# Efficient Regular Modular Exponentiation Using Multiplicative Half-Size Splitting

Christophe Negre, Thomas Plantard



# Efficient Regular Modular Exponentiation Using Multiplicative Half-Size Splitting

Christophe Negre and Thomas Plantard

**Abstract**

In this paper, we consider efficient RSA modular exponentiation $x^K \mod N$ protected against simple side channel analyses like timing attack and simple power analysis. To reach this goal we use a multiplicative splitting of the integer $x$ modulo $N$ into two half-size integers. We then take advantage of this splitting to modify the square-and-multiply exponentiation as a regular sequence of squarings always followed by a multiplication by a half-size integer. The proposed method requires around 16% less word operations compared to Montgomery-ladder, square-always and square-and-multiply-always exponentiations. These theoretical results are validated by our implementation results which show an improvement around 16%.

## I. Introduction

RSA [1] is nowadays the most used public key cryptosystem. The main operation in RSA protocols is an exponentiation $x^K \mod N$ where $N = pq$ with $p$ and $q$ prime. The private data are the two prime factors of $N$ and the private exponent $K$ used to decrypt or sign a message. In order to insure a sufficient level security $N$ and $K$ are chosen large enough to render the factorization of $N$ impossible: they are typically 2000 bit integers. The basic approach to perform efficiently the modular exponentiation is the square-and-multiply algorithm which scans the bits $k_i$ of the exponent $K$ and perform a sequence of squarings followed by a multiplication when $k_i$ is equal to one.

When the cryptographic computations are performed on an embedded device, an adversary can monitor power consumption [2] or electronic emanation [3]. If the power or electromagnetic traces of a multiplication and a squaring differ slightly, an adversary can read the sequence of squarings and multiplications directly on a single power or electromagnetic trace of a modular exponentiation. In the literature these attacks are referred to as simple power analysis (SPA) and simple electromagnetic analysis (SEMA), respectively.

Consequently, modular exponentiation have to be implemented in order to prevent such side channel analysis. The first direct approach which prevents this attack is the multiply-always exponentiation which performs all squarings as multiplications. But, unfortunately, it has been shown in [4] that this multiply-always strategy is still weak against an SPA or SEMA: an operation $r \times r$ and $r \times r'$ have different output Hamming weight. The authors in [5] proposed a square-always approach which performs a multiplication as the combination of two squarings. They then notice that in this case the attack of [4] does not apply. But both multiply-always and square-always approaches still leak

some information about the exponent: the computation time is correlated to the Hamming weight of the exponent, which is then leaked out.

A prerequisite to be SPA resistant is then to be regular and constant time. A first method which satisfies both of these property is the square-and-multiply-always exponentiation proposed by Coron [6]. Its principle is to always perform a multiplication after a squaring, i.e., if the bit $k_i = 0$ then a dummy multiplication is performed. Another popular strategy is the Montgomery-ladder [7] which also performs an exponentiation through a regular sequence of squarings always followed by a multiplication.

We present in this paper an alternative approach for regular and constant time exponentiation $x^K \mod N$. Our method uses a multiplicative splitting of $x$ into two halves. We modify the square-and-multiply algorithm as a regular sequence of squarings always followed by a multiplication with half-size integer. The half-size multiplications and squarings modulo $N$ are computed with the method of Montgomery [8], we then also provide a version of the proposed exponentiation with Montgomery modular multiplications adapted to the size of the operands. We analyze the complexity of this approach: Table I, below, contains the basic cost per loop turn of an exponentiation. We notice that the proposed approach always reach the best complexity while having the higher security level compared to the best known method of the literature.

Table I
COMPLEXITY IN TERMS OF WORD OPERATIONS PER LOOP TURN

| Algorithm | Regular | Constant time | Complexity per loop turn | |
|---|---|---|---|---|
| | | | # word add. | # word mult. |
| Square-and-multiply | ✗ | ✗ | $5t^2 + O(t)$ | $\frac{5}{2}t^2 + O(t)$ |
| Multiply-always [5] | ✓ | ✗ | $6t^2 + O(t)$ | $3t^2 + O(t)$ |
| Square-always [5] | ✓ | ✗ | $6t^2 + O(t)$ | $3t^2 + O(t)$ |
| Square-and-multiply-always [6] | ✓ | ✓ | $7t^2 + O(t)$ | $\frac{7}{2}t^2 + O(t)$ |
| Montgomery-ladder [7] | ✓ | ✓ | $7t^2 + O(t)$ | $\frac{7}{2}t^2 + O(t)$ |
| Montgomery-ladder with CM [9] | ✓ | ✓ | $6t^2 + O(t)$ | $3t^2 + O(t)$ |
| Proposed approach | ✓ | ✓ | $5t^2 + O(t)$ | $\frac{5}{2}t^2 + O(t)$ |

The remainder of the paper is organized as follows. Section II summarizes state of the art methods for regular modular exponentiation. In Section II-C we review techniques to compute a multiplicative splitting of an integer modulo $N$. In Section III we then present a new modular exponentiation algorithm which uses this splitting to render regular the square-and-multiply exponentiation. In Section IV, we present a version of the proposed exponentiation which incorporates Montgomery modular multiplications. Finally, in Section V, we evaluate the complexity of the proposed algorithm, provide implementations results and discuss security issues related to side channel analysis.

## II. REVIEW OF REGULAR MODULAR EXPONENTIATION

We review in this section several methods for performing an exponentiation $x^K \mod N$. The simplest and the most popular method is the square-and-multiply exponentiation [10]. The bits of the exponent $K$ are scanned from

left to right, for each bit a squaring is performed and is followed by a multiplication by $x$ if the bit is equal to 1. This method is detailed in Algorithm 1.

