Highly Regular m-Ary Powering Ladders

Marc Joye

Thomson R&D, Security Competence Center,
1 avenue de Belle Fontaine, 35576 Cesson-Sévigné Cedex, France
marc.joye@thomson.net
http://joye.site88.net/

Abstract. This paper describes new exponentiation algorithms with applications to cryptography. The proposed algorithms can be seen as m-ary generalizations of the so-called Montgomery ladder. Both left-to-right and right-to-left versions are presented.

Similarly to Montgomery ladder, the proposed algorithms always repeat the same instructions in the same order, without inserting dummy operations, and so offer a natural protection against certain implementation attacks. Moreover, as they are available in any radix m and in any scan direction, the proposed algorithms offer improved performance and greater flexibility.

Keywords: Exponentiation algorithms, Montgomery ladder, SPA-type attacks, safe-error attacks.

1 Introduction

We consider the general problem of evaluating $y = g^d$ in a (multiplicatively written) group $\mathbb{G}$ with identity element $1_{\mathbb{G}}$, on input $g \in \mathbb{G}$ and $d \in \mathbb{Z}_{>0}$. The m-ary expansion of d is given by $d = \sum_{i=0}^{\ell-1} d_i\, m^i$ with $0 \leqslant d_i < m$ and $d_{\ell-1} \neq 0$. Integer $\ell = \ell(m)$ represents the number of digits (in radix m) for the m-ary representation of d and is called the m-ary length of d.

1.1 Left-to-Right Algorithms

The most widely used exponentiation algorithm is the *binary method* (a.k.a. "square-and-multiply" algorithm) [15, Section 4.6.3]. It relies on the simple observation that $g^d = \left(g^{d/2}\right)^2$ if d is even, and $g^d = \left(g^{(d-1)/2}\right)^2 \cdot g$ if d is odd.

The binary method extends easily to any radix m. Let $H_i = \sum_{j=i}^{\ell-1} d_j\, m^{j-i}$. Since $H_i = \left(\sum_{j=i+1}^{\ell-1} d_j\, m^{j-i}\right) + d_i = mH_{i+1} + d_i$, we get

$$g^{H_i} = \begin{cases} \left(g^{H_{i+1}}\right)^m & \text{if } d_i = 0\,, \\ \left(g^{H_{i+1}}\right)^m \cdot g^{d_i} & \text{otherwise}\ . \end{cases} \tag{1}$$

Noting that $g^d = g^{H_0}$, the previous relation gives rise to an exponentiation algorithm. It can be readily programmed by scanning the m-ary representation

M.J. Jacobson Jr., V. Rijmen, and R. Safavi-Naini (Eds.): SAC 2009, LNCS 5867, pp. 350–363, 2009.

of d from left to right. As, at iteration i, for $\ell - 2 \geqslant i \geqslant 0$, the method requires a multiplication by g^{d_i} when $d_i \neq 0$, the values of g^j with $1 \leqslant j \leqslant m-1$ are precomputed and stored in $(m-1)$ temporary variables; namely, $\mathsf{R}[j] \leftarrow g^j$ for $1 \leqslant j \leqslant m-1$. If the successive values of g^{H_i} are kept track of in an accumulator A, Equation (1) then translates into

$$\mathsf{A} \leftarrow \begin{cases} \mathsf{A}^m & \text{if } d_i = 0 \\ \mathsf{A}^m \cdot \mathsf{R}[d_i] & \text{otherwise} \end{cases} \qquad (\text{for } \ell - 2 \geqslant i \geqslant 0)$$

and where A is initialized to $\mathsf{R}[d_{\ell-1}]$. The corresponding algorithm is referred to as the *(left-to-right) m-ary algorithm.*

1.2 Right-to-Left Algorithms

It is also possible to devise a similar algorithm based on a right-to-left scan of exponent d. This may be convenient when the m-ary length of d is unknown in advance. In the binary case (i.e., when $m = 2$), letting $d = \sum_{i=0}^{\ell-1} d_i\, 2^i$ the binary expansion of d, the method makes use of the relation $g^d = \prod_{\substack{0 \leqslant i \leqslant \ell-1 \\ d_i \neq 0}} g^{2^i}$. An accumulator A is initialized to g and squared at each iteration, so that it contains g^{2^i} at iteration i. Another accumulator, say $\mathsf{R}[1]$, initialized to $1_{\mathbb{G}}$, is multiplied with A if $d_i \neq 0$. Hence, we see that at iteration $\ell - 1$, accumulator $\mathsf{R}[1]$ contains the value of $\prod_{\substack{0 \leqslant i \leqslant \ell-1 \\ d_i \neq 0}} g^{2^i} = g^d$.

Although less known than its left-to-right counterpart, as shown by Yao [29], this method can be extended to higher radices. The basic idea remains the same. If $d = \sum_{i=0}^{\ell-1} d_i\, m^i$ denotes the m-ary expansion of d, we can write

$$\begin{aligned} g^d &= \prod_{\substack{0 \leqslant i \leqslant \ell-1 \\ d_i = 1}} g^{m^i} \cdot \prod_{\substack{0 \leqslant i \leqslant \ell-1 \\ d_i = 2}} g^{2 \cdot m^i} \cdots \prod_{\substack{0 \leqslant i \leqslant \ell-1 \\ d_i = m-1}} g^{(m-1) \cdot m^i} \\ &= \prod_{j=1}^{m-1} (L_j)^j \quad \text{where } L_j = \prod_{\substack{0 \leqslant i \leqslant \ell-1 \\ d_i = j}} g^{m^i} \ . \end{aligned} \qquad (2)$$

