Trade-off Approaches for Leak Resistant Modular Arithmetic in RNS

C. Negre[1,2] and G. Perin[3]

[1] Team DALI, Université de Perpignan, France
[2] LIRMM, UMR 5506, Université Montpellier 2 and CNRS, France
[3] Riscure, Netherlands

Abstract On an embedded device, an implementation of cryptographic operation, like an RSA modular exponentiation [12], can be attacked by side channel analysis. In particular, recent improvements on horizontal power analysis [3,10] render ineffective the usual counter-measures which randomize the data at the very beginning of the computations [4,2]. To counteract horizontal analysis it is necessary to randomize the computations all along the exponentiation. The leak resistant arithmetic (LRA) proposed in [1] implements modular arithmetic in residue number system (RNS) and randomizes the computations by randomly changing the RNS bases. We propose in this paper a variant of the LRA in RNS: we propose to change only one or a few moduli of the RNS basis. This reduces the cost of the randomization and makes it possible to be executed at each loop of a modular exponentiation.

Keywords: Leak resistant arithmetic, randomization, modular multiplication, residue number system, RSA.

1 Introduction

Nowadays, the RSA cryptosystem [12] is constantly used in e-commerce and credit card transactions. The main operation in RSA protocols is an exponentiation $x^K \mod N$ where N is a product of two primes $N = pq$. The secret data are the two prime factors of N and the private exponent K used to decrypt or sign a message. The actual recommended size for N is around 2000-4000 bits to insure the intractability of the factorization of N. The basic approach to perform efficiently the modular exponentiation is the square-and-multiply algorithm: it scans the bits k_i of the exponent K and performs a sequence of squarings followed by a multiplication only when k_i is equal to one. Thus the cryptographic operations are quite costly since they involve a few thousands of multiplications or squarings modulo a large integer N.

A cryptographic computation performed on an embedded device can be threaten by side channel analysis. These attacks monitor power consumption or electromagnetic emanation leaked by the device to extract the secret data. The simplest attack is the simple power analysis (SPA) [8] which applies when

the power trace of a modular squaring and a modular multiplication are different. This makes it possible to read the sequence of operations on the power trace of an exponentiation and then derive the key bits of the exponent. This attack is easily overcome by using an exponentiation algorithm like the Montgomery-ladder [6] which render the sequence of operation uncorrelated to the key bits. A more powerful attack, the differential power analysis (DPA) [8], makes this counter-measure against SPA inefficient. Specifically, DPA uses a large number of traces and correlate the intermediate values with the power trace: it then track the intermediate value all along the computation and then guess the bits of the exponent. Coron in [4] has shown that the exponentiation can be protected from DPA by randomizing the exponent and by blinding the integer x. Recently the horizontal attacks presented in [13,3] require only one power trace of an exponentiation, and threaten implementations which are protected against SPA and DPA with the method of Coron [4]. The authors in [3] explains that the best approach to counteract horizontal attack is to randomize the computations all along the exponentiation.

One popular approach to randomize modular arithmetic is the leak-resistant approach presented in [1] based on residue number system (RNS). Indeed, in [1], the authors noticed that the mask induced by Montgomery modular multiplication can be randomized in RNS by permuting the moduli of the RNS bases. In this paper we investigate an alternative method to perform this permutation of bases. Our method changes only one modulus at a time. We provide formula for this kind of randomization along with the required updates of the constants involved in RNS computations. The complexity analysis shows that this approach can be advantageous for a lower level of randomization compared to [1]. In other words this provides a trade-off between efficiency and randomization.

The remainder of the paper is organized as follows. In Section 2 we review modular exponentiation methods and modular arithmetic in RNS. We then recall in Section 3 the leak resistant arithmetic in RNS of [1]. In Sections 4 and Appendix A we present our methods for randomizing the modular arithmetic in RNS. We then conclude the paper in Section 5 by a complexity comparison and some concluding remarks.

2 Review of modular exponentiation in RNS

2.1 Modular exponentiation

The basic operation in RSA protocols is the modular exponentiation: given an RSA modulus N, an exponent K and a message $x \in \{0, 1, \ldots, N-1\}$, a modular exponentiation consists to compute

$$z = x^K \mod N.$$

This exponentiation can be performed efficiently with the square-and-multiply algorithm. This method scans the bits k_i of the exponent $K = (k_{\ell-1}, \ldots, k_0)_2$ from left to right and performs a sequence of squarings followed by multiplications by x if the bit $k_i = 1$ as follows:

$r \leftarrow 1$
for i **from** $\ell - 1$ **downto** 0 **do**
  $r \leftarrow r^2 \mod N$
  if $k_i = 1$ **then**
    $r \leftarrow r \times x \mod N$
  end if
end for

The complexity of this approach is, in average, ℓ squarings and $\ell/2$ multiplications.

Koche *et al.* in [8] showed that the square-and-multiply exponentiation is weak against simple power analysis. Indeed, if a squaring and a multiplication have different power traces, an eavesdropper can read on the trace of a modular exponentiation the exact sequence of squarings and multiplications, and then deduce the corresponding bits of K. It is thus recommended to perform an exponentiation using, for example, the Montgomery-ladder [6] which computes $x^K \mod N$ through a regular sequence of squarings and multiplications. This method is detailed in Algorithm 1. The regularity of the exponentiation prevents an attacker to directly read the key bits on a single trace.

Algorithm 1 Montgomery-ladder [6]

Require: $x \in \{0, \ldots, N-1\}$ and $K = (k_{\ell-1}, \ldots, k_0)_2$
1: $r_0 \leftarrow 1$
2: $r_1 \leftarrow x$
3: **for** i **from** $\ell - 1$ **downto** 0 **do**
4: **if** $k_i = 0$ **then**
5: $r_1 \leftarrow r_1 \times r_0$
6: $r_0 \leftarrow r_0^2$
7: **end if**
8: **if** $k_i = 1$ **then**
9: $r_0 \leftarrow r_0 \times r_1$
10: $r_1 \leftarrow r_1^2$
11: **end if**
12: **end for**
13: **return** (r_0)

Some more sophisticated attacks can threaten a naive implementation of Montgomery-ladder exponentiation. For example differential power analysis [8] makes it necessary to randomize the exponent and blind the integer x by random mask as explained in [4]. Horizontal approaches [13,3] are even more powerful since they require only a single trace to complete the attack and is effective even if the exponent K is masked and the data x is blinded. The authors in [3] propose to counteract horizontal power analysis by randomizing each multiplication and squaring using some temporary mask. In this paper we deal with the problem of randomizing modular multiplications and squarings: we will use the residue

number system (RNS) to represent integers and perform efficiently modular operations.

2.2 Montgomery multiplication in RNS

Let N be a modulus and let x, y be two integers such that $0 \leq x, y < N$. One of the most used methods to perform modular multiplication $x \times y \mod N$ is the method of Montgomery in [9]. This approach avoids Euclidean division as follows: it uses an integer A such that $A > N$ and $\gcd(A, N) = 1$ and computes $z = xyA^{-1} \mod N$ as follows:

$$\begin{array}{l} q \leftarrow -xyN^{-1} \mod A \\ z \leftarrow (xy + qN)/A \end{array} \tag{1}$$

To check the validity of the above method we notice that $(xy+qN) \mod A = 0$, this means that the division by A is exact in the computation of z and then $z = xyA^{-1} \mod N$. The integer z is almost reduced modulo N since $z = (xy + qN)/A < (N^2 + AN)/A < 2N$: if $z > N$, with a single subtraction of N we can have $z < N$. In practice the integer A is often taken as a power of 2 in order to have almost free reduction and division by A.

