Efficient Modular Exponentiation Based on Multiple Multiplications by a Common Operand
Christophe Negre, Thomas Plantard, Jean-Marc Robert

To cite this version:

HAL Id: lirmm-01142327
https://hal-lirmm.ccsd.cnrs.fr/lirmm-01142327
Submitted on 15 Apr 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Efficient Modular Exponentiation Based on Multiple Multiplications by a Common Operand

Christophe Negre*, Thomas Plantard†, Jean-Marc Robert‡
* Team DALI (UPVD) and LIRMM (UM2, CNRS), France
† CCISR, SCIT, (University of Wollongong), Australia

Abstract—The main operation in RSA encryption/decryption is the modular exponentiation, which involves a long sequence of modular squarings and multiplications. In this paper, we propose to improve modular multiplications $AB, AC$ which have a common operand. To reach this goal we modify the Montgomery modular multiplication in order to share common computations in $AB$ and $AC$. We extend this idea to reduce the cost of multiple modular multiplications $AB_1,\ldots, AB_\ell$ by the same operand $A$. We then take advantage of these improvements in the Montgomery-ladder and SPA resistant $m$-ary exponentiation algorithms. The complexity analysis shows that for an RSA modulus of size 2048 bits, the proposed improvements reduce the number of word operations (ADD and MUL) by 14% for the Montgomery-ladder and by 5%-8% for the $m$-ary exponentiations. Our implementations show a speed-up by 8%-14% for the Montgomery-ladder and by 1%-8% for the $m$-ary exponentiations for modulus of size 1024, 2048 and 4048 bits.

Keywords—Montgomery multiplication, word level, common operand, modular exponentiation.

I. INTRODUCTION

RSA [7] is the most widely used public key cryptosystem. It is for example used to secure credit card transaction and to generate SSL/TLS certificates with an RSA signature. The main operation in RSA protocols is the modular exponentiation $G^e \mod N$. In order to ensure a sufficient security level, the modulus $N$ is typically of size 1000-4000 bits. The most widely used method to compute a modular exponentiation is the square-and-multiply approach, which consists of a long sequence of a few thousands of squarings and multiplications modulo $N$.

In practice, such exponentiation methods can be threatened by side channel analysis. For example, by measuring several computation times of a modular exponentiation and exploiting the difference of squaring and multiplication times, an attacker can recover part of the key (cf. [3]). An attacker can also monitor the power consumption and then read the sequence of operations on the power trace and recover the key bits: this is the Simple Power Analysis (SPA) [4]. In this paper, we focus on the variants of the square-and-multiply robust against timing and SPA attacks, by ensuring a regularity of the sequence of multiplications and squarings. Specifically, the considered methods are the Montgomery-ladder [2] and the regular $m$-ary exponentiations [1].

Let us assume that $N$ is stored in $n$ computer words. A multiplication modulo $N$ consists of a multi-precision integer multiplication followed by a reduction modulo $N$. The modulus $N$ used in RSA cryptosystem has a random form. Consequently, the reduction modulo $N$ is generally performed using the Montgomery approach [6]. This reduction is thus quite costly since an integer multiplication requires $\approx 3n^2$ word operations (word additions or multiplications) while the reduction also requires $\approx 3n^2$ word operations.

In this paper, we present a modified version of the Montgomery multiplication which performs two multiplications $AB, AC$ with a common operand $A$. We reduce the complexity by sharing part of the computations involved in the costly reductions modulo $N$ in $AB$ and $AC$. This reduces the overall complexity of $AB, AC$ by 25%. We extend this idea to improve multiple multiplications $AB_1,\ldots, AB_\ell$ by a common operand $A$: some redundant operations can be avoided by storing some data in memory and by subsequently reusing them in the multiplications $AB_i, i=1,\ldots, \ell$. For sufficiently large $\ell$, this saves $\approx 50\%$ of word operations per multiplication.

The remaining of the paper is organized as follows. Section II is a review the word level version of the Montgomery modular multiplication. In Section III, we present an algorithm which performs two Montgomery multiplications with a common operand $AB, AC$, and we extend this approach to reduce the cost of multiple multiplications by a common operand $A$. In Section IV, a modified version of the Montgomery-ladder and $m$-ary exponentiations are presented, which perform part of the modular multiplications using the proposed improvement on multiplications with a common operand. Section V gives a comparison of the complexity of the proposed approaches with their conventional counterparts and provide implementation results. In Section VI, we give a few concluding remarks.

II. REVIEW OF MONTGOMERY MODULAR MULTIPLICATION

We consider an RSA modulus $N$ such that $N < 2^{wn}$. Let $A$ and $B$ be two integers in $[0, N]$. The classical way to compute $Y = A \cdot B \mod N$ is to perform an Euclidean
division of \( C = A \times B \) by \( N \)
\[
Q \leftarrow \left\lfloor (A \times B)/N \right\rfloor \text{ and } Y \leftarrow A \times B - QN,
\]
and \( Y \) satisfies \( Y < N \). In other words, the most significant bits of \( C \) are cleared by subtracting \( QN \). Montgomery in [6] performs this reduction in a different way: instead of clearing the most significant bits of \( C = A \times B \), one clears the least significant bits, by first computing
\[
Q \leftarrow A \times B \cdot (-N^{-1}) \mod 2^{wn},
\]
and then computing the following exact division
\[
Y \leftarrow (A \times B + Q \cdot N)/2^{wn}.
\]
This produces \( Y \) which satisfies \( Y = A \times B \times 2^{-wn} \mod N \) and \( Y < 2N \).

