same as in Algorithm 2 and the second row indicates how many times an nonzero entry in the first row can be divided by two.

Algorithm 4 Creating a two-row dictionary $\mathcal{D}_S^w$

Input: A set $S = \{s_1, \ldots, s_l\}$ of integers and a width w

Output: $\mathcal{D}_S^w = \begin{pmatrix} d_{1,1} & \ldots & d_{1,2^w-1} \\ d_{2,1} & \ldots & d_{2,2^w-1} \end{pmatrix}$

$\mathcal{D}_S^w = \begin{pmatrix} 0, \ldots, 0 \\ 0, \ldots, 0 \end{pmatrix}$
for $i = 1$ to l **do**
  $div2 = 0; s_i' = s_i$
  while $s_i' \mod 2 = 0$ **do**
    $s_i' = s_i'/2; div2 = div2 + 1$
  end while
  for $j = 1$ to w **do**
    $r = s_i' \mod 2^j$
    if $d_{1,r} = 0$ **then** $d_{1,r} = s_i$ and $d_{2,r} = div2$
    end if
  end for
end for
return $\begin{pmatrix} d_{1,1} & \ldots & d_{1,2^w-1} \\ d_{2,1} & \ldots & d_{2,2^w-1} \end{pmatrix}$

The main difference between Algorithms 2 and 4 is that, for each element s of S, we compute s' and $div2$ so that $s = s' \times 2^{div2}$ with s' odd. Then we apply Algorithm 2 to s' and store both s' and $div2$.

Example 4.1 *Let* $S = \{1, 3, 6, 14\}$ *and* $w = 3$*:*

i=1: $s_1 = 1 \times 2^0$ *so* $div2 = 0$*,*

- $s_1 = 1 \mod 2^j, \forall j \leq 3$ *so* $d_{1,1} = s_1 = 1$ *and* $d_{2,1} = div2 = 0$

i=2: $s_2 = 3 \times 2^0$ *so* $div2 = 0$*,*

- $s_2 = 1 \mod 2$ *and* $d_{1,1} \neq 0$ *so* 3 *is not used as a digit for 1*
- $s_2 = 3 \mod 2^2$ *(and* 2^3*),* $d_{1,3} = 0$ *so* $d_{1,3} = 3$ *and* $d_{2,3} = 0$

i=3: $s_3 = 3 \times 2^1$ *so* $div2 = 1$ *and* $s_3' = 3$*,*

- $s_3' = 1 \mod 2$ *and* $d_{1,1} \neq 0$
- $s_3' = 3 \mod 2^2$ *(and* 2^3*) and* $d_{1,3} \neq 0$

i=4: $s_4 = 7 \times 2^1$ *so* $div2 = 1$ *and* $s_4' = 7$*,*

- $s_4' = 1 \mod 2$ *and* $d_{1,1} \neq 0$
- $s_4' = 3 \mod 2^2$ *and* $d_{1,3} \neq 0$
- $s_4' = 7 \mod 2^3$ *and* $d_{1,7} = 0$ *so* $d_{1,7} = s_4 = 14$ *and* $d_{2,7} = 1$

The final two-row dictionary is: $\begin{pmatrix} 1 & 0 & 3 & 0 & 0 & 0 & 14 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$

Now we can rewrite Algorithm 3 to take into account the new dictionary:

Algorithm 5 Computing a Large-Digit Representation using a two-row dictionary $\mathcal{D}_S^w$

Input: An integer n, a window width w and a two-row dictionary D_S^w
Output: $n = (n_t n_{t-1} \dots n_1 n_0)$ such that $n = \sum_{i=0}^{t} n_i 2^i$

```
i = 0
Zeros = 0
while n > 0 do
    if n mod 2 ≠ 0 then
        for j = 0 to w − 1 do
            r = n mod 2^{w−j}
            if d_{1,r} ≠ 0 and d_{1,r} ≤ n and d_{2,r} ≤ Zeros then
                i = i − d_{2,r}
                n_i = d_{1,r}
                n = n × 2^{d_{2,r}}
                Zeros = 0
                break
            end if
        end for
    else
        n_i = 0
        Zeros = Zeros + 1
    end if
    n = (n − n_i)/2
    i = i + 1
end while
return n_{i−1} ... n_1 n_0
```

In this algorithm, the $Zeros$ variable represents the number of consecutive 0's between two nonzero digits. Each time a nonzero digit is found, we first look in the dictionary if there is a representative for this digit and then verify if the number of 0's preceding this digit is sufficient. As an example, we must have computed a least one 0 before using 14 as a digit for 7.