**Algorithm 1** Square-and-multiply

**Require:** $x \in \{0, \ldots, N-1\}$ and $K = (k_{\ell-1}, \ldots, k_0)_2$

1: $r \leftarrow 1$
2: **for** $i$ **from** $\ell - 1$ **downto** $0$ **do**
3: $\quad r \leftarrow r^2 \mod N$
4: $\quad$ **if** $k_i = 1$ **then**
5: $\quad\quad r \leftarrow r \times x \mod N$
6: $\quad$ **end if**
7: **end for**
8: **return** $r$

The sequence of squarings and multiplications in the square-and-multiply method has some irregularities due to the irregular sequence of the bits $k_i$ equal to 1. This can be used to mount a side channel attack by monitoring the power consumption or the electromagnetic emanation of the circuit performing the computations. Indeed, if the monitored signal of a multiplication and a squaring have a different shape, then, we can directly read on the power trace the sequence of squarings and multiplications. If a trace of a multiplication appears between two subsequent squarings then we deduce that the corresponding bit is 1, otherwise it is 0.

This means that a secure implementation of modular exponentiation must be computed through a regular sequence of squarings and multiplications uncorrelated to the key bits.

### *A. Non-constant time regular exponentiation*

We review in this subsection two methods which perform an exponentiation through a regular sequence of operation (squarings or multiplications). The first one is the multiply-always approach which performs all the squarings in Algorithm 1 as they were multiplication with distinct operands [5]. This approach is shown in Algorithm 2 and its cost is in average $\frac{3\ell}{2}$ multiplications.

This multiply-always approach can be threaten by the attack of [4]: this attack differentiates a power trace of a multiplication $r \times r$ (i.e. a hidden square) by a multiplication $r \times x$ with $x \neq r$ based on a difference of the Hamming weight of the output bits. To overcome this problem the authors in [5] use the fact that a multiplication can be performed with two squarings:

$$r \times x = \frac{(r+x)^2 - (r-x)^2}{4}. \tag{1}$$

They could then re-express all the multiplications of the square-and-multiply exponentiation in order to get a square-always exponentiation. This leads to Algorithm 3 which has a complexity of $2\ell$ squarings in average.

**Algorithm 2** Multiply-always [5]

**Require:** $x \in \{0, \ldots, N-1\}$ and $K = (k_{\ell-1}, \ldots, k_0)_2$

1: $r \leftarrow 1$
2: **for** $i$ **from** $\ell - 1$ **downto** $0$ **do**
3: $\quad r \leftarrow r \times r$
4: $\quad$ **if** $k_i = 1$ **then**
5: $\quad\quad r \leftarrow r \times x$
6: $\quad$ **end if**
7: **end for**
8: **return** $r$

**Algorithm 3** Square-always [5]

**Require:** $x \in \{0, \ldots, N-1\}$ and $K = (k_{\ell-1}, \ldots, k_0)_2$

1: $r \leftarrow 1$
2: **for** $i$ **from** $\ell - 1$ **downto** $0$ **do**
3: $\quad r \leftarrow r^2$
4: $\quad$ **if** $k_i = 1$ **then**
5: $\quad\quad r \leftarrow \frac{(r+x)^2-(r-x)^2}{4}$
6: $\quad$ **end if**
7: **end for**
8: **return** $r$

Both multiply-always and square-always approaches suffer from a weakness: they do not process the exponentiation with a constant time. In terms of side channel analysis this means that the time of the computation leaks some information of the key: its Hamming weight. In the next subsection we review two approaches which are regular and also constant time.

### *B. Constant time regular exponentiation*

The first method which satisfies this property is the square-and-multiply-always exponentiation proposed by Coron in [6]. The idea of Coron is to perform a dummy multiplication when we read a bit which is equal to 0. This results in a power trace of a regular sequence of traces of squarings always followed by a trace of a multiplication. This method is given in Algorithm 4.

The square-and-multiply-always exponentiation is effective to counteract SPA and SEMA along with timing attacks. But it is still under the threat of another kind of side channel attack: the fault injection attack [11], [12]. The idea of this attack is to inject an error during the $i$-th loop of the square-and-multiply-always algorithm. If the error is injected during a dummy multiplication it will not affect the final result and it would reveal a bit $k_i$ equal

**Algorithm 4** Square-and-multiply-always [6]

**Require:** $x \in \{0, \ldots, N-1\}$ and $K = (k_{\ell-1}, \ldots, k_0)_2$

1: $r \leftarrow 1$
2: **for** $i$ **from** $\ell - 1$ **downto** $0$ **do**
3: $\quad r \leftarrow r^2$
4: $\quad$ **if** $k_i = 0$ **then**
5: $\quad\quad r' \leftarrow r \times x$
6: $\quad$ **else**
7: $\quad\quad r \leftarrow r \times x$
8: $\quad$ **end if**
9: **end for**
10: **return** $r$

to zero, otherwise the result will be erroneous and this will reveal a bit $k_i$ equal to one.

This problem was fixed by the Montgomery-ladder approach (Algorithm 5) for modular exponentiation [7]. In this method there are two integers $r_0$ and $r_1$ where $r_0$ contains the same value as $r$ in the square-and-multiply algorithm, and $r_1$ satisfies $r_1 = r_0 \times x \mod N$ during the whole computation. At each loop iteration we always perform a multiplication $r_{1-k_i} \leftarrow r_1 \times r_0$ and a squaring $r_{k_i} \leftarrow r_{k_i}^2$ depending on the value of the current scanned bit $k_i$. The algorithm is regular: we have for each bit a multiplication and a squaring. It also satisfies the important property that any error injected in any intermediate value would affect the final results. This renders the error injection attack ineffective.