When one needs to compute a long sequence of modular multiplications, the elements modulo \( N \) are generally set in the so-called Montgomery representation \( \tilde{A} = A \cdot 2^{wn} \mod N \). Indeed, a Montgomery multiplication computes
\[
\tilde{A} \cdot \tilde{B} \cdot 2^{-wn} \mod N = (AB) \cdot 2^{wn} \mod N
\]
which is the Montgomery representation of the product.

We now review the word level variant of the Montgomery representation which can be found in [5]. We first review the two basic operations involved in this algorithm which are the word-multiplication and the small reduction.

**Word-multiplication:** We first study the operation \( a \cdot B \) where \( a \) is a \( w \)-bit integer and \( B = (b_{n-1}, \ldots, b_0)_{2w} \) is an \( nw \)-bit integer. This operation will be the building block of most of the algorithms presented in this paper. The basic method to compute \( a \cdot B \) consists in expanding the product relatively to \( B = \sum_{j=0}^{n-1} b_j 2^{wj} \):
\[
a \cdot B = \sum_{j=0}^{n-1} ab_j 2^{wj}.
\]
Then, the products \( ab_j \) for \( j = 0, \ldots, n-1 \) are sequentially computed and added to the intermediate result \( X \) as shown Algorithm 1.

**Algorithm 1 Word-multiplication**

**Require:** \( a, B \) with \( B = (b_{n-1}, \ldots, b_0)_{2w} \) and \( 0 \leq a, b_i < 2^w \)

**Ensure:** \( X = (x_{n-1}, \ldots, x_0)_{2w} \) with \( X = a \cdot B < 2^{(n+1)w} \)

\( \{ v, u \} \leftarrow a \cdot b_0 \) \( \text{//u is the lower and v the upper part} \)
\( x_0 \leftarrow u, x_1 \leftarrow v \)

\( \text{for } j = 1 \text{ to } n-1 \) do
\( \{ v, u \} \leftarrow a \cdot b_j \) \( \text{//u is the lower and v the upper part} \)
\( x_i \leftarrow \text{ADDC}(x_i, u) \) \( \text{// addition with carry} \)
\( x_{i+1} \leftarrow v \)
\( x_n \leftarrow \text{ADDC}(x_n, 0) \) \( \text{// absorption of the last carry} \)

return \( X = (x_{n-1}, \ldots, x_0)_{2w} \).

The above Word-multiplication algorithm requires \( n \) word multiplications (MUL) and \( n \) word additions (ADD).

**Small Montgomery reduction:** The small reduction is a version of the Montgomery modular reduction which reduces an integer \( A \) modulo \( N \) by \( w \) bits. This method is depicted in Algorithm 2.

**Algorithm 2 SmallRed**

**Require:** A modulus \( N < 2^{wn-2} \) and a positive integer \( X = (x_{n-1}, \ldots, x_0)_{2w} \) of \( n \) words and \( N' = (-N^{-1}) \mod 2^w \)

**Ensure:** \( Y = X \cdot 2^{-w} \mod N \) with \( Y < X/2^w + N \)
\( 1: q \leftarrow x_0 \cdot N' \mod 2^w \)
\( 2: Y \leftarrow (X + q \cdot N)/2^w \)
\( 3: \text{return } Y \)

The complexity of SmallRed with an \( n' \) word input \( X \) is \( \max(n', n+1) + n \) ADD and \( n+1 \) MUL. The following lemma establishes some basic facts concerning the SmallRed algorithm which will be useful in the sequel.

**Lemma 1.** Let \( X \geq 0 \) be the input of Algorithm 2. Then, the output \( Y \) satisfies

i) \( Y < X/2^w + N \) and \( Y = X2^{-w} \mod N \).

ii) If \( X < 2N \) then \( Y < 2N \).

**Proof:**

i) By construction, one has \( X + q \cdot N) = 0 \mod 2^w \). Thus the division by \( 2^w \) (Step 2 of Algorithm 2) is exact and \( Y \) is well defined. This implies that \( Y2^w = (X + q \cdot N) \equiv X \mod N \) and then \( Y \equiv X \cdot 2^{-w} \mod N \).

It is also straightforward to get \( Y = (X + qN)/2^w \leq X/2^w + N \) since \( q < 2^w \).

ii) In the special case \( X < 2N \), one has \( Y = (X + qN)/2^w \leq 2N/2^w + N < 2N \).

**Word level Montgomery multiplication:** We review the word level form of a Montgomery multiplication. Let us consider two integers \( A \) and \( B \) of size \( < N \). The idea is to interleave Word-multiplications (Algo. 1) with SmallReds to maintain the intermediate product \( Y \) in the range \([0, 2N]\) at the end of each loop turn. A final subtraction by \( N \) produces \( Y < N \). This method is detailed in Algorithm 3.

The complexity of Algorithm 3 is evaluated step by step in Table I by using the complexity of Word-multiplication.

A word level Montgomery squaring can be computed more efficiently than a Montgomery multiplication. Indeed, some of the products \( a_i a_j \) are redundant and thus can be avoided:

\[
A^2 = \sum_{i=0}^{n-1} a_i 2^{w(2i)}(a_i + 2 \sum_{j=1}^{n-i-1} a_{i+j} 2^{wj})
\]

\( = \sum_{i=0}^{n-1} a_i 2^{w(2i)} \tilde{A}_i. \) (1)

The integers \( \tilde{A}_i = (a_i + 2 \sum_{j=1}^{n-i-1} a_{i+j} 2^{wj}) \) can be deduced
Algorithm 3 Word level Montgomery multiplication [5]