Remark 4.2 *It is better not to allocate memory for data that are always equal to 0. When using Algorithm 4, it is obvious that $d_{1,r}$ and $d_{2,r}$ will always be equal to 0 as long as r is even. Taking this into account, one should not allocate memory for those values of r. As an example, the dictionary in Example 4.1 can be* $\begin{pmatrix} 1 & 3 & 0 & 14 \\ 0 & 0 & 0 & 1 \end{pmatrix}$ *instead of* $\begin{pmatrix} 1 & 0 & 3 & 0 & 0 & 0 & 14 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$.

We take this remark into account in the rest of this work, but in order to make the algorithms easier to follow, we do not change the indices of the dictionary (14 will still be equal to $d_{1,7}$ instead of $d_{1,4}$).

4.2 Using better addition chains

The set $\{1, 2, 4, 8, 9, 18, 19\}$ is not a good choice for our recoding method because most of its elements are not very useful. More precisely, even if we allow the use of even numbers, they still remain unused because they are obtained as the double of a previous integer, which means that they have almost the same binary representation.

As an example, 2= $(10)_2$, 4=$(100)_2$ and 8=$(1000)_2$ are not useful because they are obtained by doubling 1 three times. Thus, using the double-and-add algorithm to compute the topmost bits of a scalar n will always result in a waste of at least half of the computed integers. To solve this problem, we propose to compute the leftmost bits of n using Euclidean addition chains. They are only made of additions, so that the resulting integer set provides more different digits.

Definition 4.3 *An Euclidean addition chain computing an integer n is an addition chain which satisfies $v_1 = 1, v_2 = 2, v_3 = v_2+v_1$ and $\forall\, 3 \leq i \leq s-1$, if $v_i = v_{i-1}+v_j$ for some $j < i-1$, then $v_{i+1} = v_i+v_{i-1}$ or $v_{i+1} = v_i + v_j$.*

For example, $(1, 2, 3, 5, 7, 12, 19)$ is an Euclidean addition chain computing 19. Finding such chains is quite simple: it suffices to choose an integer g coprime with k and apply the subtractive form of Euclid's algorithm. However, finding short chains can be very difficult, depending on the bit length of the computed integer. Basically, to limit the length of Euclidean chains, the best way is to try different g close to $\frac{k}{\phi}$, where $\phi = \frac{1+\sqrt{5}}{2}$ is the golden ratio. For small values of k (less than 2^{21}, we have performed an exhaustive study of chain's length and have found that, on average, 20 tries are sufficient to find small chains. As an example, for each integer between 2^{20} and $2^{21}-1$, trying to compute chains using values of g in $\{\frac{k}{\phi}, \ldots, \frac{k}{\phi}+19\}$ leads to chains of length 34, on average.

Bigger integers will require a greater number of tries, but as we will only deal with relatively small integers (less than 40 bits), we neglect the additional cost.

If we use the preceding set of integers to compute our dictionary we obtain:

$$\begin{pmatrix} 1 & 3 & 5 & 7 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

whereas the use of the original set $\{1, 2, 4, 8, 9, 18, 19\}$ leads to the following dictionary:

$$\begin{pmatrix} 1 & 19 & 0 & 0 & 9 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

We clearly see in this example that the Euclidean addition chain has provided a *better* set of integers, whereas it has required just as much computations.