For a long sequence of multiplications, the use of the so-called Montgomery representation is used

$$\widetilde{x} = (x \times A) \mod N. \tag{2}$$

Indeed, the Montgomery multiplication applied to $\widetilde{x}$ and $\widetilde{y}$ output $\widetilde{z} = xyA \mod N$, i.e., the Montgomery representation of the product of x and y.

Residue number system. In [11] the authors showed that the use of residue number system (RNS) makes it possible to perform Montgomery multiplication efficiently with an alternative choice for A. Let $a_1, \ldots, a_t$ be t coprime integers. In the residue number system an integer x such that $0 \leq x < A = \prod_{i=1} a_i$ is represented by the t residues

$$x_i = x \mod a_i \text{ for } i = 1, \ldots, t.$$

Moreover, x can be recovered from its RNS expression using the Chinese remainder theorem (CRT) as follows

$$x = \left(\sum_{i=1}^{t} \left[x_i \times A_i^{-1} \right]_{a_i} \times A_i \right) \mod A \tag{3}$$

where $A_i = \prod_{j=1, j \neq i}^{t} a_i$ and the brackets $[\,\cdot\,]_{a_i}$ denotes a reduction modulo a_i. The set $\mathcal{A} = \{a_1, \ldots, a_t\}$ is generally called an RNS basis.

Let $x = (x_1, \ldots, x_t)_{\mathcal{A}}$ and $y = (y_1, \ldots, y_t)_{\mathcal{A}}$ be two integers given in an RNS basis $\mathcal{A}$. Then, the CRT provides that an integer addition $x+y$ or multiplication $x \times y$ in RNS consists in t independent additions/multiplications modulo a_i

$$\begin{array}{l} x + y = ([x_1 + y_1]_{a_1}, \ldots, [x_t + y_t]_{a_t}), \\ x \times y = ([x_1 \times y_1]_{a_1}, \ldots, [x_t \times y_t]_{a_t}). \end{array}$$

The main advantage is that these operations can be implemented in parallel since each operation modulo a_i are independent from the others. Only comparisons and Euclidean divisions are not easy to perform in RNS and require partial reconstruction of the integers x and y.

Montgomery multiplication in RNS. In [11] Posch and Posch notice that the Montgomery multiplication can be efficiently implemented in RNS: they use the fact that we can modify the second step of the Montgomery multiplication (1) as

$$z \leftarrow (xy + qN)A^{-1} \mod B$$

where B is an integer coprime with A and N and greater than $2N$. Furthermore, Posch and Posch propose to perform this modified version of the Montgomery multiplication in RNS. Specifically, they choose two RNS bases $\mathcal{A} = (a_1, \ldots, a_t)$ and $\mathcal{B} = (b_1, \ldots, b_t)$ such that $\gcd(a_i, b_j) = 1$ for all i, j. They perform $z = xyA^{-1} \mod N$ as it is shown in Algorithm 2: the multiplications modulo A are done in the RNS basis $\mathcal{A}$ and the operations modulo B are done in $\mathcal{B}$.

Algorithm 2 Basic-MM-RNS($x, y, \mathcal{A}, \mathcal{B}$)

Require: x, y in $\mathcal{A} \cup \mathcal{B}$
Ensure: $xyA^{-1} \mod N$ in $\mathcal{A} \cup \mathcal{B}$
1: $[q]_{\mathcal{A}} \leftarrow [-xyN^{-1}]_{\mathcal{A}}$
2: $BE_{\mathcal{A}\to\mathcal{B}}([q]_{\mathcal{A}})$
3: $[z]_{\mathcal{B}} \leftarrow [(xy + qN)A^{-1}]_{\mathcal{B}}$
4: $BE_{\mathcal{B}\to\mathcal{A}}([z]_{\mathcal{B}})$
5: **return** $(z_{\mathcal{A}\cup\mathcal{B}})$

The second and fourth steps are necessary since if we want to compute $z \leftarrow (xy+qN)A^{-1} \mod B$ in $\mathcal{B}$ we need to convert the RNS representation of q from the basis $\mathcal{A}$ to the basis $\mathcal{B}$: the base extension (BE) performs this conversion. The fourth step is also necessary to have z represented in both bases $\mathcal{A}$ and $\mathcal{B}$.

Base extension. This is the most costly step in the RNS version of the Montgomery multiplication (Algorithm 2). We review the best known method to perform such RNS base extension. Let $x = (x_1, \ldots, x_t)_{\mathcal{A}}$ be the representation of an integer x in the RNS basis $\mathcal{A}$, the CRT 3 reconstructs x as follows:

$$\hat{x}_{a_i} = \left[x_{a_i} \times A_i^{-1}\right]_{a_i} \text{ for } i = 1, \ldots, t, \tag{4}$$

$$x = \left(\sum_{i=1}^{t} \hat{x}_{a_i} \times A_i\right) - \alpha A \tag{5}$$

The correcting term $-\alpha A$ corresponds to the reduction modulo A in (3). We get the RNS representation $[x]_{b_j}$ for $j = 1, \ldots, t$ of x in $\mathcal{B}$ by simply reducing

modulo b_j the expression in (5):

$$\begin{aligned} x^*_{b_j} &= \left[\sum_{i=1}^{t} \hat{x}_{a_i} \times A_i\right]_{b_j}, \text{ for } j = 1, \ldots, t, \\ [x]_{b_j} &= \left[x^*_{b_j} - \alpha A\right]_{b_j} \text{ for } j = 1, \ldots, t. \end{aligned} \tag{6}$$

We give some details on how to perform the above computations.

- *Computations of* $x^*_{b_j}$. If the constants $[A_i]_{b_j}$ are precomputed then $x^*_{b_j}$ for $j = 1, \ldots, t$ can be computed as

$$x^*_{b_j} = \left[\sum_{i=1}^{t} \hat{x}_{a_i} \times [A_i]_{b_j}\right]_{b_j}.$$

  There is an alternative method proposed by Garner in [5] which computes $x^*_{b_j}$, but we will not use it in this paper, so we do not recall it here. The reader may refer to [5] to further details on this method.
- *Computations of* α. The base extension in (6) necessitates also to compute α. We arrange (5) as follows

$$\sum_{i=1}^{t} \hat{x}_{a_i} \times A_i = x + \alpha A \Longrightarrow \sum_{i=1}^{t} \frac{\hat{x}_{a_i}}{a_i} = \frac{x}{A} + \alpha \Longrightarrow \alpha = \left\lfloor \sum_{i=1}^{t} \frac{\hat{x}_{a_i}}{a_i} \right\rfloor \tag{7}$$

  since when $0 < x < A$ we have $0 < x/A < 1$.

The MM-RNS algorithm. Following [7] we inject in Algorithm 2 the formulas (4), (5) and (7) corresponding to the computations of the base extensions. We obtain the Montgomery multiplication in RNS (MM-RNS) shown in Algorithm 3 after some modifications. Specifically, the base extension of q and the computation of z are merged as follows

$$z_{b_i} \leftarrow [(s_{b_i} + \sum_{j=1}^{t} q_{a_j} A_j N - \alpha A N) A^{-1}]_{b_i} = [s_{b_i} A^{-1} + (\sum_{j=1}^{t} q_{a_j} a_j^{-1} - \alpha) N]_{b_i}.$$

In the second base extension $BE_{\mathcal{B}\to\mathcal{A}}$ we rewrite $[B_j]_{a_i} = [b_j^{-1} B]_{a_i}$.