**Algorithm 5** Montgomery-ladder [7]

**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: $\quad$ **if** $k_i = 0$ **then**
5: $\quad\quad r_0 \leftarrow r_0^2$
6: $\quad\quad r_1 \leftarrow r_1 \times r_0$
7: $\quad$ **else**
8: $\quad\quad r_0 \leftarrow r_0 \times r_1$
9: $\quad\quad r_1 \leftarrow r_1^2$
10: $\quad$ **end if**
11: **end for**
12: **return** $r_0$

Both square-and-multiply-always and Montgomery-ladder exponentiations have a complexity equal to $\ell$ squarings and $\ell$ multiplications for an $\ell$-bit integer $K$.

**Remark 1.** There are some alternative methods in the literature insuring a regularity of the operation while reducing the number of multiplications. This is for example the case of the methods reported in [13] which use a regular windowing recoding of the exponent $K$. The drawback of those methods is that they require additional resources to store some precomputed data. In this paper we focus on methods which require at most one or two intermediate variables, and are thus suitable for embedded devices with limited resources and thus the most susceptible to be attack by side channel analysis.

### *C. Multiplicative splitting of an integer $x$ modulo $N$*

We consider an RSA modulus $N$ and an integer $x \in [0, N]$ which corresponds to the message we want to decrypt or sign by computing $x^K \mod N$. We will show in this section that $x$ can be split into two parts as follows

$$x = x_0^{-1} \times x_1 \mod N \text{ with } |x_0|, |x_1| \leq \lceil N^{1/2} \rceil. \tag{2}$$

In order to get a multiplicative splitting of $x$ modulo $N$, we use the method presented in [14] which consists in a partial execution of the extended Euclidean algorithm. The Euclidean algorithm computes the greatest common divisor of $x$ and $N$ through a sequence of reductions: we start with $r_0 = N, r_1 = x$ and perform the following iteration

$$r_{i+1} = r_{i-1} \mod r_i \quad \text{for } i = 1, 2, \ldots. \tag{3}$$

The sequence $r_0, r_1, \ldots, r_i$ is a decreasing sequence of positive integers and the last non zero $r_i$ satisfies $r_i = \gcd(x, N)$.

The extended Euclidean algorithm computes, in addition to $\gcd(x, N)$, two integers $a, b$ satisfying

$$ax + bN = \gcd(x, N), \tag{4}$$

which is called a Bezout identity. In order to compute $a$ and $b$ the extended Euclidean algorithm maintains two sequences $a_i$ and $b_i$ satisfying

$$a_i x + b_i N = r_i \tag{5}$$

where the integers $r_i, i = 0, 1, \ldots$, are the consecutive remainders in (3) computed in the Euclidean algorithm. The integers $a_i, b_i, i = 1, 2, \ldots$, are computed as follows

$$\begin{aligned} q_i &= \lfloor r_{i-1}/r_i \rfloor, \\ r_{i+1} &= r_{i-1} - q_i r_i, \\ a_{i+1} &= a_{i-1} - q_i a_i, \\ b_{i+1} &= b_{i-1} - q_i b_i, \end{aligned} \tag{6}$$

starting from $r_0 = N, r_1 = x$ and $a_0 = 0, a_1 = 1$ and $b_0 = 1, b_1 = 0$. Then, when $r_i$ is equal to $\gcd(x, N)$ the identity (5) is a valid Bezout relation (4). For a detailed presentation of this method the reader may refer to [15].

In order to obtain a multiplicative splitting of $x$, the authors in [14] stop the extended Euclidean algorithm when $r_i \cong N^{1/2}$ and $a_i \cong N^{1/2}$: indeed, due to (5), for any $i$ we have $x = a_i^{-1} r_i \mod N$. This method to compute the splitting of an integer $x$ is reviewed in Algorithm 6.

**Algorithm 6** Multiplicative splitting modulo $N$ [14]

**Require:** $0 \leq x < N < c^2 \in \mathbb{N}$ with $\gcd(x, N) < c$.

**Ensure:** $x_0$ and $x_1$ such that $x = x_0^{-1} x_1 \mod N$ and $|x_0|, |x_1| < c$.

1: $a_0 \leftarrow 0; a_1 \leftarrow x; r_0 \leftarrow N; r_1 \leftarrow x, i \leftarrow 1$
2: **while** $|r_i| \geq c$ **do**
3: $\quad q_i \leftarrow \lfloor r_{i-1}/r_i \rfloor$
4: $\quad r_{i+1} \leftarrow r_{i-1} - q_i r_i$
5: $\quad a_{i+1} \leftarrow a_{i-1} - q_i a_i$
6: $\quad i \leftarrow i + 1$
7: **end while**
8: **return** $a_i, r_i$

The following lemma asserts that the output $a_i$ and $r_i$ satisfy $|a_i|, |r_i| < c$.