Hence, using $(m-1)$ accumulators, $\mathsf{R}[1], \ldots, \mathsf{R}[m-1]$, to keep track of the values of L_j, $1 \leqslant j \leqslant m-1$, and an accumulator A that stores the successive values of g^{m^i}, at iteration i, the accumulators are updated as

$$\begin{cases} \mathsf{R}[d_i] \leftarrow \mathsf{R}[d_i] \cdot \mathsf{A} & \text{if } d_i \neq 0 \\ \mathsf{A} \leftarrow \mathsf{A}^m \end{cases} \qquad (\text{for } 1 \leqslant i \leqslant \ell - 1)$$

where A is initialized to g and $\mathsf{R}[1], \ldots, \mathsf{R}[m-1]$ are initialized to $1_{\mathbb{G}}$. Equation (2) says that g^d is then given by $\mathsf{A} \leftarrow \prod_{j=1}^{m-1} \mathsf{R}[j]^j$. The so-obtained algorithm, also known as *Yao's algorithm*, is referred to as the *right-to-left m-ary algorithm.*

1.3 Implementation Attacks

If not properly implemented, exponentiation algorithms may be vulnerable to *side-channel attacks* [16,17] (see also [6,19]). Another threat against implementations of exponentiation algorithms resides in fault attacks [5] (see also [2,11]).

Two implementation attacks, namely *SPA-type attacks* and *safe-error attacks*, are particularly relevant in the context of exponentiation.

SPA-Type Attacks. By observing a suitable side channel, such as the power consumption [16] or electromagnetic emanations [10,24], an attacker may recover secret information. For exponentiation-based cryptosystems, the goal of the attacker is to recover the value of exponent d (or a part thereof) used in the computation of g^d in some group $\mathbb{G}$. *SPA-type attacks*[1] assume that the attacker infers secret information (typically one or several bits of d) from a single execution of g^d.

Consider for example the square-and-multiply algorithm (that is, the left-to-right m-ary algorithm with $m = 2$).[2]

Algorithm 1. Square-and-Multiply Algorithm

Input: $g \in \mathbb{G}$, $d = \sum_{i=0}^{\ell-1} d_i\, 2^i$
Output: g^d

1 $\mathsf{R}[1] \leftarrow g$; $\mathsf{A} \leftarrow 1_{\mathbb{G}}$
2 **for** $i = \ell - 1$ **down to** 0 **do**
3 $\quad \mathsf{A} \leftarrow \mathsf{A}^2$
4 $\quad$ **if** $(d_i \neq 0)$ **then** $\mathsf{A} \leftarrow \mathsf{A} \cdot \mathsf{R}[1]$
5 **end**
6 **return** A

Each iteration comprises a 'square' and, when the bit exponent is non-zero, a subsequent 'multiply'. Since the algorithm behaves differently depending on the bit values, this may be observed from a suitable side channel. The information thus gleaned may enable the attacker to deduce one or more bits of exponent d.

One way of preventing an attacker from recovering the bit values is to execute the same instructions regardless of the value of input bit d_i. Such an algorithm is said to be *regular*. There are several implementations of this idea.

- The test of whether a digit is nonzero may be removed if Line 1 in Algorithm 1 is replaced with $\mathsf{A} \leftarrow \mathsf{A} \cdot \mathsf{R}[d_i]$ and where temporary variable $\mathsf{R}[0]$ is initialized to $1_{\mathbb{G}}$. Alternatively, a fake multiply may be performed when $d_i = 0$, as suggested in [9]. Doing so, there will be no longer conditional branchings: at each iteration, there is a square always followed by a multiply.

[1] SPA stands for "Simple Power Analysis."

[2] We slightly differ from the presentation of Section 1.1 and initialize accumulator A with $1_{\mathbb{G}}$. This prevents the necessity of requiring $d_{\ell-1} \neq 0$ and therefore ℓ may denote any upper bound on the binary length of d. If $\ell' \leqslant \ell$ is the exact binary length of d, observe that accumulator A is correctly set in the for-loop to $g^{d_{\ell'-1}}$, as required.

This algorithm is known as the "square-and-multiply-always" algorithm. However, as will be explained in a moment, the resulting implementation now becomes vulnerable to safe-error attacks.
- Another possibility to get a regular exponentiation is to recode exponent d in such a way that none of the digits are zero [21,23,27,28]. As exemplified in [26], this however supposes that the recoding algorithm itself is resistant to SPA-type attacks.

The above analysis is not restricted to the square-and-multiply algorithm and generalizes to the m-ary exponentiation algorithms mentioned in Sections 1.1 and 1.2. While it may argued that, for larger m, m-ary exponentiation algorithms are more regular and therefore more resistant to SPA-type attacks, these algorithms are not entirely regular since two cases are to be distinguished: $d_i = 0$ and $d_i \neq 0$.

Safe-Error Attacks. By timely inducing a fault during the execution of an instruction, an attacker may deduce whether the targeted instruction is fake: if the final result is correct then the instruction is indeed fake (or dummy); if not, the instruction is effective. This knowledge may then be used to obtain one or more bits of exponent d. Such attacks are referred to as *safe-error attacks* [30,31].

Back to the "square-and-multiply-always" algorithm, an attacker can induce a fault during a multiply. If the final result is correct then the attacker may deduce that the corresponding exponent bit is a zero (i.e., fake multiply); otherwise, the attacker may deduce that the exponent bit is a one. Safe-error attacks apply likewise to higher-radix similar m-ary methods to distinguish zero digits.

1.4 Our Contributions

Using the terminology of [12], we deal in this paper with *highly* regular exponentiation algorithms, that is, exponentiation algorithms that

- are regular; i.e., always repeat the same instructions in the same order for any inputs;
- do not insert dummy operations.

Highly regular exponentiation algorithms protect against SPA-type attacks *and* safe-error attacks, at the same time [14]. Examples of such algorithms include the so-called Montgomery ladder [22] and a recent powering ladder presented at CHES 2007 [12, Algorithm 1″]. These two algorithms are depicted below.