The complexity of each step of the MM-RNS algorithm is given in terms of the number of additions and multiplications modulo a_i or b_i. These complexities are detailed in Table 1. For the computation of α and β we assume that each a_i and b_i can be approximated by 2^w which simplifies the computations in Step 6 and Step 10 as a sequence of additions (cf. [7] for further details).

Constants used in MM-RNS. In Algorithm 3, an important number of constants take part of the computations:

$$\begin{aligned} &[N^{-1}]_{\mathcal{A}}, [N]_{\mathcal{B}}, \\ &[b_j^{-1}]_{a_i}, [a_j^{-1}]_{b_i} \text{ for } i, j = 1, \ldots, t, \\ &[B]_{a_i}, [B_i^{-1}]_{b_i} \text{ for } i = 1, \ldots, t, \\ &[A^{-1}]_{b_i}, [A_i^{-1}]_{a_i} \text{ for } i = 1, \ldots, t. \end{aligned} \tag{8}$$

Algorithm 3 MM-RNS$(x, y, \mathcal{A}, \mathcal{B})$

Require: x, y in $\mathcal{A} \cup \mathcal{B}$ for two RNS bases $\mathcal{A} = \{a_1, \ldots, a_t\}$ and $\mathcal{B} = \{b_1, \ldots, b_t\}$ s.t. $A = \prod_{i=1}^{t} a_i, B = \prod_{i=1}^{t} b_i$, $\gcd(A, B) = 1$, $1 \leq x, y \leq N, B \geq 4N$ et $A > 2N$.
Ensure: $xyA^{-1} \mod N$ in $\mathcal{A} \cup \mathcal{B}$
1: **Precomputations in** $\mathcal{B}$: $[N]_{\mathcal{B}}, [A^{-1}]_{\mathcal{B}}, [b_j^{-1}]_{\mathcal{B}}$ for $j = 1, \ldots, t$ and $[B_i^{-1}]_{b_i}$ for $i = 1, \ldots, t$
2: **Precomputations in** $\mathcal{A}$: $[N^{-1}]_{\mathcal{A}}$ and $[a_j^{-1}]_{\mathcal{A}}, [A_j^{-1}]_{a_j}$ for $j = 1, \ldots, t$ and $[B]_{\mathcal{A}}$
3: $s = [x \cdot y]_{\mathcal{A} \cup \mathcal{B}}$
4: //- - - - - - - - - base extension $\mathcal{A} \to \mathcal{B}$ - - - - - -
5: $q_{a_i} \leftarrow [s_{a_i} \times (-N^{-1}) \times A_i^{-1}]_{a_i}$ for $i = 1$ to t
6: $\alpha \leftarrow \left\lfloor \sum_{i=1}^{t} q_{a_i}/a_i \right\rfloor$
7: $z_{b_i} \leftarrow [s_{b_i} A^{-1} + (\sum_{j=1}^{t} q_{a_j} a_j^{-1} - \alpha)N]_{b_i}$ for $i = 1$ to t
8: //- - - - - - - - - base extension $\mathcal{B} \to \mathcal{A}$ - - - - - - -
9: $q_{b_i} \leftarrow [z_{b_i} \times B_i^{-1}]_{b_i}$ for $i = 1$ to t
10: $\beta \leftarrow \left\lfloor \sum_{i=1}^{t} q_{b_i}/b_i \right\rfloor$
11: $z_{a_i} \leftarrow [(\sum_{j=1}^{t} q_{b_j} b_j^{-1} - \beta)B]_{a_i}$ for $i = 1$ to t

Table 1. Complexity of MM-RNS

Step	#Mult.	#Add.
3	$2t$	0
5	$2t$	0
6	0	$t-1$
7	$t(t+2)$	$t(t+1)$
9	t	0
10	0	$t-1$
11	$t(t+1)$	t^2
Total	$t(2t+8)$	$t(2t+3)-2$

Only, the constants $[B^{-1}]_{a_i}, [B_i^{-1}]_{b_i}, [A]_{b_i}$ and $[A_i^{-1}]_{a_i}$ are susceptible to change and to be updated during the run of a modular exponentiation if the bases $\mathcal{A}$ and $\mathcal{B}$ are modified.

3 Leak resistant arithmetic in RNS

The authors in [1] notice that the use of RNS facilitates the randomization of the representation of an integer and consequently the randomization of a modular multiplication. Indeed, if a modular exponentiation $x^K \mod N$ is computed with MM-RNS the element is set in Montgomery representation

$$\widetilde{x} = x \times A \mod N$$

and in the RNS bases $\mathcal{A}$ and $\mathcal{B}$, i.e., $[\widetilde{x}]_{\mathcal{A} \cup \mathcal{B}}$. The Montgomery representation induces a multiplicative masking of the data x by the factor A. The authors in [1] propose to randomly construct the basis $\mathcal{A}$ to get a random multiplicative mask A on the data.

Specifically, the authors in [1] propose two levels of such randomization: random initialization of the bases $\mathcal{A}$ and $\mathcal{B}$ at the very beginning of a modular exponentiation and random permutations of RNS bases $\mathcal{A}$ and $\mathcal{B}$ all along the modular exponentiation.

Random initialization of the bases $\mathcal{A}$ and $\mathcal{B}$ and $\widetilde{x}$. We assume that we have a set of $2t$ moduli $\mathcal{M} = \{m_1, \ldots, m_{2t}\}$. At the beginning of the computations we randomly set

$$\begin{array}{l} \mathcal{A} \longleftarrow \{t \text{ random distinct elements in } \mathcal{M}\} \\ \mathcal{B} \longleftarrow \mathcal{M} \backslash \mathcal{A} \end{array} \tag{9}$$

Note that we always have $\mathcal{A} \cup \mathcal{B} = \mathcal{M}$.

Then the input of x of the modular exponentiation algorithm is first set in the residue number system $\mathcal{M} = \mathcal{A} \cup \mathcal{B}$ by reducing x modulo each a_i and b_i

$$[x]_{\mathcal{A}\cup\mathcal{B}} = ([x]_{a_0}, \ldots, [x]_{a_t}, [x]_{b_0}, \ldots, [x]_{b_t}).$$

Then we need to compute the Montgomery representation $[\widetilde{x}]_{\mathcal{A}\cup\mathcal{B}}$ from $[x]_{\mathcal{A}\cup\mathcal{B}}$. The authors in [1] give a method which simplifies this computation. They assume that the RNS representation of

$$\begin{array}{rl} M \mod N = (\prod_{i=1}^{2t} m_i) & \mod N \\ = A \times B & \mod N \end{array}$$

is precomputed. They compute $[\widetilde{x}]_{\mathcal{A}\cup\mathcal{B}}$ from $[x]_{\mathcal{A}\cup\mathcal{B}}$ by a single MM-RNS with bases $\mathcal{B}$ and $\mathcal{A}$ in reverse order:

$$\text{MM-RNS}([x]_{\mathcal{A}\cup\mathcal{B}}, [(M \bmod N)]_{\mathcal{A}\cup\mathcal{B}}, \mathcal{B}, \mathcal{A})$$