**Lemma 1.** *Let $c \in \mathbb{N}$ such that $c > N^{1/2}$ and let $a_0, a_1, \ldots, a_i, \ldots$ and $r_0, r_1, \ldots, r_i, \ldots$ be the sequences computed in Algorithm 6. Then Algorithm 6 correctly outputs a pair $a_{i_c}, r_{i_c}$ such that*

$$x = a_{i_c}^{-1} \times r_{i_c} \mod N \text{ with } |a_{i_c}| < c \text{ and } |r_{i_c}| < c.$$

*Proof.* The proof is a direct extension of [14]. A well known property on extended Euclidean algorithm (cf. Chapter 3 in [15]) provides that, for $i \geq 1$, we have $|a_i| < |a_{i+1}|$ and $r_i > 0$ and also that

$$r_{i-1}|a_i| + r_i|a_{i-1}| = N. \tag{7}$$

So if $r_{i_c}$ is the first remainder such that $r_{i_c} < c$ we have $r_{i_c+1} \geq c > \sqrt{N}$. Then taking $i = i_c + 1$ in (7) we have $r_{i_c}|a_{i_c+1}| + r_{i_c+1}|a_{i_c}| = N$ then one must have $a_{i_c} < N/r_{i_c+1} < N/c < c$. □

If Algorithm 6 is executed with $c = \lceil N^{1/2} \rceil$ then the multiplicative splitting $a_{i_c}, r_{i_c}$ output by the algorithm satisfies

$$|a_{i_c}| < \lceil N^{1/2} \rceil \text{ and } |r_{i_c}| < \lceil N^{1/2} \rceil.$$

In other words, it is a half-size splitting.

**Complexity.** For the sake of simplicity we will only give an upper bound of cost a the multiplicative splitting. Specifically, since computing a multiplicative splitting consists in a partial execution of the extended Euclidean algorithm, we can bound above its cost with an upper bound of the complexity of an extended Euclidean algorithm. We use the following lemma inspired from [15].

**Lemma 2** (Complexity of the extended Euclidean algorithm)**.** *The extended Euclidean algorithm (i.e. Algorithm 6 with $c = 1$), with two positive integers $a \leq b$ of $w$-bit word length $t$ as input, requires at most $4wt^2$ word additions.*

*Proof.* We will consider a modified version of Algorithm 6: we assume that the quotients $q_i$ are of the form $q_i = 2^{\alpha_i}$. In other words, we expand the Euclidean division through several shift and subtraction operations. In this case, if we assume that the integers $a_i$ and $r_i$ in Algorithm 6 are stored on $t$ words, each loop turns requires $2t$ words subtractions. Furthermore we have the following:

$$r_i = r_{i-1}q_i + r_{i-2} \geq r_{i-1} + r_{i-2} > 2r_{i-2}$$

since $q_i \geq 1$ and $r_{i-1} > r_{i-2}$. This implies that $r_i < \frac{r_0}{2^{i/2}} = \frac{b}{2^{i/2}}$ and consequently the number of loop iterations before we get $r_i = 0$ is at most $2\log_2(b) \leq 2tw$. Then at the end the total number of operations is at most $2tw \times 2t = 4t^2w$ word subtractions. □

## III. Regular exponentiation with half-size multiplicative splitting

Given a multiplicative splitting (2) of $x$ into two half-size integers, we can modify the square-and-multiply method in order to distribute a full multiplication by $x$ to one half-size multiplication by $x_0$ when $k_i = 0$ and one half-size multiplication by $x_1$ when $k_i = 1$. This approach is depicted in Algorithm 7. This algorithm reaches our goal since it is regular: each loop iteration is a squaring followed by a half-size multiplication. It is also robust against fault injection attack: each error in one half-size multiplication will affect the final result.

**Algorithm 7** Regular exponentiation with half-size multiplications

**Require:** $x \in \{0, \ldots, N-1\}$ and $K = (k_{\ell-1}, \ldots, k_0)_2$

**Ensure:** $r = x^k \mod N$

1: `Split.` $x = x_0^{-1} \times x_1 \mod N$ with $x_0, x_1 \cong N^{1/2}$.
2: $r \leftarrow x_0^{-1}$
3: **for** $i$ **from** $\ell - 1$ **downto** $0$ **do**
4: $\quad r \leftarrow r^2 \mod N$
5: $\quad$ **if** $k_i = 0$ **then**
6: $\quad\quad r = r \times x_0$
7: $\quad$ **else**
8: $\quad\quad r = r \times x_1$
9: $\quad$ **end if**
10: **end for**
11: $r \leftarrow r \times x_0$
12: **return** $r$

The following lemma establishes the validity of Algorithm 7, i.e., that it correctly computes $r = x^K \mod N$.

**Lemma 3.** *Let $K = (k_{\ell-1}, \ldots, k_0)_2$ with $k_i \in \{0, 1\}$ be an $\ell$ bit integer and let $N$ and $x$ be two positive integers such that $x < N$. If we set $K_i = (k_{\ell-1}, \ldots, k_i)_2$, then the value of $r$ after loop $i$ satisfies:*

$$r = x^{K_i} x_0^{-1} \mod N.$$

*Proof.* We prove the assertion by a decreasing induction on $i$: we assume it is true for $i$ and we prove it for $i-1$. By induction hypothesis, $r_i$ the value of $r$ after the execution of loop $i$ in Algorithm 7 satisfies $r_i = x^{K_i} \times x_0^{-1}$. Now if $k_{i-1} = 1$ the execution of loop $i-1$ gives:

$$\begin{aligned} r_{i-1} &= r_i^2 \times x_1 \\ &= x^{2K_i} \times x_0^{-2} \times x_1 \\ &= x^{2K_i+1} \times x_0^{-1} \\ &= x^{K_{i-1}} \times x_0^{-1}. \end{aligned}$$