Algorithm 2. Montgomery Ladder

Input: $g \in \mathbb{G}$, $d = \sum_{i=0}^{\ell-1} d_i\, 2^i$
Output: g^d

1 $\mathsf{R}[0] \leftarrow 1_{\mathbb{G}}$; $\mathsf{R}[1] \leftarrow g$
2 **for** $i = \ell - 1$ **down to** 0 **do**
3 $\quad \mathsf{R}[1 - d_i] \leftarrow \mathsf{R}[1 - d_i] \cdot \mathsf{R}[d_i]$
4 $\quad \mathsf{R}[d_i] \leftarrow \mathsf{R}[d_i]^2$
5 **end**
6 **return** $\mathsf{R}[0]$

Algorithm 3. Joye's Square-Multiply Ladder

Input: $g \in \mathbb{G}$, $d = \sum_{i=0}^{\ell-1} d_i\, 2^i$
Output: g^d

```
1 R[0] ← 1_G; R[1] ← g
2 for i = 0 to ℓ − 1 do
3     R[1 − d_i] ← R[1 − d_i]^2 · R[d_i]
4 end
5 return R[0]
```

Montgomery ladder and Joye's square-multiply ladder both rely on specific properties of the binary representation. In particular, it is unclear how to generalize these two algorithms to higher radices.

In this paper, we present a new method to derive highly regular exponentiation algorithms by considering a representation of $d-1$ rather than that of plain exponent d. The proposed method is independent of the radix representation and of the scan direction (left-to-right or right-to-left). Interestingly, when particularized to $m = 2$, the method yields algorithms dual to Algorithms 3 and 2; i.e., similar algorithms but with the opposite scan direction.

Outline of the Paper. The rest of this paper is organized as follows. The next section is the core of our paper. We describe our new exponentiation algorithms. In Section 3, we present some applications thereof. Finally, we conclude in Section 4.

2 New Exponentiation Algorithms

As aforementioned, the goal is to evaluate $y = g^d$ given an element $g \in \mathbb{G}$ and an ℓ-digit exponent $d = \sum_{i=0}^{\ell-1} d_i\, m^i$. Our algorithms rely on the following proposition.

Proposition 1. *Let $d = \sum_{i=0}^{\ell-1} d_i\, m^i$ denote the m-ary expansion of d. Then*

$$d = (d_{\ell-1} - 1)m^{\ell-1} + \left(\sum_{i=0}^{\ell-2}(d_i + m - 1)m^i\right) + 1 \ .$$

Proof. Straightforward by noting that $\sum_{i=0}^{\ell-2}(d_i + m - 1)m^i = \sum_{i=0}^{\ell-2} d_i\, m^i + \sum_{i=0}^{\ell-2}(m-1)m^i = (d - d_{\ell-1}\, m^{\ell-1}) + (m^{\ell-1} - 1)$. □

2.1 General Case

Proposition 1 can be rewritten as

$$d - 1 = \sum_{i=0}^{\ell-1} d_i^*\, m^i \quad \text{where } d_i^* = \begin{cases} d_i + m - 1 & \text{for } 0 \leqslant i \leqslant \ell - 2 \\ d_{\ell-1} - 1 & \text{for } i = \ell - 1 \end{cases} . \qquad (3)$$

Left-to-Right Algorithm. If $d > 0$, it follows that $d_{\ell-1} \geqslant 1$ and so $d^*_{\ell-1} \geqslant 0$. Remember that the m-ary algorithm can accommodate a leading zero digit (i.e., when $d^*_{\ell-1} = 0$); see Footnote 2. It is also important to note that all the subsequent digits are nonzero (i.e., $d^*_i > 0$ for $i \leqslant \ell - 2$). We can therefore devise a *regular* method to get the value of g^{d-1} for some $d > 0$. The value of $y = g^d$ is then obtained as $y = g^{d-1} \cdot g$.

The algorithm is an adaptation of the m-ary algorithm, as described in Section 1.1. It makes use of an accumulator A, initialized to $g^{d^*_{\ell-1}}$. At each iteration of the main loop, accumulator A is raised to the power of m and then always multiplied by $g^{d^*_i}$ (remember that $d^*_i \neq 0$). Since $d^*_i \in \{m-1, \ldots, 2m-2\}$, the values of $g^{m-1}, \ldots, g^{2m-2}$ are precomputed and stored in temporary variables $\mathsf{R}[1] \ldots, \mathsf{R}[m]$. At the end of the main loop, the accumulator is multiplied by g to get the correct result.

Precomputation & Initialization. Accumulator A has to be initialized to $g^{d^*_{\ell-1}}$ with $d^*_{\ell-1} = (d_{\ell-1} - 1)$ in $\{0, \ldots, m-2\}$ and this must be done in a regular manner. Moreover, since *(i)* the values of $g^{m-1}, \ldots, g^{2m-2}$ have to be precomputed and stored in registers $\mathsf{R}[1], \ldots, \mathsf{R}[m-1]$ before entering the main loop and *(ii)* $d_{\ell-1} \in \{1, \ldots, m-1\}$, it is possible to

1. write g^{j-1} in $\mathsf{R}[j]$ for $1 \leqslant j \leqslant m$,
2. assign A to the corresponding register so that it contains $g^{d_{\ell-1}-1}$ (i.e., $\mathsf{A} \leftarrow \mathsf{R}[d_{\ell-1}]$), and
3. multiply registers $\mathsf{R}[1], \ldots, \mathsf{R}[m]$ by g^{m-1} so that they contain $g^{m-1}, \ldots, g^{2m-2}$, respectively;

or algorithmically, we replace Lines 1 and 2 in Algorithm 4 with

```
   ▷ Precomputation & Initialization
1  R[1] ← 1_G; R[2] ← g;  for i = 3 to m do R[i] ← R[i − 1] · R[2]
2  A ← R[d_{ℓ−1}];  for i = 1 to m do R[i] ← R[i] · R[m]
```

Doing so, the evaluation of $g^{d_{\ell-1}-1}$ is regular.