The output of this multiplication is the expected value:

$$[(x \times M \times B^{-1} \bmod N)]_{\mathcal{A}\cup\mathcal{B}} = [(x \times A \bmod N)]_{\mathcal{A}\cup\mathcal{B}} = [\widetilde{x}]_{\mathcal{A}\cup\mathcal{B}}.$$

Random change of the bases $\mathcal{A}$ and $\mathcal{B}$. The authors in [1] propose to change the bases $\mathcal{A}$ and $\mathcal{B}$ during the RSA exponentiation as follows:

$$\begin{array}{l} \mathcal{A}_{new} \longleftarrow \{t \text{ random distinct elements in } \mathcal{M}\} \\ \mathcal{B}_{new} \longleftarrow \mathcal{M} \backslash \mathcal{A}_{new} \end{array} \tag{10}$$

The bases $\mathcal{A}$ and $\mathcal{B}$ change all along the exponentiation, this implies to perform the base extension (BE) in MM-RNS using the approach of Garner [5] instead of the CRT formula. Otherwise the constants A_i and B_i would have to be updated which can be expensive.

The update of the bases $\mathcal{A}$ and $\mathcal{B}$ implies to also update the Montgomery representation $\widetilde{x} = x \times A_{old} \mod N$ of x from the old bases $\mathcal{A}_{old} \cup \mathcal{B}_{old}$ to the new

Algorithm 4 Update of $\widetilde{x}$

Require: $\widetilde{x}_{old}$ and $\mathcal{A}_{new}, \mathcal{B}_{new}, \mathcal{A}_{old}, \mathcal{B}_{old}$ and $[M \mod N]_{\mathcal{M}}$
Ensure: $\widetilde{x}_{new}$
1: $temp \leftarrow$ MM-RNS$(\widetilde{x}_{old}, (M \bmod N), \mathcal{B}_{new}, \mathcal{A}_{new})$
2: $x_{new} \leftarrow$ MM-RNS$(temp, 1, \mathcal{A}_{old}, \mathcal{B}_{old})$

representation $\widetilde{x} = x \times A_{new} \mod N$ in the new bases $\mathcal{A}_{new} \cup \mathcal{B}_{new}$. The proposed approach in [1] consists in two modular multiplications (cf. Algorithm 4).

We can easily check the validity of Algorithm 4: Step 1 computes $temp = (xA_{old}) \times (A_{new} \times B_{new}) \times B_{new}^{-1} \mod N$ and Step 2 correctly computes $x_{new} = (xA_{old}A_{new}) \times A_{old}^{-1} \mod N = xA_{new} \mod N$ in the required RNS bases.

The main drawback of this technique is that it is a bit costly: it requires two MM-RNS multiplications to perform the change of RNS representation. Consequently, using Table 1, we deduce that the amount of computation involved in this approach is as follows

$$\#\text{Mult.} = 2t(2t+8) \text{ and } \#\text{Add.} = 2t(2t+3) - 4.$$

4 Random update of the RNS bases with a set of spare moduli

In this section, our goal is to provide a cheaper variant of the leak resistant arithmetic in RNS proposed in [1] and reviewed in Section 3.

4.1 Proposed update of the bases and Montgomery representation

We present a first strategy which modifies only one modulus in $\mathcal{A}$ while keeping $\mathcal{B}$ unchanged during each update of the RNS bases. We need an additional set $\mathcal{A}'$ of spare moduli where we randomly pick the new modulus for $\mathcal{A}$. We have three sets of moduli:

- The first RNS basis $\mathcal{A} = \{a_1, \ldots, a_{t+1}\}$ which is modified after each loop iteration.
- The set $\mathcal{A}' = \{a'_1, \ldots, a'_{t+1}\}$ of spare moduli.
- The second RNS basis $\mathcal{B} = \{b_1, \ldots, b_{t+1}\}$ which is fixed at the beginning of the exponentiation.

The integers a_i, b_i and a'_i are all pairwise co-prime and are all of the form $2^w - \mu_i$ where w is the same for all moduli and $\mu_i < 2^{w/2}$. We will state later in Subsection 4.2 how large A and B have to be compared to N to render the proposed approach effective. But to give an insight A and B are roughly w-bits larger than N which means that the considered RNS bases contain $t+1$ moduli.

Update of the base $\mathcal{A}$. Updating the basis $\mathcal{A}$ is quite simple: we just swap one element of $\mathcal{A}$ with one element of $\mathcal{A}'$ as follows

- $r \leftarrow$ random in $\{1, \ldots, t, t+1\}$
- $r' \leftarrow$ random in $\{1, \ldots, t, t+1\}$
- $a_{r,new} \leftarrow a'_{r',old}$
- $a_{r',new} \leftarrow a_{r,old}$

In the sequel we will denote $\mathcal{A}_{old}$ and $\mathcal{A}_{new}$ the base $\mathcal{A}$ before and after the update, we will use similar notation for other updated data.

Update of the Montgomery-RNS representation. The modification of the basis requires at the same time the corresponding update of the Montgomery representation of x. Indeed we need to compute $\widetilde{x}_{new} = [(xA_{new} \mod N)]_{\mathcal{A}_{new} \cup \mathcal{B}}$ from its old Montgomery representation $\widetilde{x}_{old} = [(xA_{old} \mod N)]_{\mathcal{A}_{old} \cup \mathcal{B}}$. The following lemma establishes how to perform this update.

Lemma 1. *We consider two RNS bases $\mathcal{A}$ and $\mathcal{B}$ and let $\mathcal{A}'$ be the set of spare moduli. We consider an integer x modulo N given by its RNS-Montgomery representation $[\widetilde{x}]_{\mathcal{A} \cup \mathcal{B}} = [(x \times A \mod N)]_{\mathcal{A} \cup \mathcal{B}}$ where $A = a_1 a_2 \cdots a_{t+1}$. Let r and r' be two random integers in $\{1, \ldots, t, t+1\}$ and $\mathcal{A}_{new}$ and $\mathcal{A}'_{new}$ be the two set of moduli obtained after exchanging $a_{r,old}$ et $a'_{r',old}$. Then the new Montgomery-RNS representation of x in $\mathcal{A}_{new} \cup \mathcal{B}$ can be computed as follows:*

$$\begin{aligned} \lambda &= [-\widetilde{x}_{a_{r,old}} \times N^{-1}]_{a_{r,old}}, \\ \widetilde{x}_{new} &= [(\widetilde{x}_{old} + \lambda \times N) \times a_{r,old}^{-1} \times a_{r,new}]_{\mathcal{A}_{new} \cup \mathcal{B}} \end{aligned} \tag{11}$$

and satisfies $\widetilde{x}_{new} = (x \times A_{new}) \mod N$ given in the bases $\mathcal{A}_{new} \cup \mathcal{B}$.