Require: $N < 2^{w(n-2)}$ the modulus, $w$ the word size, $A = (a_{n-1}, \ldots, a_0)_{2^w}$ and $B = (b_{n-1}, \ldots, b_0)_{2^w}$ two $n$-word integers in $[0, N]$ and $N' = (-N - 1) \mod 2^w$

Ensure: $Y = A \cdot B \mod 2^{w(n-1)}$

\begin{algorithm}
1: $Y \leftarrow a_0 \cdot B$
2: $q \leftarrow |Y|_{2^w} \cdot N' \mod 2^w$
3: $Y \leftarrow (Y + q \cdot N)/2^w$
4: for $i = 1$ to $n - 1$
5: \hspace{5mm} $Y \leftarrow Y + a_i \cdot B$
6: \hspace{5mm} $q \leftarrow |Y|_{2^w} \cdot N' \mod 2^w$
7: \hspace{5mm} $Y \leftarrow (Y + q \cdot N)/2^w$
8: if $Y > N$
9: \hspace{5mm} $Y \leftarrow Y - N$
10: return $Y$
\end{algorithm}

Table I

<table>
<thead>
<tr>
<th>Step</th>
<th>Operations</th>
<th># ADD</th>
<th># MUL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$a_0 \times B$</td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td>2</td>
<td>$</td>
<td>Y</td>
<td>_{2^w} \cdot N'$</td>
</tr>
<tr>
<td>3</td>
<td>$Y + (q \cdot N)/2^w$</td>
<td>$(n-1)n$</td>
<td>$(n-1)n$</td>
</tr>
<tr>
<td>4</td>
<td>$(a_i \times B) \cdot Y + (a_i \cdot B) / 2^w$</td>
<td>$(n-1)(n+1)$</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>$(</td>
<td>Y</td>
<td>_{2^w} \cdot N') y (Y + a_i \cdot B) / 2^w$</td>
</tr>
<tr>
<td>6</td>
<td>$(Y + (q \cdot N)/2^w$</td>
<td>$n$</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>$Y - N$</td>
<td>$n$</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>$4n^2 + 2n -1$</td>
<td>$n(2n+1)$</td>
</tr>
</tbody>
</table>

from $A' = 2A = (a'_{n-1}, \ldots, a'_0)_{2^w}$ as

\[
\tilde{A}_i = (a'_{n-1}, \ldots, a'_i+2, 2a_{i+1} \mod 2^w, a_i)_{2^w}.
\]

With the formulation (1), we derive the word level Montgomery squaring shown in Algorithm 4.

The overall complexity of Algorithm 4 is obtained by adding the contribution of each step: thus, one has $3n^2 + 5n - 1$ ADD and $3n^2 + 5n - 1$ MUL.

Table II gathers all the complexities of the algorithms reviewed in this section: SmallRed, Montgomery multiplication (Algorithm 3) and Montgomery squaring (Algorithm 4).

Table II

<table>
<thead>
<tr>
<th>Operation</th>
<th># ADD</th>
<th># MUL</th>
</tr>
</thead>
<tbody>
<tr>
<td>SmallRed (Alg. 2) with an $n'$-word input</td>
<td>$\max(n', n+1) + n$</td>
<td>$n + 1$</td>
</tr>
<tr>
<td>MontSqu (Alg. 4)</td>
<td>$3n^2 + 5n - 1$</td>
<td>$3n^2 + 5n - 1$</td>
</tr>
<tr>
<td>MontMul (Alg. 3)</td>
<td>$4n^2 + 2n - 1$</td>
<td>$2n^2 + n$</td>
</tr>
</tbody>
</table>

III. IMPROVED MONTGOMERY MULTIPLICATIONS WITH A COMMON OPERAND

In this section, we present some improvements of multiple Montgomery multiplications with a common operand. We first deal with the case $A \times B, A \times C$ and then generalize this result to $A \times B_i, i = 1, \ldots, \ell$.

A. Improved combined multiplications $A \times B, A \times C$

We present in this section an algorithm which takes as input $A, B, C$ and $N$, and outputs $ABR^{-1} \mod N$ and $ACR^{-1} \mod N$ with $R = 2^{w(n+1)}$. Our goal is to share some common computations performed in the Montgomery products $ABR^{-1} \mod N$ and $ACR^{-1} \mod N$. To reach this goal, the product $ABR^{-1}$ relatively to $B$ are expanded as follows:

\[
ABR^{-1} = (\sum_{j=0}^{n-1} b_j 2^{w(j)} \cdot A \cdot 2^{-w(n+1)} \mod N
\]

\[
= \sum_{j=0}^{n-1} b_j A \cdot 2^{-w(n+1-j)} \mod N
\]

\[
= \sum_{j=0}^{n-1} b_j (A \cdot 2^{-w(n+1-j)} \mod N) 2^{-2w} \mod N
\]

\[
= (\sum_{j=0}^{n-1} b_j A(j)) \cdot 2^{-2w} \mod N
\]

where $A(j) = 2^{-w(n+1-j)} \mod N$ for $j = 0, \ldots, n-1$.