Algorithm 4. Regular Left-to-Right Exponentiation (General description)

Input: $g \in \mathbb{G}$, $d = \sum_{i=0}^{\ell-1} d_i\, m^i$ $(d > 0)$
Output: g^d
Uses: A and $\mathsf{R}[1], \ldots, \mathsf{R}[m]$

```
   ▷ Precomputation & Initialization
1  for i = 1 to m do R[i] ← g^{m+i−2}
2  A ← g^{d_{ℓ−1}−1}
   ▷ Main loop
3  for i = ℓ − 2 down to 0 do
4      A ← A^m · R[1 + d_i]
5  end
   ▷ Final correction
6  A ← A · g
7  return A
```

Yet another way of obtaining a regular evaluation is to force the leading digit to a predetermined value by adding to d a suitable multiple of the order of g prior to the exponentiation. When applicable, this method should be preferred. Furthermore, it nicely combines with the classical DPA countermeasure consisting in adding to d a random multiple of the order of g [9].

Final correction. The final correction can be avoided by replacing d with $d+1$ prior to the exponentiation, $d \leftarrow d+1$. This may be useful when the memory is scarce and that the value of g is not available in memory. Note also that this step may be combined with the addition of a multiple of the order of g.

Right-to-Left Algorithm. We can likewise devise a right-to-left m-ary exponentiation algorithm. We follow the presentation of Section 1.2. From Equation (3), we have

$$g^{d-1} = \left(g^{m^{\ell-1}}\right)^{d^*_{\ell-1}} \cdot \prod_{j=1}^{m-1} (L^*_j)^{m+j-2} \quad \text{where } L^*_j = \prod_{\substack{0 \leqslant i \leqslant \ell-2 \\ d^*_i = j}} g^{m^i} \ . \tag{4}$$

The algorithm makes use of m accumulators, $\mathsf{R}[1], \ldots, \mathsf{R}[m]$, to keep track of the values of L^*_j, $1 \leqslant j \leqslant m$, and an accumulator that keeps track of the successive values of g^{m^i}. Accumulators $\mathsf{R}[1], \ldots, \mathsf{R}[m]$ are initialized to $1_{\mathbb{G}}$ and accumulator A is initialized to g. Again, it is to be noted that all digits d^*_i are nonzero (i.e., $d^*_i \in \{m-1, \ldots, 2m-2\}$ for $0 \leqslant i \leqslant \ell-2$). As a consequence, at each iteration i, an accumulator $\mathsf{R}[j]$ is updated (namely, $\mathsf{R}[d^*_i] \leftarrow \mathsf{R}[d^*_i] \cdot \mathsf{A}$) and accumulator A is updated as $\mathsf{A} \leftarrow \mathsf{A}^m$. Hence, we see that the evaluation of L^*_j is regular. It then remains to evaluate the above relation in a regular manner to obtain a *regular* right-to-left m-ary exponentiation algorithm to get g^{d-1} and thus $y = g^d$ as $g^{d-1} \cdot g$.

Initialization. In certain groups, the neutral element $1_{\mathbb{G}}$ requires special treatment (e.g., elliptic curves given by the Weierstraß form).[3] In such groups, the multiplication of two elements A and B is typically implemented by checking whether A or B is $1_{\mathbb{G}}$: if this is the case, then the other element is returned; if not, the 'regular' multiplication, $A \cdot B$, is evaluated and returned. As this may be observed through SPA, this can leak the first occurrence of a digit in $\{0, \ldots, m-1\}$ in the m-ary representation of d. One way to prevent this leakage is to initialize $\mathsf{R}[1], \ldots, \mathsf{R}[m]$ to values different from $1_{\mathbb{G}}$.

[3] By special treatment, we mean that the group operation is not unified. The usual addition formulas obtained by the chord-and-tangent rule on Weierstraß elliptic curves are not valid for $1_{\mathbb{G}}$ (i.e., the point at infinity). In contrast, in $\mathbb{G} = \mathbb{Z}^*_N$, neutral element $1_{\mathbb{G}} = 1$ does not require a special treatment. Further, in this latter case, it is easy to get SPA-resistance even if multiplication by 1 modulo N may be observed through some side channel. For example, this can be achieved by working in $\mathbb{Z}^*_{2^w N}$ and replacing 1 with an equivalent representation $1 + \alpha N$; the correct result is then obtained by reducing the final output modulo N.

Algorithm 5. Regular Right-to-Left Exponentiation (General description)

Input: $g \in \mathbb{G}$, $d = \sum_{i=0}^{\ell-1} d_i\, m^i$ $(d > 0)$
Output: g^d
Uses: A and $\mathsf{R}[1], \ldots, \mathsf{R}[m]$

▷ `Initialization`
1 **for** $i = 1$ **to** m **do** $\mathsf{R}[i] \leftarrow 1_{\mathbb{G}}$
▷ `Main loop`
2 $\mathsf{A} \leftarrow g$
3 **for** $i = 0$ **to** $\ell - 2$ **do**
4 $\quad \mathsf{R}[1 + d_i] \leftarrow \mathsf{R}[1 + d_i] \cdot \mathsf{A}$
5 $\quad \mathsf{A} \leftarrow \mathsf{A}^m$
6 **end**
▷ `Aggregation`
7 $\mathsf{A} \leftarrow \mathsf{A}^{d_{\ell-1}-1} \cdot \prod_{i=1}^{m} \mathsf{R}[i]^{m+i-2}$
▷ `Final correction`
8 $\mathsf{A} \leftarrow \mathsf{A} \cdot g$
9 **return** A