And, if $k_{i-1} = 0$, the execution of loop $i-1$ gives:

$$\begin{aligned} r_{i-1} &= r_i^2 \times x_0 \\ &= x^{2K_i} \times x_0^{-2} \times x_0 \\ &= x^{2K_i} \times x_0^{-1} \\ &= x^{K_{i-1}} \times x_0^{-1}. \end{aligned}$$

□

## IV. Exponentiation with half-size splitting and Montgomery multiplication

An RSA modulus $N$ looks like a random integer: it has not a sparse binary representation and has no other underlying structure which can be used to speed-up a reduction modulo $N$. The most used method to perform a multiplication modulo a random integer is the Montgomery method [8]. We modify Algorithm 7 in order to use the Montgomery multiplication for the squarings and multiplications modulo $N$. A squaring in Algorithm 7 involves integers of size $\lceil \log_2(N) \rceil$ bits while a multiplication involves two kinds of multiplicands: one integer of size $\lceil \log_2(N) \rceil$ bits and one integer of size $\cong \lceil \log_2(N)/2 \rceil$ bits. This pushes us to use two kinds of Montgomery multiplications:

- *Full Montgomery Multiplication (FMM)*: Let $M$ be an integer such that $M > N$ and $\gcd(N, M) = 1$ . Let $y$ and $x$ be two $\lceil \log_2(N) \rceil$ bit integers. Then the FMM works as follows:

  $q \leftarrow (-x \times y \times N^{-1}) \mod M$
  $z \leftarrow (x \times y + q \times N)/M$

  and $z$ satisfies $z = (xyM^{-1}) \mod N$ and $z < 2N$. In practice taking $M = 2^{n+1}$ with $n = \lceil \log_2(N) \rceil$ simplifies the reduction and the division by $M$. This method also applies for a squaring, i.e., $x = y$ and, in the sequel this will be referred to as FMS for Full Montgomery Squaring.
- *Half Montgomery Multiplication (HMM)*: Let $m$ be an integer such that $m > \sqrt{N}$ and $\gcd(N, m) = 1$. Let $y$ be a $\lceil \log_2(N) \rceil$ bit integer and $x$ be a $\lceil \log_2(N)/2 \rceil$ bit integer. Then the HMM works as follows:

$q \leftarrow (-x \times y \times N^{-1}) \mod m$

$z \leftarrow (x \times y + q \times N)/m$

and $z$ satisfies $z = (xym^{-1}) \mod N$ and $z < 2N$. Then, in practice, taking $m = 2^{\lceil n/2 \rceil + 1}$ where $n = \lceil \log_2(N) \rceil$ simplifies the computation of a reduction and a division by $m$.

The proposed regular exponentiation which incorporates FMS and HMM is depicted in Algorithm 8.

**Algorithm 8** Regular exponentiation with half-size Montgomery modular multiplications

**Require:** $x \in \{0, \ldots, N-1\}$ and $K = (k_{\ell-1}, \ldots, k_0)_2$

**Ensure:** $r = x^K \mod N$

1: Split $x = x_0^{-1} \times x_1 \mod N$ with $x_0, x_1 \cong N^{1/2}$.
2: $r = x_0^{-1} \times m \times M \mod N$ *// Montgomery representation*
3: **for** $i$ **from** $\ell - 1$ **downto** $0$ **do**
4: $r \leftarrow FMS(r, r)$
5: **if** $k_i = 0$ **then**
6: $r \leftarrow HMM(r, x_0)$
7: **else**
8: $r \leftarrow HMM(r, x_1)$
9: **end if**
10: **end for**
11: $r \leftarrow \left(r \times x_0 \times m^{-1} \times M^{-1}\right) \mod N$
12: **return** $r$

**Lemma 4.** *Let $K = (k_{\ell-1}, \ldots, k_0)_2$ with $k_i \in \{0, 1\}$ be an $\ell$ bit integer, and let $N$ be a positive integer and $x \in [0, N-1]$. If we set $K_i = (k_{\ell-1}, \ldots, k_i)_2$ then the value $r$ after the loop $i$ in Algorithm 8 satisfies:*

$$r = (x^{K_i} x_0^{-1} M m) \mod N.$$

*Proof.* We prove it by induction on $i$. If we denote $r_i$ the value of $r$ after the loop $i$, then it satisfies $r_i = (x^{K_i} x_0^{-1} M m) \mod N$. Then the squaring with FMS provides:

$$\begin{aligned} \text{FMS}(r_i) &= x^{2K_i} x_0^{-2} M^2 m^2 M^{-1} \mod N \\ &= x^{2K_i} x_0^{-2} M m^2 \mod N. \end{aligned}$$

Now if $k_{i-1} = 0$ then the algorithm computes:

$$\begin{aligned} r_{i-1} &= \text{HMM}(x^{2K_i} x_0^{-2} M m^2, x_0) \\ &= (x^{2K_i} x_0^{-2} M m^2) x_0 m^{-1} \mod N \\ &= x^{2K_i} x_0^{-1} M m \mod N \end{aligned}$$

which satisfies the induction hypothesis since $K_{i-1} = 2K_i$. Now if $k_{i-1} = 1$ the algorithm computes:

$$\begin{aligned} r_{i-1} &= \text{HMM}(x^{2K_i}x_0^{-2}Mm^2, x_1) \\ &= (x^{2K_i}x_0^{-2}Mm^2)x_1m^{-1} \mod N \\ &= x^{2K_i+1}x_0^{-1}Mm \mod N \end{aligned}$$

which satisfies the induction hypothesis since $K_{i-1} = 2K_i + 1$. □

## V. Complexity comparison and security evaluation

In this section we first briefly the review word-level forms of Montgomery multiplication and squaring along with their complexities. We then deduce the complexity of the proposed exponentiation and compare it with the approaches reviewed in Section II.