With the same for $A \times C \times R^{-1}$, one obtains

\[
A \times C \times R^{-1} = (\sum_{j=0}^{n-1} A(j)) \cdot 2^{-2w} \mod N
\]
We notice that the expression in (2) for $ABR^{-1}$ and the expression in (3) for $ACR^{-1}$ contain the terms $A^{(j)} = A^{-w(n-1-j)} \mod N, j = 0, 1, \ldots, n-1$. To compute $A^{(j)}$, we start from $A^{(n-1)} = A$ and we apply a sequence of $n-1$ SmallReds as follows:

\[
\begin{align*}
\text{SmallRed}(A^{(n-1)}) &= A \cdot 2^{-w} \mod N \\
\text{SmallRed}(A^{(n-2)}) &= (A \cdot 2^{-w}) \cdot 2^{-w} \mod N \\
&= A^{(n-3)}, \\
&\vdots \\
\text{SmallRed}(A^{(1)}) &= (A \cdot 2^{-w(n-2)}) \cdot 2^{-w} \mod N \\
&= A \cdot 2^{-w(n-1)} = A^{(0)}.
\end{align*}
\]

Therefore, one computes $AB$ and $AC$ as follows: the terms $A^{(j)}$ are computed and then $b_jA^{(j)}$ and $c_jA^{(j)}$ are accumulated in $Y$ and $Z$ for $j = 0, \ldots, n-1$. From Lemma 1, if $A < 2N$, then all the values $A^{(j)}$, $j = 0, \ldots, n-1$, satisfy $A^{(j)} < 2N$. The resulting accumulation $\sum_{j=0}^{n-1} b_jA^{(j)} < 2N_2$ in $Y$ is reduced to a value $< 2N$ by performing two consecutive SmallReds. At the same time, these two SmallReds produce the missing factor 2$^{-2w}$ in (2). The same is done for $Z$ to get (3). This approach is depicted in Algorithm 5. Its complexity is evaluated step by step in Table III.

**Algorithm 5 CombinedMontMul($A, B, C$)**

Require: the modulus $N < 2^{w(n-2)}$, three integers $A = (a_{n-1}, \ldots, a_0), B = (b_{n-1}, \ldots, b_0), C = (c_{n-1}, \ldots, c_0)$ such that $A, B, C < 2N$, $w$ the word size, $R = 2^{w(n+1)}$ the Montgomery constant.

Ensure: $Y = A \cdot B \cdot R^{-1} \mod N$ and $Z = A \cdot C \cdot R^{-1} \mod N$.

\[
\begin{align*}
1: & \quad X \leftarrow A \\
2: & \quad Y \leftarrow b_{n-1} \cdot X, Z \leftarrow c_{n-1} \cdot X \\
3: & \quad \text{for } j = n-2 \text{ downto } 0 \text{ do} \\
4: & \quad q \leftarrow \lfloor X \rfloor \mod 2^w \\
5: & \quad X \leftarrow (X + q \cdot N) \mod 2^w \quad // A^{(j)} \text{ for } j = n-2, \ldots, 0 \\
6: & \quad Y \leftarrow Y + b_j \cdot X, Z \leftarrow Z + c_j \cdot X \\
7: & \quad Y \leftarrow \text{SmallRed}(Y), Z \leftarrow \text{SmallRed}(Z) \\
8: & \quad Y \leftarrow \text{SmallRed}(Y), Z \leftarrow \text{SmallRed}(Z) \\
9: & \quad \text{return } Y \text{ and } Z
\end{align*}
\]

**B. Multiple multiplications with a common operand**

In this subsection, we extend the idea used in the CombinedMontMul algorithm to multiple multiplications by a common operand. Specifically, given a fixed element $A \in \{0, \ldots, N-1\}$ and a sequence of $B_i, i = 1, \ldots, \ell$, we want to compute:

\[
\begin{align*}
Y_1 &= A \cdot B_1 \cdot 2^{-w(n+1)} \mod N, \\
Y_2 &= A \cdot B_2 \cdot 2^{-w(n+1)} \mod N, \\
&\vdots \\
Y_\ell &= A \cdot B_\ell \cdot 2^{-w(n+1)} \mod N.
\end{align*}
\]

As in Subsection III-A, we expand each multiplication $A \cdot B_i \cdot 2^{-w(n+1)} \mod N$ relatively to $B_i$ and rewrite the product as follows:

\[
A \cdot B_i \cdot 2^{-w(n+1)} = \left( \sum_{j=0}^{n-1} b_{ij} \cdot 2^{wj} \right) A^{-w(n+1)} \mod N.
\]

One may notice that the above expression of $A \cdot B_i \cdot 2^{-w(n+1)}$ contains $A^{(j)} = A^{-w(n-1-j)} \mod N$ for $j = 0, \ldots, n-1$. We propose to precompute these $n$ terms $A^{(j)}$ and store them in memory. They are computed through a sequence of $n-1$ SmallReds as shown Algorithm 6.

**Algorithm 6 PrecompMultByComOp($A$)**

Require: $A, N \in \mathbb{Z}$ with $A < 2N$ and $N < 2^{n-2}$, a precomputed value $N' = (-N^{-1}) \mod 2^w$.

Ensure: $A^{(0)}, \ldots, A^{(n-1)}$ such as $A^{(j)} = 2^{-w(n-1-j)} \cdot A \mod N$ and $A^{(j)} < 2N$.

\[
\begin{align*}
1: & \quad A^{(n-1)} \leftarrow A \mod N \\
2: & \quad \text{for } j = n-2 \text{ downto } 0 \text{ do} \\
3: & \quad A^{(j)} \leftarrow \text{SmallRed}(A^{(j+1)}) \\
4: & \quad \text{return } A^{(0)}, \ldots, A^{(n-1)}
\end{align*}
\]

**Complexity of Algorithm 6.** This algorithm consists of $n-1$ SmallReds, all with an $n$-word input. Using the complexity of SmallRed shown in Table II, we obtain the complexity of Algorithm 6, which is $(2n+1)(n-1) \text{ ADD and } (n^2-1) \text{ MUL.}
Now, with the $n$ precomputed terms $A^{(0)}, \ldots, A^{(n-1)}$, one can compute any of the products $A \cdot B_i \cdot 2^{-w(n+1)} \mod N$ for $i = 1, \ldots, \ell$ using (4). Indeed, one computes $Y = \sum_{j=0}^{n-1} b_{i,j} A^{(j)}$ through the sequence $Y \leftarrow Y + b_{i,j} A^{(j)}$ for $j = 0, \ldots, n-1$, and $Y$ satisfies $Y = A \cdot B_i \cdot 2^{-w(n+1)}$ and $Y < 2N n^{2w}$. One needs two final SmallReds in order to reduce $Y$ to a value $< 2N$ and also to multiply $Y$ by $2^{-2w}$ resulting in $Y = AB_i 2^{-w(n+1)} \mod N$, as required. In Algorithm 7, this strategy is used to compute $A \cdot B \cdot 2^{-w(n+1)} \mod N$ for an arbitrary input $B$ in $[0, 2N]$.