As an example $\mathsf{R}[1], \ldots, \mathsf{R}[m]$ are initialized to g. Since each $\mathsf{R}[i]$ will be raised to the power of $(m + i - 2)$ during the aggregation step, we subtract $\sum_{i=1}^{m}(m + i - 2) = \frac{3m(m-1)}{2}$ from d prior to the exponentiation. In more detail, we replace Line 1 in Algorithm 5 with

▷ `Initialization`
1a **for** $i = 1$ **to** m **do** $\mathsf{R}[i] \leftarrow g$
1b $d \leftarrow d - 3m(m-1)/2$

In groups where inverses can be easily obtained (e.g., on elliptic curves), another option is to keep the value of d unchanged but to correct the result at the end of the computation. This can be for example achieved by replacing Line 8 in Algorithm 5 with

▷ `Final correction`
8 $\mathsf{A} \leftarrow \mathsf{A} \cdot g^{3m(m-1)/2+1}$

Alternatively, $\mathsf{R}[1], \ldots, \mathsf{R}[m]$ can be initialized to elements of small order in $\mathbb{G}$. Suppose that $\mathsf{R}[1], \ldots, \mathsf{R}[m]$ are all initialized to h with $\mathrm{ord}_{\mathbb{G}}(h) = t$. Define $b = 3m(m-1)/2 \bmod t$. At the end of the computation, accumulator A then contains a multiplicative surplus factor of h^b. Hence, the correct result is obtained by multiplying A by h^{t-b}. For example, in RSA groups $\mathbb{G} = \mathbb{Z}_N^*$, we can take $h = N - 1$, which is of order $t = 2$.

Aggregation. If done naively, the aggregation step at Line 5 (i.e., the evaluation of $\prod_{i=1}^{m} \mathsf{R}[i]^{m+i-2}$) can be somewhat expensive. We extend a technique described in [15, p. 634] to suit our present needs. It requires an accumulator A initialized to $\mathsf{R}[m]$. If we set $\mathsf{R}[i] \leftarrow \mathsf{R}[i] \cdot \mathsf{R}[i + 1]$ and $\mathsf{A} \leftarrow \mathsf{A} \cdot \mathsf{R}[i]$ for $i = m - 1, \ldots, 1$,

we end up with $\mathsf{R}[1] \leftarrow \prod_{1 \leqslant i \leqslant m} \mathsf{R}[i]$ and $\mathsf{A} \leftarrow \prod_{i=1}^{m} \mathsf{R}[i]^i$. Therefore, writing $\prod_{i=1}^{m} \mathsf{R}[i]^{m+i-2}$ as $\prod_{i=1}^{m} \mathsf{R}[i]^i \cdot (\prod_{i=1}^{m} \mathsf{R}[i])^{m-2}$, we can use the above technique to get it as $\mathsf{A} \cdot \mathsf{R}[1]^{m-2}$. In our case, accumulator A is initialized to $\mathsf{A}^{d_{\ell-1}-1} \cdot \mathsf{R}[m]$ to get the value of g^{d-1} as per Eq. (4).

▷ `Aggregation`
7a $\mathsf{A} \leftarrow \mathsf{A}^{d_{\ell-1}-1}$; $\mathsf{A} \leftarrow \mathsf{A} \cdot \mathsf{R}[m]$
7b **for** $i = m-1$ **down to** 1 **do**
7c $\quad \mathsf{R}[i] \leftarrow \mathsf{R}[i] \cdot \mathsf{R}[i+1]$; $\mathsf{A} \leftarrow \mathsf{A} \cdot \mathsf{R}[i]$
7d **end**
7e $\mathsf{A} \leftarrow \mathsf{A} \cdot \mathsf{R}[1]^{m-2}$

The initialization of accumulator A (i.e., $\mathsf{A} \leftarrow \mathsf{A}^{d_{\ell-1}-1}$) must be performed in a regular manner. An easy way to do so is to add to d a suitable multiple of the order of g so as to force the leading digit of the resulting d to a predetermined value. An alternative method is described in Appendix A.

Final correction. As for the left-to-right version, the final correction can be avoided by replacing d with $d+1$. Again, this step can be combined with other steps, including the initialization step when neutral element needs a special treatment or the initialization of accumulator A in the aggregation step to force the leading digit.

2.2 Binary Case

The m-ary algorithms we developed are subject to numerous variants. We present now algorithms tailored to the binary case.

In the binary case, we have $m = 2$ and thus, provided that $d > 0$, $d_{\ell-1} = 1$. Equation (3) then simplifies to $d - 1 = \sum_{i=0}^{\ell-2} d_i^* \, 2^i$ with $d_i^* = d_i + 1$.

Left-to-Right Algorithm. We can use Algorithm 4 as is, where m is set to 2 and accumulator A is initialized to $g^{d_{\ell-1}-1} = 1_{\mathbb{G}}$. Alternatively, assuming $d > 1$ (and thus $\ell \geqslant 2$), we can initialize the accumulator to $g^{d_{\ell-2}^*}$ and start the loop at index $\ell - 3$; this avoids dealing with neutral element $1_{\mathbb{G}}$.

Algorithm 6. Regular Left-to-Right Binary Exponentiation

Input: $g \in \mathbb{G}$, $d = \sum_{i=0}^{\ell-1} d_i \, 2^i$ $(d > 1)$
Output: g^d

1 $\mathsf{R}[1] \leftarrow g$; $\mathsf{R}[2] \leftarrow \mathsf{R}[1]^2$
2 $\mathsf{A} \leftarrow \mathsf{R}[1 + d_{\ell-2}]$
3 **for** $i = \ell - 3$ **down to** 0 **do**
4 $\quad \mathsf{A} \leftarrow \mathsf{A}^2 \cdot \mathsf{R}[1 + d_i]$
5 **end**
6 $\mathsf{A} \leftarrow \mathsf{A} \cdot \mathsf{R}[1]$
7 **return** A

Right-to-Left Algorithm. A direct application of Algorithm 5 with $m = 2$ yields a regular right-to-left algorithm. To prevent the final correction,[4] assuming $d > 1$, we can initialize accumulators $\mathsf{R}[1]$ to g^{d_0} and $\mathsf{R}[2]$ to g. We then swap the order of squaring and multiplication and start the loop at index 1.