### *A. Word level Montgomery multiplication and squaring*

The proposed exponentiation in Algorithm 8 involves Montgomery modular squarings and multiplications with adapted sizes to the operands, i.e., of size either $\lceil \log_2(N) \rceil$ or $\lceil \log_2(N)/2 \rceil$ bits. The subsequent word level form of Montgomery multiplication can take as input two integers of different sizes.

**Word-level Montgomery multiplication.** We consider two integers $x = (x_{t-1}, \ldots, x_0)_{2^w}$ where $t = \lceil N/2^w \rceil$ and $y = (y_{s-1}, \ldots, y_0)_{2^w}$ with $s = t$ or $s = \lceil t/2 \rceil$. The word level form of the Montgomery multiplication interleaves multi-precision multiplication and small Montgomery reduction by sequentially performing for $i = 0, 1, \ldots, s-1$:

$$\begin{aligned} z &\leftarrow z + x \times y_i \\ q &\leftarrow -z \times N^{-1} \mod 2^w \\ z &\leftarrow (z + qN)/2^w \end{aligned}$$

where $z$ is initially set to 0 and, at the end, it is equal to $x \times y \times 2^{-sw} \mod N$. This method is detailed in Algorithm 9.

The complexity of Algorithm 9 is evaluated step by step in Table II. The cost of each step is expressed in terms of the complexity of a $t$-word addition or of a $1 \times t$ multiplication which costs $t$ word multiplications and $t$ word additions with carry.

**Word level Montgomery squaring.** The Montgomery squaring of a $t$-word integer $x$ can be computed with the word-level Montgomery multiplication. However, a squaring can be optimized by considering that we may save some redundant word multiplications $x_i \cdot x_j$ and $x_j \cdot x_i$. We review here the formulation of the Montgomery squaring provided in [9]. The squaring $x^2$ is rewritten as follows:

$$\begin{aligned} x^2 &= \sum_{i=0}^{t-1}\sum_{j=0}^{t-1} x_i x_j 2^{w(i+j)} \\ &= 2\sum_{i=0}^{t-2}\sum_{j=i+1}^{t-1} x_i x_j 2^{w(i+j)} + \sum_{i=0}^{t-1} x_i^2 2^{2iw} \\ &= \sum_{i=0}^{t-1} x_i 2^{w(2i)}(x_i + 2\sum_{j=1}^{t-i-1} x_{i+j} 2^{wj}) \\ &= \sum_{i=0}^{t-1} x_i 2^{w(2i)} \widetilde{x}_i. \end{aligned} \tag{8}$$

**Algorithm 9** Word level Montgomery multiplication [16]

**Require:** $N < 2^{wt-2}$ the modulus, $w$ the word size, $x = (x_{t-1}, \ldots, x_0)_{2^w}$ and $y = (y_{s-1}, \ldots, y_0)_{2^w}$ integers in $[0, N]$ and $N' = (-N^{-1}) \mod 2^w$

**Ensure:** $z = x \cdot y \cdot 2^{-ws} \mod N$

1: $z \leftarrow 0$
2: **for** $i = 0$ to $s - 1$ **do**
3: $\quad z \leftarrow z + y_i \cdot x$
4: $\quad q \leftarrow |z|_{2^w} \cdot N' \mod 2^w$
5: $\quad z \leftarrow (z + q \cdot N)/2^w$
6: **end for**
7: **if** $z \geq N$ **then**
8: $\quad z \leftarrow z - N$
9: **end if**
10: **return** $z$

Table II
STEP BY STEP COMPLEXITY EVALUATION OF WORD LEVEL MONTGOMERY MULTIPLICATION ( ALGORITHM 9)

| | Operations | # word add. | # word mul. |
|---|---|---|---|
| $s$ Step 3 | $x_i \times y$ | $st$ | $st$ |
| | $z + (x_i y)$ | $s(t+1)$ | 0 |
| $s$ Step 4 | $\lvert z\rvert_{2^w} \cdot N'$ | 0 | $s$ |
| $s$ Step 5 | $q \times N$ | $st$ | $st$ |
| | $z + (qN)$ | $s(t+1)$ | 0 |
| Step 7 | $z - N$ | $t$ | 0 |
| Total | | $s(4t+2)+t$ | $s(2t+1)$ |

The integer $\widetilde{x}_i = (x_i + 2\sum_{j=1}^{t-i-1} x_{i+j} 2^{wj})$ can be deduced from $x' = 2x = (x'_{t-1}, \ldots, x'_0)_{2^w}$ as

$$\widetilde{x}_i = (x'_{t-1}, \ldots, x'_{i+2}, |2x_{i+1}|_{2^w}, x_i)_{2^w}.$$

With the formulation (8) the authors in [9] could derive a word level Montgomery squaring as shown in Algorithm 10.

The complexity of Algorithm 10 is evaluated step by step in Table III. Only the complexity evaluation of Step 5 needs to be detailed. We first notice that:

- $\widetilde{x}_i \times x_i$ requires $t - i$ word multiplications and $t - i$ word additions.
- $z + 2^{wi}(\widetilde{x}_i x_i)$ requires $t - i + 1$ word additions.