**Algorithm 7 MultByComOp $(B, A^{(0)}, \ldots, A^{(n-1)})$**

**Require:** the modulus $N < 2^{w(n-2)}$, an integer $B = (b_{i-1}, \ldots, b_0, 0)_{2w}$ such that $B < 2N$ and $A^{(j)} = 2^{-w(n-1-j)} \cdot A \mod N$ for $j = 0, \ldots, n-1$

**Ensure:** $Y = A \cdot B \cdot 2^{-w(n+1)} \mod N$ with $Y < 2N$

1. $Y \leftarrow b_0 \cdot A^{(0)}$
2. for $j = 1$ to $n-1$ do
3. $Y \leftarrow Y + b_j \cdot A^{(j)}$
4. $Y \leftarrow$ SmallRed($Y$)
5. $Y \leftarrow$ SmallRed($Y$)
6. return $Y$

The complexity of Algorithm 7 is evaluated step by step in Table IV.

### Table IV

**Complexity of Algorithm 7**

<table>
<thead>
<tr>
<th>Operations</th>
<th># ADD</th>
<th># MUL</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b_0 \cdot A^{(0)}$</td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td>$b_j \cdot A^{(j)}$, $Y + (b_2 \cdot A^{(j)})$</td>
<td>$(n-1)n$</td>
<td>$(n-1)n$</td>
</tr>
<tr>
<td>$Y = Y + b_{n-2} \cdot A^{(j)}$</td>
<td>$(n-1)n + 2$</td>
<td>$0$</td>
</tr>
<tr>
<td>SmallRed with $n = n + 2$</td>
<td>$2n + 2$</td>
<td>$n + 1$</td>
</tr>
<tr>
<td>SmallRed with $n = n + 1$</td>
<td>$2n + 1$</td>
<td>$n + 1$</td>
</tr>
<tr>
<td>Total</td>
<td>$2n^2 + 5n + 1$</td>
<td>$n^2 + 2n + 2$</td>
</tr>
</tbody>
</table>

The computation of $\ell$ multiplications $A \cdot B_i \cdot 2^{-w(n+1)} \mod N$ for $i = 1, \ldots, \ell$ consists of one execution of Algorithm 6 and $\ell$ executions of Algorithm 7. The cost for $\ell$ multiplications using this strategy is then $\ell(2n^2 + 5n + 1 + (2n + 1)(n - 1))$ ADD and $\ell(n^2 + 2n + 2 + (n^2 - 1))$ MUL.

**C. Complexity comparison**

In Table V, we report the complexity of the proposed CombinedMontMul and MultiByComOp approaches. We also provide the complexities of the usual approaches which perform two independent MontMuls for $AB, AC$ and one MontMul and one MontSqu for $AB, A^2$. For $\ell$ multiplications by a common operand, we provide the complexity of $\ell$ MontMuls.

### Table V

**Complexity Comparison for $AB, AC$ and $AB, A^2$ and Multiple Multiplications by $A$**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Algorithm</th>
<th># ADD</th>
<th># MUL</th>
</tr>
</thead>
<tbody>
<tr>
<td>$AB, AC$</td>
<td>Two MontMuls</td>
<td>$8n^2 + 4n - 2$</td>
<td>$4n^2 + 2n$</td>
</tr>
<tr>
<td>$AB, A^2$</td>
<td>MontMul and MontSqu</td>
<td>$7n^2 + 7n - 2$</td>
<td>$\frac{7}{2}n^2 + \frac{7}{2}n - 1$</td>
</tr>
<tr>
<td>$AB, AC$</td>
<td>CombinedMontMul</td>
<td>$6n^2 + 9n + 1$</td>
<td>$3n^2 + 4n + 3$</td>
</tr>
<tr>
<td>$AB_i, i = 1\ldots, \ell$</td>
<td>$\ell$ MultMonts</td>
<td>$\ell(4n^2 + 2n - 1)$</td>
<td>$\ell(2n^2 + n)$</td>
</tr>
<tr>
<td>$AB_i, i = 1\ldots, \ell$</td>
<td>$\ell$ MultiByComOp and one PrecomMulByComOp</td>
<td>$\ell(2n^2 + 5n + 1)$</td>
<td>$\ell(n^2 + 2n + 2)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$+(2n + 1)(n - 1)$</td>
<td>$+(n^2 - 1)$</td>
</tr>
</tbody>
</table>

One can check that our proposed CombinedMontMul for $AB, AC$ is better by 25% than the classical approach of two independent MontMuls. For $AB, A^2$, the CombinedMontMul approach reduces the complexity by 13% compared to MontMul and MontSqu.

For $\ell$ multiplications $AB_1, \ldots, AB_\ell$, we notice that when $\ell \geq 2$, the proposed approach is more efficient by 25% compared to $\ell$ independent MontMuls. When $\ell \geq n$ and when $n$ is large enough, the complexity of our approach tends to be 50% less than the one of $\ell$ independent MontMuls.