Algorithm 7. Regular Right-to-Left Binary Exponentiation

Input: $g \in \mathbb{G}$, $d = \sum_{i=0}^{\ell-1} d_i\, 2^i$ $(d > 1)$
Output: g^d

1 $\mathsf{R}[1] \leftarrow g^{d_0}$; $\mathsf{R}[2] \leftarrow g$
2 $\mathsf{A} \leftarrow \mathsf{R}[2]$
3 **for** $i = 1$ **to** $\ell - 2$ **do**
4 $\quad \mathsf{A} \leftarrow \mathsf{A}^2$
5 $\quad \mathsf{R}[1 + d_i] \leftarrow \mathsf{R}[1 + d_i] \cdot \mathsf{A}$
6 **end**
7 $\mathsf{A} \leftarrow \mathsf{R}[1] \cdot \mathsf{R}[2]^2$
8 **return** A

Implementation notes. In some cases, exponent d is known to be odd (this is for example the case in RSA [25]). If so, $\mathsf{R}[1]$ can be initialized to g. When the least significant bit of d is arbitrary, $\mathsf{R}[1]$ and $\mathsf{R}[2]$ can be initialized as $\mathsf{R}[1] \leftarrow 1_{\mathbb{G}}$; $\mathsf{R}[2] \leftarrow g$; $\mathsf{R}[1] \leftarrow \mathsf{R}[1] \cdot \mathsf{R}[1 + d_0]$. Yet another strategy, provided that the order of g is odd, is to add a suitable multiple thereof to force the parity of d.

Comparison. It is striking to see the resemblance between the so-obtained algorithms (i.e., Algorithms 6 and 7) and Algorithms 3 and 2, respectively. For Algorithm 7 and Montgomery ladder, this is even more apparent from the general description (i.e., when the multiply is performed prior the squaring). Actually, our algorithms when $m = 2$ may be considered as dual of Algorithms 3 and 2 in the sense that they execute similar instructions but scan the exponent in the opposite direction.

3 Further Results

The proposed exponentiation algorithms apply to any group $\mathbb{G}$. In this section, we exploit some of their features to get faster yet secure implementations in certain groups. Our focus will be on the group of points of an elliptic curve over a large prime field. We note however that similar speed-ups may be available in other groups.

Composite Group Operations. Elliptic curves over prime field $\mathbb{F}_p$ are usually implemented using Jacobian coordinates. A point P on elliptic curve E given by

$$E_{/\mathbb{F}_p} : Y^2 = X^3 + a_4 X Z^4 + a_6 Z^6$$

[4] Note that, contrarily to the left-to-right version, the value of g is not readily available from $\mathsf{R}[1]$.

is then represented as a triple $(X_1 : Y_1 : Z_1)$. Such a representation is not unique: $(X_2 : Y_2 : Z_2) \sim (X_1 : Y_1 : Z_1)$ if $X_2 = \lambda^2 X_1$, $Y_2 = \lambda^3 Y_1$ and $Z_2 = \lambda Z_1$ for some nonzero $\lambda \in \mathbb{F}_p$. We refer the reader to [3,4] for state-of-the-art formulas for point addition and point doubling in Jacobian coordinates.

In [20], Meloni developed new point addition formulas for points with the same Z-coordinate. This technique was successfully applied in [18] to derive efficient composite point addition formulas of the form $kP + Q$ for some $k \geqslant 2$. The key observation is that the intermediate calculations in the computation of $P + Q = (X_3 : Y_3 : Z_3)$ with $Z_3 = \alpha Z_1$ involve quantities $\alpha^2 X_1$ and $\alpha^3 Y_1$. Initial point P can then be viewed as $(\alpha^2 X_1 : \alpha^3 Y_1 : Z_3)$ and the evaluation of $2P + Q$ can be done as $(P + Q) + P$ where P and $P + Q$ have the same Z-coordinate. This technique can be used recursively to obtain the value of $kP + Q$.

As the main loop of our regular left-to-right exponentiation algorithm (Algorithm 4) consists of evaluating such a composite operation (i.e., $m\mathsf{A} + \mathsf{R}[1 + d_i]$ in additive notation), it can benefit from these improved formulas for faster computation of a point multiple on E.

Repeated Powerings. Building on [7], Cohen et al. [8] suggested considering mixed coordinate systems for representing points. An interesting case for point doubling is when curve parameter a_4 is equal to -3 as it saves some multiplications (in $\mathbb{F}_p$). Similar performance for an *arbitrary* parameter a_4 can be achieved by representing points in modified Jacobian coordinates, namely tuples of the form $(X_1 : Y_1 : Z_1 : W_1)$ where $W_1 = a_4 {Z_1}^4$.

For efficiency purposes, m is usually chosen as a power of 2, say $m = 2^k$, in m-ary exponentiation algorithms. Raising to the power of m (resp. multiplying by scalar m, in additive notation) then amounts to computing k squarings (resp. k doublings). As in [13], our right-to-left m-ary algorithm (Algorithm 5) repeatedly updates accumulator A as $\mathsf{A} \leftarrow \mathsf{A}^m$ (resp. $\mathsf{A} \leftarrow m\mathsf{A}$). The key observation here is that accumulator A is only modified in this step during the main loop (i.e., Line 5 in Algorithm 5).