We can now perform an exponentiation on a group element x using exponent n and width w:

- decompose $n = (n_{t-1} \ldots n_0)_2$ as
  $n_H * 2^{t-l-1} + n_L$ where $n_H = (n_{t-1} \ldots n_{t-l-1})_2$ and $n_L = (n_{t-l-2} \ldots n_0)_2$
- find a short Euclidean addition chain $S = (s_1, \ldots s_l)$ computing n_H
- compute the dictionary $\mathcal{D}_S^w$ with Algorithm 4
- compute the large digit representation of n_L using Algorithm 5
- compute x^{n_H} using chain S and store the values $(x, x^{i_1}, x^{i_2}, \ldots, x^{i_m})$ used to represent n_L
- use LDR of n_L to perform an exponentiation starting from x^{n_H}

Remark 4.4 *In some groups, the inversion operation (i.e. computing x^{-1}) can be done easily. It is the case, for example, with the group of the points of an elliptic curve. In this case, the use of negative digits allows us to reduce the number of nonzero digits in any representation. Combined with the 2^k-ary method, it leads to the so called NAF_w representation. Such an improvement is not specific to our new representation and it does not change the general idea it is based on. Hence, we propose the signed version of Algorithm 5 in Appendix A.*

5 Comparisons

In this section we give some practical results about the complexities of our new representation, in both signed and unsigned versions, and compare these representations to the 2^k-ary method and its signed version. For different bit-length exponents, we summarize the number of group operations (squarings and multiplications), the number of stored elements, the optimal width and the optimal size for n_H.

Exp. size	Method	storage	width	n_H size	# group operations
256	2^k-ary	7	4	-	$254s + 58m = 312$
	LDR	15	7	20	$238s + 67m = 305$
	NAF_w	7	5	-	$254s + 49m = 303$
	signed LDR	14	8	16	$238s + 62m = 300$
512	2^k-ary	15	5	-	$510s + 100m = 610$
	LDR	24	11	24	$490s + 111m = 601$
	NAF_w	15	6	-	$510s + 87m = 597$
	signed LDR	20	9	20	$490s + 103m = 593$
1024	2^k-ary	31	6	-	$1021s + 177m = 1198$
	LDR	40	10	40	$994s + 192m = 1186$
	NAF_w	31	7	-	$1021s + 159m = 1180$
	signed LDR	32	11	28	$994s + 179m = 1173$

Table 1. Comparison between the LDR and the 2^k-ary representations for different exponent sizes

We can see that our Large Digit Representation, be it in its signed version or not, always gives better results in terms of computational cost. Moreover, LDR not only allows to decrease the overall number of group operations but also provides some trade-offs between squarings and multiplications. In particular, LDR reduces the number of squarings, but increases the number of multiplications. This can be seen as a drawback when using groups such as finite fields, but it can become an advantage on groups where the multiplication is cheaper than squaring.

In the next section, we propose an example of such group an some implementation results.

6 Elliptic curve implementation

Definition 6.1 *An elliptic curve E over a field K denoted by E/K is given by the equation*

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

where $a_1, a_2, a_3, a_4, a_6 \in K$ are such that, for each point (x, y) on E, the partial derivatives do not vanish simultaneously.

In this work, we only deal with curves defined over a prime finite field ($K = \mathbb{F}_p$) of characteristic greater than 3. In this case, the equation can be simplified to $y^2 = x^3 + ax + b$ where $a, b \in K$ and $4a^3 + 27b^2 \neq 0$. The set of points of E/K, plus a special point called point at infinity, is an abelian group. There exist explicit formulae to compute the sum of two points that involves field inversions. When the field inversion operation is considerably costlier than a field multiplication, one usually uses a projective version of the above equation. In this case, a point is represented by three, or more, coordinates, and many such projective coordinate systems have been proposed to speed up elliptic curve group operations. For a complete overview of those coordinates, one can

refer to [7, 10]. Finally, as elliptic curves are considered as additive group, we talk about addition, doubling and scalar multiplication instead of, respectively, multiplication, squaring and exponentiation. One interesting aspect of those curves is that very efficient formulae have been proposed [15] to perform scalar multiplication based on Euclidean addition chains. These formulae make the point additions faster than point doublings.