**IV. Exponentiation with Improved Multiplications by a Common Operand**

In this section, we consider modular exponentiation algorithms which are robust against SPA and timing attacks: Montgomery-ladder [2] and regular $m$-ary exponentiation [1]. We use the approaches presented in the previous section to improve these exponentiations modulo $N$. Specifically, we show that the Montgomery-ladder and the right-to-left $m$-ary exponentiation can take advantage of the CombinedMontMul algorithm and that the left-to-right exponentiation can take advantage of the MultiByComOp algorithm.

**A. Montgomery-ladder with CombinedMontMul**

The Montgomery-ladder (cf. [2]) computes an exponentiation $G^k \mod N$ through a sequence of operations of the form $A \cdot B, A \cdot A$. These operations can be performed using the CombinedMontMul algorithm since CombinedMontMul($A, B, A$) returns $A \cdot B$ and $A \cdot A$. Based on the complexities reported in Table V, the use of CombinedMontMul is more efficient than the classical approach based on MontMul and MontSqu. This will reduce the complexity of the Montgomery-ladder. The resulting modified Montgomery-ladder is given Algorithm 8.

**Complexity of Algorithm 8.** The operations performed in Algorithm 8 consist of two Montgomery multiplications, two SmallReds and $k$ CombinedMontMul (in the for loop).
Using the complexities of Table V, one has:

\[
\begin{align*}
\#\text{ADD} & = k(6n^2 + 9n + 1) + (8n^2 + 8n), \\
\#\text{MUL} & = k(3n^2 + 4n + 3) + (4n^2 + 4n + 2).
\end{align*}
\]

B. Right-to-left \(2^t\)-ary exponentiation with CombinedMontMul

We first review the right-to-left \(m\)-ary method for modular exponentiation \(G^e \mod N\). Let us assume that the exponent \(e\) is recoded as \(e = (e_{k-1}, \ldots, e_0)_m\) where \(e_i \in \{1, \ldots, m\}\) and \(m = 2^t\) using the method of [1]. The right-to-left version of the \(m\)-ary exponentiation is based on the following expression of \(G^e \mod N\):

\[
G^e \mod N = \prod_{i=0}^{k-1} G_{m^i} \cdot \prod_{i=0}^{k-1} G_{2m^i} \cdot \cdots \cdot \prod_{i=0}^{k-1} G_{m^{m^i}} \mod N
\]

where \(Y_j = \prod_{i=0}^{k-1} G_{m^i} \mod N\).

Consequently, \(G^e \mod N\) is computed through a sequence \(X_i \leftarrow G^{m^i} \mod N\) followed by \(Y_{e_i} \leftarrow Y_{e_i} \cdot X_i\) for \(i = 0, \ldots, k-1\). At the end, one gets \(Y_j\) for \(j = 1, \ldots, m\).

The final result \(G^e = \prod_{j=1}^m Y_j^j\) is computed as follows with \(2(m-1)\) multiplications:

\[
Z \leftarrow Y_{m-1}
\]

\[
\text{for } i = m-1 \text{ downto } 1 \text{ do}
\]

\[
Y_i \leftarrow Y_i \cdot Y_{i+1} \mod N
\]

\[
Z \leftarrow Z \cdot Y_i \mod N
\]

And at the end, one has \(Z = G^e \mod N\).

When \(m = 2^t\), we can take advantage of the CombinedMontMul algorithm to improve the right-to-left \(m\)-ary method. Indeed, the operations \(Y_{e_i} \leftarrow Y_{e_i} \cdot X_i\) and \(X_{i+1} \leftarrow X^2\) can be performed by first computing \(Y_{e_i}, X_{i+1} \leftarrow \text{CombinedMontMul}(X_i, Y_{e_i}, X_i)\) and afterwards by computing \(t-1\) Montgomery squarings \(X_{i+1} \leftarrow \text{MontSqu}(X_{i+1})\) followed by \(\text{SmallRed}(X_{i+1})\).

The final reconstruction \(Z = \prod_{j=1}^m Y_j^j\) can also take advantage of the CombinedMontMul algorithm. Indeed, it consists of a sequence of two multiplications with a common operand: \(Y_i \cdot Y_{i+1} \mod N\) and \(Z \cdot Y_i \mod N\). This can also be performed using a sequence of \(2^t - 2\) CombinedMontMuls.

The resulting improved right-to-left \(2^t\)-ary exponentiation is shown in Algorithm 9. Its complexity is evaluated step by step in Table VI.

Algorithm 9 Right-to-left regular \(2^t\)-ary exponentiation with CombinedMontMul

Require: \(N < 2^{wn-2}\) the modulus, an integer \(0 \leq G < N\), an exponent \(e = (e_{k-1}, \ldots, e_0)_2\) with \(e_i \in \{1, \ldots, 2^t\}\), \(R = 2^n(n+1)\) the Montgomery constant.