As a consequence, back to elliptic curves, dP can be evaluated using mixed coordinate systems: $\mathsf{R}[1], \ldots, \mathsf{R}[m]$ are tuples $(X : Y : Z)$ representing points in Jacobian coordinates and A is a tuple $(X : Y : Z : W)$ representing a point in modified Jacobian coordinates. Line 5 (i.e., $\mathsf{R}[1+d_i] \leftarrow \mathsf{R}[1+d_i]+\mathsf{A}$ using additive notation) only use the three first coordinates of A to evaluate a regular Jacobian point addition whereas Line 5 (i.e., $\mathsf{A} \leftarrow 2^k\mathsf{A}$ in additive notation) updates accumulator A as a series of k doublings in modified Jacobian coordinates. This allows one to have a fast point doubling without increasing the cost of a point addition, regardless of the value of a_4. More precisely, the evaluation of dP can be implemented using the fastest formulas [3] for both point doubling (i.e., the same speed as when $a_4 = -3$ even if $a_4 \neq -3$) and point addition.

Other improvements using different mixed coordinate systems for right-to-left algorithms can be found in [1].

4 Conclusion

In this paper, we developed new m-ary exponentiation algorithms. Remarkably, the proposed algorithms are highly regular: they always repeat the same (effective) instructions in the same order. This feature is useful in the implementation of exponentiation-based cryptosystems protected against SPA-type attacks and safe-error attacks. Contrary to previous regular exponentiation algorithms, our algorithms are not restricted to radix 2 but are available in any radix m. They can also accommodate a left-to-right or a right-to-left exponent scanning. Both scan directions have their own advantages. Furthermore, being generic, we note that the proposed algorithms can easily be combined with other known countermeasures to protect against other classes of attacks, including DPA-type attacks and fault attacks.

Acknowledgments. I am grateful to the anonymous referees for useful comments.

References

1. Avanzi, R.M.: Delaying and merging operations in scalar multiplication: Applications to curve-based cryptosystems. In: Biham, E., Youssef, A.M. (eds.) SAC 2006. LNCS, vol. 4356, pp. 203–219. Springer, Heidelberg (2007)
2. Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., Whelan, C.: The sorcerer's apprentice guide to fault attacks. Proceedings the IEEE 94(2), 370–382 (2004); Earlier version in Proc. of FDTC 2004
3. Bernstein, D.J., Lange, T.: Explicit-formulas database, `http://www.hyperelliptic.org/EFD/jacobian.html`
4. Bernstein, D.J., Lange, T.: Faster addition and doubling on elliptic curves. In: Kurosawa, K. (ed.) ASIACRYPT 2007. LNCS, vol. 4833, pp. 29–50. Springer, Heidelberg (2007)
5. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the importance of eliminating errors in cryptographic computations. Journal of Cryptology 14(2), 110–119 (2001); Extended abstract in Proc. of EUROCRYPT 1997
6. Koç, Ç.K. (ed.): Cryptographic Engineering. Springer, Heidelberg (2009)
7. Chudnovsky, D.V., Chudnovsky, G.V.: Sequences of numbers generated by addition in formal groups and new primality and factorization tests. Advances in Applied Mathematics 7(4), 385–434 (1986)
8. Cohen, H., Miyaji, A., Ono, T.: Efficient elliptic curve exponentiation using mixed coordinates. In: Ohta, K., Pei, D. (eds.) ASIACRYPT 1998. LNCS, vol. 1514, pp. 51–65. Springer, Heidelberg (1998)
9. Coron, J.-S.: Resistance against differential power analysis for elliptic curve cryptosystems. In: Koç, Ç.K., Paar, C. (eds.) CHES 1999. LNCS, vol. 1717, pp. 292–302. Springer, Heidelberg (1999)
10. Gandolfi, K., Mourtel, C., Olivier, F.: Electromagnetic analysis: Concrete results. In: Koç, Ç.K., Naccache, D., Paar, C. (eds.) CHES 2001. LNCS, vol. 2162, pp. 251–261. Springer, Heidelberg (2001)
11. Giraud, C., Thiebeauld, H.: A survey on fault attacks. In: Quisquater, J.-J., et al. (eds.) Smart Card Research and Advanced Applications VI (CARDIS 2004), pp. 159–176. Kluwer, Dordrecht (2004)