We present implementations over two types of curves: general Weierstraß(JAC) curves and curves with parameter $a = -3$ (JAC-3). In each case, we use two different strategies for computing the Euclidean addition chain, with and without inversion. For our implementations, we have:

- generated 10000 pseudo random integers in $\{0, \ldots, 2^t - 1\}$, for $t = 160, 256, 512$,
- computed LDR's of the integers for all values of the parameters w (width of the dictionary) and l (bit-size of n_H)
- counted all the operations involved in the point scalar multiplication process.

6.1 Precomputation strategy

Let n be a scalar and P, a point on the curve. After choosing parameters w and l we decompose $n = (n_{t-1} \ldots n_0)_2$ as $n_H * 2^{t-l-1} + n_L$ where $n_H = (n_{t-1} \ldots n_{t-l-1})_2$ and find a short Euclidean addition chain $S = (s_1, \ldots s_l)$ computing n_H. Then, after creating the dictionary $\mathcal{D}_S^w$ and finding the LDR of n_L, we compute $n_H P$ using chain S and the formulae from [15].

At this point, two strategies are considered:

- without inversion: simply consisting in storing the points $(P, i_1 P, i_2 P, \ldots, i_m P)$ used to represent n_L,
- with inversion: consisting of rescaling all the point $(P, i_1 P, i_2 P, \ldots, i_m P)$ using the method proposed in [14]. The original scheme was made for rescaling points computed through the Euclidean chain $(1, 2, 3, 5, 7, \ldots, 2m+1)$. In fact, the same scheme can be used for any set of points based on any Euclidean chains. If m points have been computed, then rescaling all of then so that they all have their z-coordinate equal to 1 cost 1I+1S+$(4(m-1)$+3)M.

6.2 Results

One feature of the elliptic curve group law is that it allows fast composite operations as well as different type of additions. In addition to the classical addition (**ADD**) and doubling (**DBL**) operations, we consider the following operations:

- **readdition (reADD)**: addition of a point that has been added before to another point,
- **mixed addition (mADD)**: addition of a point in affine coordinate (i.e., $Z = 1$) to another point,
- **Z-addition (ZADD)**: addition based on the same z-coordinate trick, used to compute Euclidean addition chain based scalar multiplication.

Table 2 gives the cost of the different curve operations, in terms of field multiplications and squarings. Table 3 summarize the results of our experiments. We compare our work to the most recent and optimized point scalar algorithm. Note that, for fair comparisons, we propose 2 versions of our results on Jac-3 curves. One using formulae from [3] and another one, using recent optimized (opt) doubling-and-addition formulae from [14]. Finally, we do not provide comparisons to the (2,3,5)NAF method for 256 and 512-bit integers, as no such data are given in the original paper.

Curve	DBL	ADD	reADD	mADD	ZADD
Jacobian	1M+8S	11M+5S	10M+4S	7M+4S	5M+2S
Jacobian-3	3M+5S	11M+5S	10M+4S	7M+4S	5M+2S

Table 2. Elliptic curve operations cost

Method	curve	inversion	storage	width	n_H size	# operations
wNAF [4]	JAC	0	7	-	-	1573.8M
	JAC	1	8	-	-	1495.8M
	JAC-3	0	7	-	-	1511.9M
	JAC-3	1	8	-	-	1434.1M
Double-base chains [2]	JAC	0	7	-	-	1558.4M
	JAC-3	0	7	-	-	1504.3M
(2,3,5)NAF [13]	JAC-3 (opt)	0	7	-	-	1448.4M
	JAC-3 (opt)	1	7	-	-	1398.9M
LDR	JAC	0	12	12	18	1519.9M
	JAC	1	7	8	8	1503.0M
	JAC-3	0	12	12	18	1463.5M
	JAC-3	1	7	8	8	1442.6M
	JAC-3 (opt)	0	12	12	18	1447.8M
	JAC-3 (opt)	1	7	8	8	1430.6M

Table 3. Comparison between different scalar multiplication algorithms using 160-bit scalars