Proof. We first notice that $s_1 = \widetilde{x}_{old} + \lambda N$ satisfies $s_1 \equiv \widetilde{x}_{old} \mod N$ and that

$$\begin{aligned} [s_1]_{a_{r,old}} &= [\widetilde{x}_{old} + \lambda \times N]_{a_{r,old}} \\ &= \left[\widetilde{x}_{old} - [\widetilde{x}]_{a_{r,old}} \times N^{-1} \times N\right]_{a_{r,old}} \\ &= 0. \end{aligned}$$

In other words s_1 can be divided by $a_{r,old}$ and then multiplied by $a_{r,new}$

$$\widetilde{x}_{new} = ((\widetilde{x}_{old} + \lambda N)/a_{r,old}) a_{r,new}.$$

which satisfies

$$\begin{aligned} \widetilde{x}_{new} \mod N &= ((\widetilde{x}_{old} + \lambda N)/a_{r,old}) a_{r,new} \mod N \\ &= x A_{old} a_{r,old}^{-1} a_{r,new} \mod N \\ &= x A_{new} \mod N \end{aligned}$$

The value of $\widetilde{x}_{new}$ is computed in the RNS basis $\mathcal{A}_{new} \cup \mathcal{B}$ by replacing the division by $a_{r,old}$ by a multiplication by $a_{r,old}^{-1}$ and by noticing that its value modulo $a_{r,new}$ is equal to 0. This leads to (11).

■

Update of the constants. If we want to apply MM-RNS (Algorithm 3) after the update of the basis $\mathcal{A}$ and the Montgomery-RNS representation of x, we need also to update the constants involved in Algorithm 3. The constants considered are the one listed in (8) along with the following additional set of constants associated to the set of moduli $\mathcal{A}'$:

$$\begin{array}{l}
[-N^{-1}]_{a'_i} \text{ i= 1,...,t+1,} \\
[A^{-1}]_{a'_i}, [B]_{a'_i} \text{ i= 1,...,t+1,} \\
[b_j^{-1}]_{a'_i}, [a_j'^{-1}]_{b_i} \text{ i,j= 1,...,t+1,} \\
[a_j^{-1}]_{a'_i}, [a_j'^{-1}]_{a_i} \text{ i,j= 1,...,t+1.}
\end{array}$$

These constants are updated as follows:

- *Constant* $-N^{-1}$. The constants $[-N^{-1}]_{a_i}$ only changes when $i = r$ and $[-N^{-1}]_{a'_i}$ only change when $i = r'$. Then this update consists only in a single swap
$$swap([-N^{-1}]_{a_r}, [-N^{-1}]_{a'_{r'}}).$$
- *Constant* N. The constants $[N]_{b_i}, i = 1, \ldots, t+1$ do not change when the base $\mathcal{A}$ is updated.
- *Constants* A_i^{-1} *and* A^{-1}. The constants $[A_i^{-1}]_{a_i}$ and $[A^{-1}]_{a'_i}$ and $[A^{-1}]_{b_i}$ are updated as follows:

$$\begin{array}{l}
[A_i^{-1}]_{a_i} \leftarrow [[A_i^{-1}]_{a_i} \times a_{r,new}^{-1} \times a_{r,old}]_{a_i} \text{ for } i \neq r, \\
[A^{-1}]_{a'_i} \leftarrow [[A^{-1}]_{a'_i} \times a_{r,new}^{-1} \times a_{r,old}]_{a'_i} \text{ for } i \neq r', \\
[A^{-1}]_{b_i} \leftarrow [[A^{-1}]_{b_i} \times a_{r,new}^{-1} \times a_{r,old}]_{b_i} \text{ for } i \neq r',
\end{array}$$

  and the two remaining special cases are:

$$\begin{array}{ll}
[A_r^{-1}]_{a_{r,new}} \leftarrow [[A^{-1}]_{a'_{r',old}} \times a_{r,old}]_{a_{r',old}} & \text{(Note that } a'_{r',old} = a_{r,new}\text{)}, \\
[A^{-1}]_{a'_{r',new}} \leftarrow [[A_r^{-1}]_{a_{r,old}} \times a_{r,new}^{-1}]_{a_{r,old}} & \text{(Note that } a_{r,old} = a'_{r',new}\text{)}.
\end{array}$$

- *Constants* $[B_i^{-1}]_{b_i}, [B]_{a_i}$ *and* $[B]_{a'_i}$. The constants $[B_i]_{b_i}$ and $[B]_{a_i}$ for $i \neq r$ and $[B]_{a'_i}$ for $i \neq r'$ are not affected by the modification on $\mathcal{A}$. The only required modification is the following swap
$$\text{swap}([B]_{a_r}, [B]_{a'_{r'}}).$$
- *Constants* $[b_j^{-1}]_{a_i}, [b_j^{-1}]_{a'_i}, [a_j^{-1}]_{a_i}, [a_j^{-1}]_{a'_i}, [a_j'^{-1}]_{a_i}$, *and* $[a_j'^{-1}]_{a'_i}$. The constants which evolve are the ones corresponding to a_r and $a'_{r'}$ and require only swaps:

$$\begin{array}{ll}
\text{swap}([b_j^{-1}]_{a_r}, [b_j^{-1}]_{a'_{r'}}) & \text{for } j = 1, \ldots, t+1, \\
\text{swap}([a_r^{-1}]_{b_j}, [a_{r'}'^{-1}]_{b_j}) & \text{for } j = 1, \ldots, t+1, \\
\text{swap}([a_j^{-1}]_{a_r}, [a_j^{-1}]_{a'_{r'}}) & \text{for } j = 1, \ldots, t+1 \text{ and } j \neq r, \\
\text{swap}([a_j'^{-1}]_{a_r}, [a_j'^{-1}]_{a'_{r'}}) & \text{for } j = 1, \ldots, t+1 \text{ and } j \neq r'.
\end{array}$$

Complexity of the updates. We evaluate the complexity of the above random change of the basis $\mathcal{A}$: the update of $\widetilde{x}$ and the update of the constants. We do not consider swap operations since they do not require any computations. The cost of the update of the Montgomery representation $\widetilde{x}$ of x contributes to $6t+4$ multiplications and $2t+1$ additions and the contribution of the update of the constants is equal to $6t+2$ multiplications.

Table 2. Complexity of the updates when using a set of spare moduli

Operation	#Mult.	#Add.
Updates of $\widetilde{x}$	$6t+4$	$2t+1$
Updates of the constants	$6t+2$	0

4.2 Proposed randomized Montgomery-ladder and its validity

We present the modified version of the Montgomery-ladder: compared to the original Montgomery-ladder (Algorithm 1), this version inserts an update of the RNS bases and related constants along with an update of the data at the beginning of the loop iteration. This approach is shown in Algorithm 5. For the sake of simplicity, conversions between RNS and regular integer representation are skipped.

Algorithm 5 Randomized-Montgomery-ladder

Require: $x \in \{0, \ldots, N-1\}$ and $K = (k_{\ell-1}, \ldots, k_0)_2$, three RNS bases $\mathcal{A}, \mathcal{A}', \mathcal{B}$
1: $\widetilde{r_0} \leftarrow [\widetilde{1}]_{\mathcal{A}\cup\mathcal{B}}$
2: $\widetilde{r_1} \leftarrow [\widetilde{x}]_{\mathcal{A}\cup\mathcal{B}}$
3: **for** i **from** $\ell-1$ **downto** 0 **do**
4: UpdateBases($\mathcal{A}, \mathcal{A}'$)
5: UpdateMontRep($\widetilde{r_0}$)
6: UpdateMontRep($\widetilde{r_1}$)
7: **if** $k_i = 0$ **then**
8: $\widetilde{r_1} \leftarrow$ MM-RNS($\widetilde{r_1}, \widetilde{r_0}$)
9: $\widetilde{r_0} \leftarrow$ MM-RNS($\widetilde{r_0}, \widetilde{r_0}$)
10: **end if**
11: **if** $k_i = 1$ **then**
12: $\widetilde{r_0} \leftarrow$ MM-RNS($\widetilde{r_1}, \widetilde{r_0}$)
13: $\widetilde{r_1} \leftarrow$ MM-RNS($\widetilde{r_1}, \widetilde{r_1}$)
14: **end if**
15: **end for**
16: **return** (r_0)

Now, we establish that the above algorithm correctly outputs the expected result $x^K \mod N$. Indeed, during the execution of the algorithm an overflow could occur: some data could become larger than A or B. To show that no overflow occurs we first establish the growing factor produced by an update of the Montgomery representation.