12. Joye, M.: Highly regular right-to-left algorithms for scalar multiplication. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 135–147. Springer, Heidelberg (2007)
13. Joye, M.: Fast point multiplication on elliptic curves without precomputation. In: von zur Gathen, J., Imaña, J.L., Koç, Ç.K. (eds.) WAIFI 2008. LNCS, vol. 5130, pp. 36–46. Springer, Heidelberg (2008)
14. Joye, M., Yen, S.-M.: The Montgomery powering ladder. In: Kaliski Jr., B.S., Koç, Ç.K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 291–302. Springer, Heidelberg (2003)
15. Knuth, D.E.: The Art of Computer Programming, 2nd edn. Seminumerical Algorithms, vol. 2. Addison-Wesley, Reading (1981)
16. Kocher, P., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 388–397. Springer, Heidelberg (1999)
17. Kocher, P.C.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 104–113. Springer, Heidelberg (1996)
18. Longa, P., Miri, A.: New composite operations and precomputation scheme for elliptic curve cryptosystems over prime fields. In: Cramer, R. (ed.) PKC 2008. LNCS, vol. 4939, pp. 229–247. Springer, Heidelberg (2008)
19. Mangard, S., Oswald, E., Popp, T.: Power Analysis Attacks: Revealing the Secrets of Smart Cards. Springer, Heidelberg (2007)
20. Meloni, N.: New point addition formulæ for ECC applications. In: Carlet, C., Sunar, B. (eds.) WAIFI 2007. LNCS, vol. 4547, pp. 189–201. Springer, Heidelberg (2007)
21. Möller, B.: Securing elliptic curve point multiplication against side-channel attacks. In: Davida, G.I., Frankel, Y. (eds.) ISC 2001. LNCS, vol. 2200, pp. 324–334. Springer, Heidelberg (2001)
22. Montgomery, P.L.: Speeding the Pollard and elliptic curve methods of factorization. Mathematics of Computation 48(177), 243–264 (1987)
23. Okeya, K., Takagi, T.: The width-w NAF method provides small memory and fast elliptic scalar multiplications secure against side channel attacks. In: Joye, M. (ed.) CT-RSA 2003. LNCS, vol. 2612, pp. 328–342. Springer, Heidelberg (2003)
24. Quisquater, J.-J., Samyde, D.: Electromagnetic analysis (EMA): Measures and counter-measures for smart cards. In: Attali, S., Jensen, T. (eds.) E-smart 2001. LNCS, vol. 2140, pp. 200–210. Springer, Heidelberg (2001)
25. Rivest, R.L., Shamir, A., Adleman, L.M.: A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM 21(2), 120–126 (1978)
26. Sakai, Y., Sakurai, K.: A new attack with side channel leakage during exponent recoding computations. In: Joye, M., Quisquater, J.-J. (eds.) CHES 2004. LNCS, vol. 3156, pp. 298–311. Springer, Heidelberg (2004)
27. Thériault, N.: SPA resistant left-to-right integer recodings. In: Preneel, B., Tavares, S. (eds.) SAC 2005. LNCS, vol. 3897, pp. 345–358. Springer, Heidelberg (2006)
28. Vuillaume, C., Okeya, K.: Flexible exponentiation with resistance to side channel attacks. In: Zhou, J., Yung, M., Bao, F. (eds.) ACNS 2006. LNCS, vol. 3989, pp. 268–283. Springer, Heidelberg (2006)
29. Yao, A.C.: On the evaluation of powers. SIAM Journal on Computing 5(1), 100–103 (1976)
30. Yen, S.-M., Joye, M.: Checking before output may not be enough against fault-based cryptanalysis. IEEE Transactions on Computers 49(9), 967–970 (2000)
31. Yen, S.-M., Kim, S.-J., Lim, S.-G., Moon, S.-J.: A countermeasure against one physical cryptanalysis may benefit another attack. In: Kim, K.-c. (ed.) ICISC 2001. LNCS, vol. 2288, pp. 414–427. Springer, Heidelberg (2002)

A Regular Aggregation

In the general description of the regular right-to-left exponentiation algorithm (i.e., Algorithm 5), the aggregation step consists in evaluating the product $\mathsf{A}^{d_{\ell-1}-1} \cdot \prod_{i=1}^{m} \mathsf{R}[i]^{m+i-2}$. When multiplication by $1_{\mathbb{G}}$ can be distinguished through SPA, the initialization of $\mathsf{A} \leftarrow \mathsf{A}^{d_{\ell-1}-1}$ is not sufficient to prevent the leakage of $d_{\ell-1}$. We present here an alternative method for evaluating $\mathsf{A}^{d_{\ell-1}-1} \cdot \prod_{i=1}^{m} \mathsf{R}[i]^{m+i-2}$ in such a case. For concreteness, we detail it for the ternary case (i.e., $m = 3$) but it can easily be extented to other radices. The case $m = 2$ is treated in § 2.2.

For $m = 3$, the aggregation step becomes $\mathsf{A}^{d_{\ell-1}-1} \cdot \prod_{i=1}^{m} \mathsf{R}[i]^{m+i-2}$ with $d_{\ell-1} \in \{1, 2\}$. To ease the presentation, we let $\mathsf{R}[0]$ denote the accumulator. So, we need to evaluate

$$\begin{cases} \mathsf{R}[0] \leftarrow \mathsf{R}[1]^2 \cdot \mathsf{R}[2]^3 \cdot \mathsf{R}[3]^4 & \text{if } d_{\ell-1} = 1 \\ \mathsf{R}[0] \leftarrow \mathsf{R}[0] \cdot \mathsf{R}[1]^2 \cdot \mathsf{R}[2]^3 \cdot \mathsf{R}[3]^4 & \text{if } d_{\ell-1} = 2 \end{cases}.$$

The idea is to rewrite the product so that the different cases appear as a same series of squarings and multiplications. For example, we can write

$$\begin{cases} \mathsf{B} \leftarrow \mathsf{R}[1]^2 \text{ and } \mathsf{R}[0] \leftarrow (\mathsf{B} \cdot \mathsf{R}[2]) \cdot (\mathsf{R}[3] \cdot \mathsf{R}[2] \cdot \mathsf{R}[3])^2 \\ \mathsf{B} \leftarrow \mathsf{R}[3]^2 \text{ and } \mathsf{R}[0] \leftarrow (\mathsf{R}[0] \cdot \mathsf{R}[2]) \cdot (\mathsf{R}[1] \cdot \mathsf{R}[2] \cdot \mathsf{B})^2 \end{cases}$$

respectively. Moreover, in order not to introduce an additional temporary variable (B in the above description), we make use of $\mathsf{R}[1]$ and $\mathsf{R}[3]$, respectively. We have:

$$\begin{array}{l} d \leftarrow d_{\ell-1} - 1 \\ \mathsf{R}[1 + 2d] \leftarrow \mathsf{R}[1 + 2d]^2 \\ \mathsf{R}[0] \leftarrow \mathsf{R}[2] \cdot \mathsf{R}[1 - d] \\ \mathsf{R}[2] \leftarrow \mathsf{R}[2] \cdot \mathsf{R}[3 - 2d];\ \mathsf{R}[2] \leftarrow \mathsf{R}[2] \cdot \mathsf{R}[3];\ \mathsf{R}[2] \leftarrow \mathsf{R}[2]^2 \\ \mathsf{R}[0] \leftarrow \mathsf{R}[0] \cdot \mathsf{R}[2] \end{array}$$

There are many possible variants of this methodology; the proposed implementation can be modified to better suit a given architecture.