We add the contributions of all loop iterations and we get $\sum_{i=0}^{t-1}(t-i) = \frac{t(t+1)}{2}$ word multiplications and $\sum_{i=0}^{t-1}(2t - 2i + 1) = t(t+1) + t = t^2 + 2t$ word additions for $t$ Step 5, as stated in Table III.

**Algorithm 10** Word level Montgomery squaring [9]

**Require:** $N < 2^{wt-2}$ the modulus, $x$, with $x = (x_{t-1}, \ldots, x_0)_{2^w}$ with $0 \leq x_i < 2^w$ where $w$ is the word size, $N' = -N^{-1} \mod 2^w$

**Ensure:** $z \equiv x^2 \times 2^{-wt} \mod N$ and $z < N$

1: $x' \leftarrow x + x$
2: $z \leftarrow 0$
3: **for** $i = 0$ to $(t-1)$ **do**
4: $\quad \widetilde{x}_i \leftarrow (x'_{t-1}, \ldots, x'_{i+2}, |2x_{i+1}|_{2^w}, x_i)_{2^w}$
5: $\quad z \leftarrow z + \widetilde{x}_i \cdot x_i \cdot 2^{wi}$
6: $\quad q \leftarrow |z|_{2^w} \cdot N' \mod 2^w$
7: $\quad z \leftarrow (z + q \cdot N)/2^w$
8: **end for**
9: **if** $z \geq N$ **then**
10: $\quad z \leftarrow z - N$
11: **end if**
12: **return** $z$

Table III
STEP BY STEP COMPLEXITY EVALUATION OF A WORD LEVEL MONTGOMERY SQUARING (ALGORITHM 9)

| | Operations | # word add. | # word mul. |
|---|---|---|---|
| Step 1 | $x + x$ | $t$ | 0 |
| $t$ Step 4 | $\|2x_{i+1}\|_{2^w}$ | $t$ | 0 |
| $t$ Step 5 | $z + 2^{wi}\widetilde{x}_i x_i$ | $t^2 + 2t$ | $\frac{t(t+1)}{2}$ |
| $t$ Step 6 | $\|z\|_{2^w} \cdot N'$ | 0 | $t$ |
| $t$ Step 7 | $q \times N$ | $t^2$ | $t^2$ |
| | $z + (qN)$ | $t(t+1)$ | 0 |
| Step 10 | $z - N$ | $t$ | 0 |
| Total | | $3t^2 + 6t$ | $\frac{3t^2}{2} + \frac{3t}{2}$ |

### *B. Complexity comparison*

Now, we can deduce the cost of a FMM, FMS and HMM from the complexity of the word-level Montgomery multiplication and squarings. Specifically, the cost of a FMS with $M = 2^{tw}$ is the same as the one shown in Table III. To obtain the complexity of FMM with $M = 2^{tw}$ we take $s = t$ in the formula of Table II and to get the complexity of a HMM with $m = 2^{tw/2}$ we take $s = t/2$ in the formula of Table II. This leads to the complexities shown in the upper part of Table IV.

Now, we deduce the cost of the following approaches for an $\ell$ bit exponent for a modular exponentiation:

- The square-and-multiplication exponentiation requires $\ell$ FMS and $\ell/2$ FMM in average.
- The multiply-always exponentiation necessitates $3\ell/2$ FMM in average.
- The square-always exponentiation necessitates $2\ell$ FMS in average.
- The square-and-multiply-always and Montgomery-ladder exponentiation require $\ell$ FMS and $\ell$ FMM.
- The Montgomery-ladder exponentiation with common multiplicand [9]: this necessitates $\ell$ word level combined Montgomery multiplications $AB, AC$, which have a reduced complexity by sharing some of the reduction computations.

The complexities of these approaches in terms of the number of word additions and multiplications are given in Table IV.

For the proposed regular exponentiation with half size Montgomery multiplication (Algorithm 8) we need $\ell$ FMS and $\ell$ HMM for the main loop computation. For the computation of the multiplicative splitting $x_0$ with Algorithm 6, the cost is, using Lemma 2, bounded above by $4t^2w$ word additions. The computation of $x_0^{-1}$ has also a cost bounded above by $4t^2w$ word additions since it is computed with the extended Euclidean algorithm. The resulting overall complexity of the proposed regular exponentiation is given in terms of the number of word additions and multiplications in Table IV.

Table IV
COMPLEXITY COMPARISON

| | Algorithm | #word add. | #word mul. |
|---|---|---|---|
| Multiplication and squaring modulo $N$ | FMM | $4t^2+3t$ | $2t^2+t$ |
| | FMS | $3t^2+6t$ | $\frac{3t^2}{2}+\frac{3t}{2}$ |
| | HMM | $2t^2+2t$ | $t^2+\frac{t}{2}$ |
| Exponentiation mod $N$ with no side channel protection | Square-and-multiply | $\ell(5t^2+\frac{15t}{2})+8t^2+6t$ | $\ell(\frac{5t^2}{2}+\frac{4t}{2})+4t^2+2t$ |
| Non constant time regular exponentiation | Multiply-always | $\ell(6t^2+\frac{9t}{2})+8t^2+6t$ | $\ell(3t^2+\frac{3t}{2})+4t^2+2t$ |
| | Square-always | $\ell(6t^2+12t)+8t^2+6t$ | $\ell(3t^2+3t)+4t^2+2t$ |
| Regular and constant time exponentiation | Square-and-multiply-always | $\ell(7t^2+9t)+8t^2+6t$ | $\ell(\frac{7t^2}{2}+\frac{5t}{2})+4t^2+2t$ |
| | Montgomery-ladder | $\ell(7t^2+9t)+8t^2+6t$ | $\ell(\frac{7t^2}{2}+\frac{5t}{2})+4t^2+2t$ |
| | Montgomery-ladder CM [9] | $\ell(6t^2+9t+1)+8t^2+8t$ | $\ell(3t^2+4t+3)+4t^2+4t+2$ |
| | Proposed (Algorithm 8) | $\ell(5t^2+8t)+10t^2+8t$ | $\ell(\frac{5t^2}{2}+2t)+8wt^2+5t^2+\frac{5t}{2}$ |