Ensure: \(G^e \mod N\)

\[
\begin{align*}
1: & \text{ // conversion } X_0 \leftarrow G \cdot R^2 \cdot R^{-1} \mod N \\
2: & X_1 \leftarrow \text{MontMul}(G, R^2 \mod N) \\
3: & X_1 \leftarrow \text{SmallRed}(X_1) \\
4: & \text{ for } i = k-1 \text{ downto } 0 \text{ do}
\end{align*}
\]

\[
\begin{align*}
6: & X_{1-e_i} \leftarrow \text{CombinedMontMul}(X_{e_i}, X_{1-e_i}, X_i) \\
7: & \text{ // conversion } X_0 \leftarrow (G \cdot R) \cdot R^{-1} \mod N \\
8: & X_0 \leftarrow \text{MontMul}(X_0, 1), X_0 \leftarrow \text{SmallRed}(X_0) \\
9: & \text{return } X_0
\end{align*}
\]

\textbf{Table VI}

\begin{tabular}{|c|c|c|c|}
\hline
Step & Op & # ADD & # MUL \\
\hline
1 & MM & 4n^2 + 2n - 1 & 2n^2 + n \\
1 & SR & 2n + 1 & n + 1 \\
\hline
2 & 2^t - 1 & MM & (t - 1)k(3n^2 + 5n - 1) + 2n + 1 & (t - 1)k(\frac{25n^2}{2} + \frac{25n}{2} - 1) + 2n + 1 \\
6 & 2^t - 1 & SR & 2n + 1 & n + 1 \\
\hline
3 & 2^t - 2 & MM & (2^t - 2)(3n^2 + 9n + 1) & (2^t - 2)(3n^2 + 4n + 3) \\
8 & 2^t - 2 & SR & 2n + 1 & n + 1 \\
\hline
4 & 4n^2 + 2n - 1 & 2n^2 + n & \\
10 & 4n^2 + 2n - 1 & 2n^2 + n & \\
10 & SR & 2n + 1 & n + 1 \\
\hline
Total & MM & t(k(3n^2 + 7n)) + k(3n^2 + 2n + 1) & tk(\frac{25n^2}{2} + \frac{25n}{2}) + k(\frac{25n^2}{2} + \frac{25n}{2} + 3) \\
Total & SR & 2n + 1 & n + 1 \\
\hline
\end{tabular}

C. MM = CombinedMontMul, MM = MontMul, MS = MontSqu, SR = SmallRed
C. Left-to-right $2^t$-ary exponentiation using MultiByComOp

In this subsection, we present an improved version of the left-to-right $2^t$-ary exponentiation using the multiple multiplications by the same operand (Subsection III-B). The left-to-right exponentiation consists in first precomputing $G_i = G^{i} \mod N, i = 1, \ldots, 2^t$ and then in computing $X = G^n \mod N$ through the sequence of operations $X \leftarrow X^{2^i} \cdot G_{e_i}$ for $i = k - 1, \ldots, 0$.

One can take advantage of the MultiByComOp algorithm as follows:

- At the very beginning, one gets $G(0), \ldots, G(n-1)$ with PrecompMultiByComOp($G$) in order to compute efficiently the $2^t - 1$ multiplications $G_i \leftarrow G \cdot G_{i-1} \mod N$.
- Then for each $G_i$, one gets $G_j^{(i)} = G_i \cdot 2^{-w(n-1-j)} \mod N$ for $j = 0, \ldots, n - 1$ by computing PrecompMultiByComOp($G_i$).
- In the main loop, which sequentially computes $X \leftarrow X^{2^i} \cdot G_{e_i}$ for $i = k - 1, \ldots, 0$, each multiplication by $G_{e_i}$ is done as $X \leftarrow$ MultiByComOp($X, G_{e_i}, \ldots, G_{e_i}^{(i-1)}$).

The resulting improved modular exponentiation is given in Algorithm 10. The reader may notice that in the sequence of squarings for the computation of $X^{2^i}$, each Montgomery squaring or multiplication is followed by a SmallRed, in order to have the correct Montgomery factor $R = 2^{-w(n+1)}$.

**Algorithm 10** Left-to-right regular $2^t$-ary exponentiation with MultiByComOp

**Require:** $N < 2^{w-1}$ the modulus, an integer $0 \leq G < N$, an exponent $e = (e_{k-1}, \ldots, e_0)_2$ with $e_{k-1} \in \{1, \ldots, m\}$, $R = 2^{w(n+1)}$ the Montgomery constant.

**Ensure:** $G^e \mod N$
1. // $G_1 = G \cdot 2^{w(n+1)} \mod N$
2. $G_1 \leftarrow$ MontMul($G, R^2 \mod N$)
3. $G_1 \leftarrow$ SmallRed($G_1$)
4. $X \leftarrow R$
5. $G^{(0)}, \ldots, G^{(n-1)} \leftarrow$ PrecompMultiByComOp($G_1$)
6. for $i = 2$ to $2^t$ do
7. $G_i \leftarrow$ MultiByComOp($G_{i-1}, G^{(0)}_1, \ldots, G^{(n-1)}_1$)
8. $G^{(0)}, \ldots, G^{(n-1)} \leftarrow$ PrecompMultiByComOp($G_i$)
9. for $i = k - 1$ downto 0 do
10. for $j = 1$ to $t$ do
11. // $X \leftarrow X \cdot 2^{-w(n+1)} \mod N$
12. $X \leftarrow$ MontSqu($X$), $X \leftarrow$ SmallRed($X$)
13. $X \leftarrow$ MultiByComOp($X, G^{(0)}_1, \ldots, G^{(n-1)}_1$)
14. // $X \leftarrow X \cdot 2^{-w(n+1)} \mod N$
15. $X \leftarrow$ MontMul($X, 1$), $X \leftarrow$ SmallRed($X$)
16. return $X$

We point out that this left-to-right version requires a large memory, i.e., $\cong n \times 2^t \times nw$ bits, to store the $n \times 2^t$ terms $G^{(j)}$.

V. COMPLEXITY AND IMPLEMENTATION

A. Complexity comparison

In Table VIII, we provide the complexities of the three considered approaches: Montgomery-ladder, right-to-left and left-to-right $2^t$-ary exponentiations. For each exponentiation method, we first give the complexities without any optimization, i.e., using regular Montgomery multiplication and squaring. We then report the complexities of the proposed improved exponentiations, i.e., Algorithm 8, Algorithm 9 and Algorithm 10.