Lemma 2. *Let $\mathcal{A}_{old}, \mathcal{A}'_{old}, \mathcal{B}_{old}, \mathcal{A}_{new}, \mathcal{A}'_{new}$ and $\mathcal{B}_{new}$ be the new and the old RNS bases. Let $a_{i_{max},old}$ the largest modulus in $\mathcal{A}_{old}$ and $a_{i_{max},new}$ the largest modulus in $\mathcal{A}_{new}$. Assume that $\widetilde{x}_{old} < N a_{i_{max},old}$ and let a_r and $a'_{r'}$ be the two*

moduli swapped in $\mathcal{A}$ and $\mathcal{A}'$. Then we have

$$\widetilde{x}_{new} < 4Na_{i_{max},new}$$

Proof. From Lemma 1 we have the following expression of $\widetilde{x}_{new}$

$$\widetilde{x}_{new} = ((\widetilde{x}_{old} + \lambda N)/a_r) \times a'_{r'}. \tag{12}$$

We then notice that $\lambda < a_{i_{max},old}$. We use the fact that $\widetilde{x}_{old} < Na_{i_{max},old}$ and we expand the product in (12), this gives:

$$\begin{aligned} \widetilde{x}_{new} &< (Na_{i_{max},old} + a_{i_{max},old}N) \times \tfrac{1}{a_r} \times a'_{r'} \\ &\leq 2N \times \tfrac{a_{i_{max},old}}{a_r} \times a'_{r'}. \end{aligned} \tag{13}$$

We then use that $a_i = 2^w - \mu_i$ with $0 \leq \mu_i < 2^{w/2}$, which implies that for any i, j

$$\begin{aligned} 0 < \tfrac{a_i}{a_j} = \tfrac{2^w-\mu_i}{2^w-\mu_j} = \tfrac{2^w-\mu_j+\mu_j-\mu_i}{2^w-\mu_j} &= 1 + \tfrac{\mu_j-\mu_i}{2^w-\mu_j} \\ &< 1 + \tfrac{2^{w/2}}{2^w-\mu_j} < 2. \end{aligned}$$

In particular for $i = i_{max}$ and $j = r$ we have $0 < \frac{a_{i_{max},old}}{a_r} < 2$. We use this to arrange (13) as follows:

$$\widetilde{x}_{new} < 4Na'_{r'} \leq 4Na_{i_{max},new}.$$

■

Knowing the growing factor induced by the update of the Montgomery representation helps us to state a sufficient condition to prevent an overflow in Algorithm 5.

Lemma 3. *Let A_{min} be the product of the t smallest moduli in $\mathcal{A} \cup \mathcal{A}'$. Let $a_{i_{max}}$ be the largest modulus of $\mathcal{A}$. If N satisfies*

$$N < \frac{A_{min}}{32} \tag{14}$$

and if B is larger than any A then the following assertions hold:

i) The data $\widetilde{r_0}$ and $\widetilde{r_1}$ in Algorithm 5 are $< Na_{i_{max}}$ at the end of each loop.
ii) Algorithm 5 correctly computes $r_0 = x^K \mod N$.

Proof. i) Let us prove that an update of the base $\mathcal{A}$ followed by a modular multiplication with MM-RNS keeps the data in the interval $[0, Na_{i_{max}}]$. We consider $\widetilde{x}_{old} < Na_{i_{max},old}$ and $\widetilde{y}_{old} < Na_{i_{max},old}$. Then, from Lemma 2, we know that the updates on $\widetilde{x}_{old}$ and $\widetilde{y}_{old}$ provide:

$$\widetilde{x}_{new} < 4Na_{i_{max},new} \text{ and } \widetilde{y}_{new} < 4Na_{i_{max},new}.$$

If we execute an MM-RNS algorithm with inputs $\widetilde{x}_{new}, \widetilde{y}_{new}$ and bases $\mathcal{A}_{new}$ and $\mathcal{B}$ we obtain a z satisfying

$$\begin{aligned} z &= (\widetilde{x}_{new} \times \widetilde{y}_{new} + qN)/A_{new} \\ &< (16N^2 a_{i_{max},new}^2 + A_{new}N)/A_{new} \\ &< N(\tfrac{16N}{A_{new}/a_{i_{max},new}} \times a_{i_{max},new} + 1) \end{aligned}$$

Now, since $\frac{A_{new}}{a_{i_{max},new}}$ is the product of t moduli of $\mathcal{A} \cup \mathcal{A}'$ it satisfies $A_{min} \leq \frac{A_{new}}{a_{i_{max},new}}$. Then using (14) we obtain that

$$\frac{32N}{A_{new}/a_{i_{max},new}} \times \frac{a_{i_{max},new}}{2} < \frac{a_{i_{max},new}}{2}$$

and consequently $z < a_{i_{max},new}N$, as required.

ii) At the beginning of each loop $\widetilde{r_0}$ and $\widetilde{r_1}$ are in $[0, Na_{i_{max}}]$ then, from i), they are in $[0, Na_{i_{max}}]$ at the end of the loop. Consequently all the computations in the algorithm are done without overflow and which then correctly outputs $r_0 = x^K \mod N$.

■

5 Complexity comparison and conclusion

In Appendix A we present a variant of the proposed randomization. This variant avoids the use of the set of spare moduli $\mathcal{A}'$: the modified modulus in $\mathcal{A}$ is randomly picked in $\mathcal{B}$. The complexity of the update of the RNS bases $\mathcal{A}, \mathcal{B}$ and the update of the Montgomery representation are sightly larger compared to the approach of Section 4, but the memory requirement is reduced and the number of moduli is also reduced.

In Table 3 we report the complexity of the randomization in the Montgomery-ladder exponentiation for the two following cases:

1. *Only one modulus is modified in the basis $\mathcal{A}$.* In this case, for each loop turn, the proposed approach in Section 4 and Appendix A requires an update of the constant and an update of $\widetilde{r_0}$ and $\widetilde{r_1}$ as shown in Algorithm 5.
2. *s moduli are modified in $\mathcal{A}$.* At each loop turn, we perform s consecutive updates of the RNS bases $\mathcal{A}, \mathcal{A}'$ and the data following the strategy of Section 4: this requires s updates of the constants and s updates of $\widetilde{r_0}$ and $\widetilde{r_1}$. In this case, since an update of $\widetilde{r_i}$ multiply by 4 (cf. Lemma 2) at the end of the s the two data $\widetilde{r_0}$ and $\widetilde{r_1}$ are multiplied by 4^s. This requires to expand the three bases $\mathcal{A}, \mathcal{A}'$ and $\mathcal{B}$ with an additional modulus assuming that $2 \times 4^{2s} < 2^w$ in order to prevent an overflow in Algorithm 5. The resulting complexity of this randomization is given in Table 3.