Method	curve	inversion	storage	width	n_H size	# operations
wNAF [4]	JAC	0	7	-	-	2492.1M
	JAC	1	8	-	-	2369.2M
	JAC-3	0	7	-	-	2391.8M
	JAC-3	1	8	-	-	2269.2M
Double-base chains [2]	JAC	0	7	-	-	2466.2M
	JAC-3	0	7	-	-	2379.0M
LDR	JAC	0	12	12	18	2416.3M
	JAC	1	10	10	10	2377.7M
	JAC-3	0	12	12	18	2322.3M
	JAC-3	1	10	10	10M	2279.8M
	JAC-3 (opt)	0	12	12	18	2295.3.8M
	JAC-3 (opt)	1	7	8	8	2255.4M

Table 4. Comparison between different scalar multiplication algorithms using 256-bit scalars

Method	curve	inversion	storage	width	n_H size	# operations
wNAF [4]	JAC	0	15	-	-	4885.2M
	JAC	1	16	-	-	4658.4M
	JAC-3	0	15	-	-	4683.0M
	JAC-3	1	16	-	-	4456.3M
Double-base chains [2]	JAC	0	10	-	-	4869.3M
	JAC-3	0	11	-	-	4693.2M
LDR	JAC	0	28	12	32	4763.4M
	JAC	1	18	12	17	4672.4M
	JAC-3	0	28	12	32	4571.8M
	JAC-3	1	18	12	17	4474.9M
	JAC-3 (opt)	0	28	32	18	4521.3M
	JAC-3 (opt)	1	18	12	17	4431.0M

Table 5. Comparison between different scalar multiplication algorithms using 512 bit-scalars

We can see that our method does not take advantage of the 1-inversion strategy. It only allows to save around 20M where more than 50M are saved using the other methods. This is due to the fact that our method requires less general additions (but more Z-additions), which reduces the motivation of rescaling all the stored points. In that case, our method is slightly slower than the algorithms using the inversion strategy. On the other hand, our approach outperforms both the wNAF and double-base chains methods, when using an inversion free strategy. Only the (2,3,5)NAF is faster; however, it must be taken into consideration that the results of (2,3,5)NAF benefit from optimized doubling-and-addition formulae. Using this new formulae with our algorithm reduce its overall cost from 1463.5 to 1447.8, which makes as efficient as the (2,3,5)NAF method. Moreover, the latter uses combination of doubling, tripling and quintupling formulae to speed up the scalar multiplication. Combining this method with our large digit representation should give even better results, but it is out of the scope of this work. As the size of the scalars increases, our method becomes more and more efficient compared to the classical wNAF and double-base chains methods, saving at least 50M for 256-bit integers, to 110M for 512-bits integers.

7 Summary

In this work we have proposed a new integer representation using a set of large digits. The main idea is to use the first few powers naturally computed during any exponentiation algorithm as precomputed points. We have showed that the overall number of operations is very dependent on the choice of the addition chain used to compute those first power and proposed to use Euclidean chains. Both our new representation and Euclidean chains have permitted us to reduce the computational cost of exponentiations, in terms of the total number of operations. We have also presented an effective implementation of our algorithm on elliptic curve defined over prime fields, and have found that our approach gives better results than most of the previous methods. Moreover, we are now working on adapting our large digit representation to the recent (2,3,5)NAF method, which should, in the end, give even better results.

References

[1] Ahmes. Rhind papyrus, -1650.

[2] D. J. Bernstein, P. Birkner, T. Lange, and C. Peters. Optimizing double-base elliptic-curve single-scalar multiplication. In *Progress in Cryptology INDOCRYPT 2007*, volume 4859/2007 of *LNCS*, pages 167–182. Springer Berlin / Heidelberg, 2007.