**Table VIII** Complexities of modular exponentiation algorithms

<table>
<thead>
<tr>
<th>Step</th>
<th>Op</th>
<th># ADD</th>
<th># MUL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MM</td>
<td>$4n^2 + 2n - 1$</td>
<td>$2n^2 + n$</td>
</tr>
<tr>
<td>2</td>
<td>SR</td>
<td>$2n + 1$</td>
<td>$n + 1$</td>
</tr>
<tr>
<td>3</td>
<td>PMBCO</td>
<td>$2n^2 - 1$</td>
<td>$n^2 - 1$</td>
</tr>
<tr>
<td>4</td>
<td>(2^t - 1) MBCO</td>
<td>$(2^t - 1)(2n^2 + 5n + 1)$</td>
<td>$(2^t - 1)(n^2 + 2n + 2)$</td>
</tr>
<tr>
<td>5</td>
<td>(2 - 1) MBCO</td>
<td>$(2 - 1)(2n^2 - n - 1)$</td>
<td>$(2 - 1)(n^2 - 1)$</td>
</tr>
<tr>
<td>6</td>
<td>tk</td>
<td>$tk(3n^2 + 5n + 1)$</td>
<td>$tk(3n^2 + 2n - 1)$</td>
</tr>
<tr>
<td>7</td>
<td>MS</td>
<td>$tk(3n^2 + 5n - 1)$</td>
<td>$tk(3n^2 + n - 1)$</td>
</tr>
<tr>
<td>8</td>
<td>SR</td>
<td>$tk(2n + 1)$</td>
<td>$tk(2n + 1)$</td>
</tr>
<tr>
<td>9</td>
<td>MBCO</td>
<td>$k(2n^2 + 5n + 1)$</td>
<td>$k(n^2 + 2n + 2)$</td>
</tr>
<tr>
<td>10</td>
<td>Step 15</td>
<td>$4n^2 + 2n - 1$</td>
<td>$2n^2 + n$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$2n + 1$</td>
<td>$n + 1$</td>
</tr>
</tbody>
</table>

$PMBCO=PrecompMulByComOp, MBCO=MulByComOp, MM=MontMul, MS=MontSqu, SR=SmallRed$

$G^{(j)}$.
the leading of the MUL complexity from $3.5kn^2$ to $3kn^2$. This represents an improvement of 14%.

For the right-to-left $2^i$-ary exponentiation, the leading terms for both ADD and MUL complexities correspond to $kt$ and are due to the sequence of squarings which are roughly the same for right-to-left and right-to-left with CMM. However, concerning the terms in $k$ and $2^t$, a reduction by $\approx 25\%$ is observed.

Finally, for the left-to-right methods, the leading terms in $tk$ and in $2^t$ are roughly the same for left-to-right and left-to-right with MulByComOp. However, a reduction by $\approx 50\%$ for the term in $k$ is observed.

As example, for a modulus $N$ of 2048 bits, which is commonly used in RSA cryptosystems, we provide the explicit complexities of the modular exponentiations in Table IX. The improvement is around 13% for the Montgomery-ladder, 4% for the right-to-left exponentiation and 8% for the left-to-right exponentiation.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>ML 1024 bits</th>
<th>ML 2048 bits</th>
<th>ML 4096 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#ADD/10$^3$</td>
<td>Improv.</td>
<td>#ADD/10$^3$</td>
</tr>
<tr>
<td>ML</td>
<td>15143</td>
<td>13183</td>
<td>4695</td>
</tr>
<tr>
<td>ML with CMM</td>
<td>2793</td>
<td>2694</td>
<td>8595</td>
</tr>
<tr>
<td></td>
<td>5.9%</td>
<td>5.9%</td>
<td>4.1%</td>
</tr>
</tbody>
</table>

B. Software implementation results

The three considered exponentiation algorithms were coded in C, compiled with gcc 4.8.2 and run on an Intel Core i7®-4770 CPU @ 3.4GHz. We used the low level functions performing word-multiplications and additions of the GMP library (GMP 6.0.0, https://gmplib.org) as building blocks of our codes. The timings were obtained by deactivating turbo-boost and hyperthreading options and by averaging the times of a few hundred executions with random inputs. The resulting timings are reported in Table X.

The reported timings show significant improvements for the Montgomery ladder for all considered sizes. For the right-to-left exponentiation, the improvement ratio is low as expected, but close to the theoretical ratio for 4096 bits. Finally, the left-to-right exponentiation with MBCO shows an improved timings only for 2048 and 4096 bits. Thus, the influence of the large memory space (256KB for 2048 bits and 1MB for 4096 bits) seems to be negligible in our tests, probably due to the fact that it is smaller than the L3 cache memory size.

VI. Conclusion

In this paper, we considered modular exponentiations for sizes corresponding to practical implementations of RSA. Multiplications and squarings modulo $N$ are performed with the costly method of Montgomery [6] due to the random form of the modulus $N$. We showed that for multiplications which share a common operand, one can save some redundant computations. Indeed, the proposed approach reduces by 25% the computation of two products $AB$, $AC$. Extending this idea to a large number of multiplications by a common operand $A$, the complexity of each multiplication is improved by $\approx 50\%$. Based on these optimized modular multiplications, we improved modular exponentiations protected against simple power analysis and timing attacks. In particular we improved the Montgomery-ladder by $9-14\%$ compared to usual implementation involving word level Montgomery multiplication and squaring.

Acknowledgement. This work is partly supported by the PAVOIS project (ANR 12 BS02 002 01) and by the Thelxinoë Erasmus Mundus European project.

REFERENCES