For comparison purpose we provide in Table 3 the complexity when the randomization of [1] is performed at each loop turn in a Montgomery ladder. The complexity can be easily deduced from the complexity results of Section 3.

Table 3. Cost of the randomization in one loop iteration of randomized Montgomery-ladder

Randomization	Method	# Mul.	# Add.	Memory
1 modulus per loop	Section 4	$18t + 10$	$4t + 2$	$8t^2 + 24t + 18$
1 modulus per loop	Appendix A	$24t + 26$	$4t + 4$	$8t^2 + 19t + 20$
s moduli per loop	$s\times$ Section 4	$s(12t + 20) + 6t + 8$	$s(4t + 6)$	$8t^2 + 42t + 52$
$t/2$ moduli per loop (in average)	[1]	$8t^2 + 32t$	$8t^2 + 12t - 4$	$8t^2$

The above complexities show that we get a cheaper randomization by changing only one modulus, at a cost of a lower level of randomization. We can increase this level by changing more than one modulus at each loop turn, resulting in a trade-off between randomization and complexity. For the average randomization of $s = t/2$ moduli changed per loop turn, our method requires $6t^2 + O(t)$ multiplications and $2t^2 + 3t$ additions: this is better than the complexity of [1]. Another advantage of our technique is that it works in the cox-rower architecture [7] which is the most popular architecture for RNS implementation.

Aknowledgement. This work was partly supported by PAVOIS ANR 12 BS02 002 02. Part of this work was initiated when G. Perin was doing his PhD thesis at the LIRMM.

References

1. J.-C. Bajard, L. Imbert, P.-Y. Liardet, and Y. Teglia. Leak Resistant Arithmetic. In *CHES*, volume 3156 of *LNCS*, pages 62–75. Springer, 2004.
2. M. Ciet and M. Joye. (Virtually) Free Randomization Techniques for Elliptic Curve Cryptography. In *ICICS 2003*, volume 2836 of *LNCS*, pages 348–359. Springer, 2003.
3. C. Clavier, B. Feix, G. Gagnerot, M. Roussellet, and V. Verneuil. Horizontal Correlation Analysis on Exponentiation. In *Proceedings of ICICS 2010*, volume 6476 of *LNCS*, pages 46–61. Springer, 2010.
4. J.-S. Coron. Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems. In *CHES*, pages 292–302, 1999.
5. H.L. Garner. The Residue Number System. *IRE Trans. on Elctronic Computers*, 8:140–147, June 1959.
6. M. Joye and S.-M. Yen. The Montgomery Powering Ladder. In *CHES 2002*, volume 2523 of *LNCS*, pages 291–302. Springer, 2002.
7. S. Kawamura, M. Koike, F. Sano, and A. Shimbo. Cox-Rower Architecture for Fast Parallel Montgomery Multiplication. In *Proc. EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 523–538. Springer Verlag, 2000.
8. P.C. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In *Advances in Cryptology, CRYPTO'99*, volume 1666 of *LNCS*, pages 388–397. Springer, 1999.

9. P. Montgomery. Modular Multiplication Without Trial Division. *Math. Computation*, pages 519–521, 1985.
10. G. Perin, L. Imbert, L. Torres, and P. Maurine. Attacking Randomized Exponentiations Using Unsupervised Learning. In *Proceedings of COSADE 2014*, volume 8622 of *LNCS*, pages 144–160. Springer, 2014.
11. K.C. Posch and R. Posch. Modulo Reduction in Residue Number Systems. *IEEE Trans. Parallel Distrib. Syst.*, 6(5):449–454, 1995.
12. 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.
13. C.D. Walter. Sliding Windows Succumbs to Big Mac Attack. In *CHES*, number Generators in LNCS, pages 286–299, 2001.

A Random update of the basis without a set of spare moduli

In this appendix we present a second approach for a low cost update of the RNS basis $\mathcal{A}$. We still want to modify the basis $\mathcal{A}$ by changing only one modulus. But we do not use a set of spare moduli $\mathcal{A}'$. The proposed approach exchanges a random modulus picked in $\mathcal{A}$ with a random modulus picked in $\mathcal{B}$. Additionally to the two RNS bases $\mathcal{A}$ and $\mathcal{B}$ we will need an additional modulus c:

- $\mathcal{A} = \{a_1, \ldots, a_{t+1}\}$,
- $\mathcal{B} = \{b_1, \ldots, b_{t+1}\}$,
- and an additional modulus $\{c\}$.

We use the following slightly modified version of the Basic-MM-RNS algorithm (Algorithm 2):

Require: x, y in $\mathcal{A} \cup \mathcal{B}$
Ensure: $xyA^{-1} \mod N$ in $\mathcal{A} \cup \{c\} \cup \mathcal{B}$
$[q]_{\mathcal{A}} \leftarrow [-xyN^{-1}]_{\mathcal{A}}$
$BE_{\mathcal{A}\to\mathcal{B}}([q]_{\mathcal{A}})$
$[z]_{\mathcal{B}} \leftarrow [(xy + qN)A^{-1}]_{\mathcal{B}}$
$BE_{\mathcal{B}\to\mathcal{A}\cup\{c\}}([z]_{\mathcal{B}})$ // modified operation
return $(z_{\mathcal{A}\cup\mathcal{B}})$

Only the last step is modified: we compute z in $\mathcal{A} \cup \{c\}$ instead of $\mathcal{A}$. This can be performed by modifying the last step of Algorithm 3 as follows

$$[z]_{\mathcal{A}\cup\{c\}} \leftarrow [(\sum_{j=1}^{t} q_{b_j} b_j^{-1} - \beta)B]_{\mathcal{A}\cup\{c\}}.$$

The random update of the bases $\mathcal{A}, \{c\}$ and $\mathcal{B}$ is performed as follows:

$$\begin{aligned} r &\leftarrow \text{ random in } \{1, \ldots, t+1\}, \\ r' &\leftarrow \text{ random in } \{1, \ldots, t+1\}, \\ a_{r,new} &\leftarrow b_{r',old}, \\ b_{r',new} &\leftarrow c_{old}, \\ c_{new} &\leftarrow a_{r,old}. \end{aligned} \tag{15}$$

Update of the Montgomery representation. The update of the bases $\mathcal{A}$, $\{c\}$ and $\mathcal{B}$ requires also to update the representation of $\widetilde{x}_{old} = xA_{old} \mod N$ given in $\mathcal{A}_{old} \cup \{c_{old}\} \cup \mathcal{B}_{old}$ to the new value $\widetilde{x}_{new} = xA_{new} \mod N$ in $\mathcal{A}_{new} \cup \mathcal{B}_{new}$. The following lemma establishes how to perform this update.