[3] D. J. Bernstein and T. Lange. Explicit-formulas database. Available at http://hyperelliptic.org/EFD.
[4] D. J. Bernstein and T. Lange. Analysis and optimization of elliptic-curve single-scalar multiplication. In *Finite fields and applications: proceedings of Fq8*, pages 1–19, 2008.
[5] A. Brauer. On addition chains. *Bulletin of the American Mathematical Society*, 45(10):736–739, 1939.
[6] M. Ciet, M. Joye, K. Lauter, and P. L. Montgomery. Trading inversions for multiplications in elliptic curve cryptography. *Designs, codes and Cryptography*, 39(2):189–206, May 2006.
[7] H. Cohen and G. Frey, editors. *Handbook of Elliptic and Hyperelliptic Cryptography*. Chapman & Hall, 2006.
[8] V. Dimitrov and T. Cooklev. Two algorithms for modular exponentiation using nonstandard arithmetics. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 78(1):82–87, 1995.
[9] V. Dimitrov, L. Imbert, and P. K. Mishra. The double-base number system and its application to elliptic curve cryptography. *Mathematics of Computations*, 77, 2008.
[10] D. Hankerson, A. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer, 2004.
[11] D. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, 2 edition, 1981.
[12] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48:203–209, 1987.
[13] P. Longa and C. Gebotys. Setting speed records with the (fractional) multibase non-adjacent form method for efficient elliptic curve scalar multiplication. Technical report, Department of Electrical and Computer Engineering University of Waterloo, Canada, 2009.
[14] P. Longa and A. Miri. New composite operations and precomputation scheme for elliptic curve cryptosystems over prime fields. In *Public Key Cryptography*, volume 4939 of *LNCS*, pages 229–247. Springer Berlin / Heidelberg, 2008.
[15] N. Meloni. New point addition formulae for ECC applications. In *Arithmetic of Finite Fields*, volume 4547 of *LNCS*, pages 189–201. Springer Berlin / Heidelberg, 2007.
[16] V. Miller. Uses of elliptic curves in cryptography. In *Advaces in Cryptology-CRYPTO*, volume 218 of *LNCS*, pages 417–428. Springer, 1986.
[17] B. Möller. Improved techniques for fast exponentiation. In S. B. . Heidelberg, editor, *Information Security and Cryptology ICISC 2002*, volume 2587 of *LNCS*, pages 298–312, 2003.
[18] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120–126, 1978.
[19] E. G. Thurber. On addition chains $l(mn) \leq l(n) - b$ and lower bounds for $c(r)$. *Duke Mathematical Journal*, 40:907–913, 1973.
[20] A. C. Yao. On the evaluation of powers. *SIAM Journal on Computing*, 5(1):100–103, 1976.

A Using negative digits

In this section we give a modified version of Algorithm 5 in which we allow the use of negative digits. It uses the same dictionary as its unsigned counterpart.

Algorithm 6 Computing a signed Large-Digit Representation using a two-row dictionary $\mathcal{D}_S^w$

Input: An integer n a width w and a dictionary D_S^w
Output: $n = (n_t n_{t-1} \dots n_1 n_0)$ such that $n = \sum_{i=0}^{t} n_i 2^i$

$i = 0$
$Zeros = 0$
while $n > 0$ **do**
  if $n \mod 2 \neq 0$ **then**
    $r = s_i \mod 2^{w+1}$
    if $r > 2^w$ **then**
      $r = r - 2^{w+1}$
      if $d_{1,-r} \neq 0$ **and** $d_{1,r} \leq n$ **and** $d_{2,-r} \leq Zeros$ **then**
        $i = i - d_{2,-r};\ n_i = -d_{1,-r}$
        $n = n \times 2^{d_{2,-r}};\ Zeros = 0$
      end if
    end if
    for $j = 0$ to $w - 1$ **do**
      $r = s_i \mod 2^{w-j}$
      if $d_{1,r} \neq 0$ **and** $d_{1,r} \leq n$ **and** $d_{2,r} \leq Zeros$ **then**
        $i = i - d_{2,r};\ n_i = d_{1,r}$
        $n = n \times 2^{d_{2,r}};\ Zeros = 0$
        `break`
      end if
      $r = r - 2^{w-j}$
      if $d_{1,-r} \neq 0$ **and** $d_{1,r} \leq n$ **and** $d_{2,-r} \leq Zeros$ **then**
        $i = i - d_{2,-r};\ n_i = -d_{1,-r}$
        $n = n \times 2^{d_{2,-r}};\ Zeros = 0$
        `break`
      end if
    end for
  else
    $n_i = 0;\ Zeros = Zeros + 1$
  end if
  $n = \frac{n - n_i}{2};\ i = i + 1$
end while
return $n_{i-1} \dots n_1 n_0$