Lemma 4. *We consider two RNS bases $\mathcal{A}$ and $\mathcal{B}$ and an additional modulus $\{c\}$ which are updated as specified in* (15)*. We consider an integer x given by its Montgomery representation $\widetilde{x}_{old} = xA_{old} \mod N$ in $\mathcal{A}_{old} \cup \{c_{old}\} \cup \mathcal{B}_{old}$. We obtain $\widetilde{x}_{new}$ by first performing*

$$\begin{aligned} x_{a_i,new} &\leftarrow x_{a_i,old}, \text{ for } i \neq r, \\ x_{a_r,new} &\leftarrow x_{b_{r'},old}, \\ x_{b_i,new} &\leftarrow x_{b_i,old}, \text{ for } i \neq r', \\ x_{b_{r'},new} &\leftarrow x_{c,old}, \\ x_{c,new} &\leftarrow x_{a_r,old}. \end{aligned} \tag{16}$$

and then computes

$$\begin{aligned} \lambda &= [-[\widetilde{x}_{old}]_{a_{r,old}} \times N^{-1}]_{a_{r,old}}, \\ \widetilde{x}_{new} &= [(\widetilde{x}_{old} + \lambda \times N) \times a_{r,old}^{-1} \times a_{r,new}]_{\mathcal{A}_{new} \cup \mathcal{B}}. \end{aligned} \tag{17}$$

Proof. The first set of operations in (16) re-expresses $\widetilde{x}_{old}$ in $\mathcal{A}_{new} \cup \mathcal{B}_{new}$ by permuting the coefficients of $\widetilde{x}_{old}$ based on the update of the bases in (15). The second set of operations (17) consists to first compute $s_1 = \widetilde{x}_{old} + \lambda N$ which satisfies $s_1 \equiv \widetilde{x}_{old} \mod N$ and

$$\begin{aligned} [s_1]_{a_{r,old}} &= [\widetilde{x}_{old} + \lambda \times N]_{a_{r,old}} \\ &= \left[\widetilde{x}_{old} - [\widetilde{x}]_{a_{r,old}} \times N^{-1} \times N\right]_{a_{r,old}} \\ &= 0. \end{aligned}$$

Then the operation $((\widetilde{x}_{old} + \lambda N)/a_{r,old}) \times a_{r,new}$ involves an exact division by $a_{r,old}$ and a multiplication by $a_{r,new}$ and produces $x \times A_{new}$ modulo N. Since $a_{r,old}$ is invertible in $\mathcal{A}_{new} \cup \mathcal{B}_{new}$ we replace the division by a multiplication with the inverse $a_{r,old}^{-1}$ in $\mathcal{A}_{new} \cup \mathcal{B}_{new}$ and a multiplication by $a_{r,new}$ which leads to (17). ■

Update of the constants. In order to apply the MM-RNS algorithm after the update of the bases $\mathcal{A} \cup \{c\} \cup \mathcal{B}$ and of $[\widetilde{x}]_{\mathcal{A} \cup \mathcal{B}}$ we need to also update the constants involved in Algorithm 3. These constants are listed in (8), but we need to consider also the following additional set of constants relative to the modulus c:

$$\begin{aligned} &[a_i^{-1}]_c, [b_i^{-1}]_c \text{ for } i = 1, \ldots, t+1, \\ &[c^{-1}]_{a_i}, [c^{-1}]_{b_i} \text{ for } i = 1, \ldots, t+1, \\ &[N]_c, [N^{-1}]_c, [A]_c, [A^{-1}]_c, [B]_c, [B^{-1}]_c. \end{aligned}$$

Moreover since a modulus in $\mathcal{A}$ is susceptible to become a modulus in $\mathcal{B}$ and reciprocally, we need to also maintain the following constants:

$$\begin{aligned} &[A]_{b_i}, [A^{-1}]_{b_i} \text{ for } i = 1, \ldots, t+1, \\ &[B]_{a_i}, [B^{-1}]_{a_i} \text{ for } i = 1, \ldots, t+1. \end{aligned}$$

The proposed updates of the constants are listed below:

- *Constants* $N, N^{-1}, a_i^{-1}, b_i^{-1}, c^{-1}$. The update of the constants only consists in permutation, and does not require any computation.
- *Constants* A^{-1} *and* A_i^{-1}. We update these constants as follows:

$$\begin{aligned}
[A^{-1}]_{\mathcal{B}_{new}} &\leftarrow [A_{old}^{-1} \times a_{r,old} \times a_{r,new}^{-1}]_{\mathcal{B}_{new}},\\
[A^{-1}]_{c_{new}} &\leftarrow [[A_{r,old}^{-1}]_{a_{r,old}} \times a_{r,new}^{-1}]_{c_{new}} (\text{ since } c_{new} = a_{r,old}),\\
[A_j^{-1}]_{a_j} &\leftarrow [[A_j^{-1}]_{a_j} \times a_{r,new}^{-1} \times a_{r,old}]_{a_j} \text{ for } j \neq r,\\
[A_r^{-1}]_{a_{r,new}} &\leftarrow [[A_{old}^{-1}]_{b_{r',old}} \times a_{r,old}]_{a_{r,new}} (\text{ since } b_{r',old} = a_{r,new}).
\end{aligned}$$

- *Constants* B *and* B_i. The updates work as follows:

$$\begin{aligned}
[B_{new}]_{a_i} &\leftarrow [B_{old} \times b_{r',old}^{-1} \times b_{r',new}]_{a_i} \text{ for } i \neq r,\\
[B_{new}]_{a_{r,new}} &\leftarrow [[B_{r',old}]_{b_{r',old}} \times b_{r',new}]_{a_{r,new}} (\text{ since } a_{r,new} = b_{r',old}),\\
[B_{new}]_{c_{new}} &\leftarrow [[B_{old}]_{a_{r,old}} \times b_{r',old}^{-1} \times b_{r',new}]_{c_{new}} (\text{ since } c_{new} = a_{r,old}),\\
[B_{j,new}]_{b_j} &\leftarrow [[B_j]_{b_j} \times b_{r',old}^{-1} \times b_{r',new}]_{b_j} \text{ for } j \neq r',\\
[B_{r',new}]_{b_{r',new}} &\leftarrow [[B_{old}]_{c_{old}} \times b_{r',old}^{-1}]_{b_{r',new}} (\text{ since } c_{old} = b_{r',new}).
\end{aligned}$$

- *Constants* B^{-1} *and* B_i^{-1}. We update these constants as follows:

$$\begin{aligned}
[B_{new}^{-1}]_{a_i} &\leftarrow [B_{old}^{-1} \times b_{r',old} \times b_{r',new}^{-1}]_{a_i}, \text{ for } i \neq r\\
[B_{new}^{-1}]_{a_{r,new}} &\leftarrow [[B_{r',old}^{-1}]_{b_{r'}} \times b_{r',new}^{-1}]_{a_{r,new}}, (\text{ since } a_{r,new} = b_{r',old}),\\
[B_{new}^{-1}]_{c_{new}} &\leftarrow [[B_{old}^{-1}]_{a_{r,old}} \times b_{r',old} \times b_{r',new}^{-1}]_{c_{new}}, (\text{ since } c_{new} = a_{r,old}),\\
[B_{j,new}^{-1}]_{b_j} &\leftarrow [[B_j^{-1}]_{b_j} \times b_{r',old} \times b_{r',new}^{-1}]_{b_j}, \text{ for } j \neq r',\\
[B_{r',new}^{-1}]_{b_{r',new}} &\leftarrow [[B_{old}^{-1}]_{c_{old}} \times b_{r',old}]_{b_{r',new}} (\text{ since } c_{old} = b_{r',new}).
\end{aligned}$$

The complexity of the update of the Montgomery representation (Lemma 4) and the updates of the constants can be directly deduced from the above formulas. Theses complexities are given in the table below.

Operation	#Mult.	#Add.
Updates of $\widetilde{x}$	$6t+7$	$2t+2$
Updates of the constants	$12t+12$	0