We notice that the fastest approach is the non-secure square-and-multiply exponentiation. We also notice that our approach has complexity really close to the one of the square-and-multiply: only the precomputation costs make it less efficient. Moreover, our approach is better by roughly 16% than the regular but non constant time approaches, i.e. square-always and multiply-always and also to constant time and regular Montgomery-ladder with CM.

### C. Implementation results

We have implemented the different approaches on an Intel Core i5 with C language and compiled with gcc-4.8.6. For modular multiplication and modular squaring we implemented Algorithm 10 and Algorithm 9 using low level functions of GMP library (cf. `GMP 6.0.0, https://gmplib.org`) for $1 \times t$ multiplications and $t$-word additions. We could then implement all the exponentiation algorithms considered in this paper. The multiplicative splitting of our approach is implemented using the low level function of gmp for Euclidean division. The timings obtained for a few different practical bit lengths of $N$ (i.e., 1020, 2040, 3050 and 4090) are reported in Table V.

Table V
TIMINGS IN $10^3$ CLOCK-CYCLES OF MODULAR EXPONENTIATION

| | Algorithm | Timings | | |
|---|---|---|---|---|
| | | 2040bits | 3070bits | 4090bits |
| Exponentiation without side channel protection | Square-and-multiply | 14201 | 46287 | 105065 |
| Non constant time regular exponentiations | Multiply-always [5] | 16178 | 52171 | 121952 |
| | Square-always [5] | 18020 | 58766 | 131297 |
| Regular and constant time exponentiations | Montgomery-ladder [7] | 21803 | 70389 | 164010 |
| | Montgomery-ladder with CM [9] | 18124 | 57401 | 129847 |
| | Square-and-multiply-always [6] | 19738 | 63292 | 146842 |
| | Proposed (Algorithm 8) | 15203 | 47540 | 108607 |

We notice that the reported timings relate closely to the complexity results shown in Table IV. Indeed, the fastest approach is the square-and-multiply exponentiation which is not protected against simple side channel analysis. Our approach is less than 7% slower than square-and-multiply for any key size but become close to 3% for 4090 bits. But it is better than all other approaches: by $6\% - 11\%$ compared to the multiply-always approach, which is not entirely secure against SPA, and more than 16% compared to all other approaches.

### D. Security evaluation

The proposed exponentiation algorithm prevent a simple power analysis (SPA) or a simple electromagnetic analysis (SEMA). We discuss here some additional features in order to have a full protection of the secret exponent against differential power analysis (DPA) [2]. This attack exploits the lack of randomness in the exponentiation. In the exponentiation algorithms considered in this paper, the value taken by $r$ in the $i$-th loop depends on $x$ and on the key bits of $k_\ell, k_{\ell-1}, \ldots, k_i$ of $K$. DPA uses the fact that if we can predict the next bit $k_{i-1}$ we can predict the next value of $r_{i-1}$ in the loop $i-1$. With this prediction we also predict the power consumption of the next loop since it is generally proportional to the Hamming weight of $r_{i-1}$. In a DPA analysis, averaging over many power traces reveals if the guess is correct (a peak appears) or not and thus reveals the value of $k_{i-1}$.

The main strategies for protecting an implementation against this DPA attack are as follows:

- *Randomization of the exponent $K$ [6], [17], [18].* This leads to unpredictable values taken by $r$ during the exponentiation. Different strategies have been proposed: the first one add to $k$ a random multiple of $\phi(N) = (p-1)(q-1)$

$$K' = K + \beta \times \phi(N)$$

with $\beta$ a random integer generally taken in $[0, 2^{20}]$. Another method consists to randomly chose $\beta \in [0, 2^{20}]$ coprime with $\phi(N)$ and compute $\beta^{-1} \mod \phi(N)$. The value of $K$ is then randomized as

$$K' = K \times \beta^{-1} \mod N.$$

The exponentiation is performed in two steps: we first compute $r' = x^{K'} \mod N$ and then the final result $r = r'^{\beta} \mod N = x^K \mod N$.

- *Blinding of the message [6].* The idea is to mask $x$ and thus makes it impossible to predict anything regarding the power trace related to $x$. We choose a random value $\rho$ and we compute

$$x' = x \times \rho^{K'} \mod N$$

where $K'$ is the public exponent. The exponentiation $\rho^{K'} \mod N$ is effective when $K'$ is small which is often the case in practice. The exponentiation algorithm then computes

$$\begin{aligned} r' &= x'^K \mod N \\ &= x^K \rho^{KK'} \mod N \\ &= x^K \rho \mod N. \end{aligned}$$

We get the final result $x^K \mod N$ by multiplying $r'$ by $\rho^{-1}$ modulo $N$.

The above strategies can be used in combination of the proposed regular exponentiation with half-size multiplication. This provides an RSA exponentiation protected against the following set of side channel attacks: SPA, SEMA, DPA, ZPA [19] and safe-error fault-injection attack.

## VI. Conclusion

We presented in this paper a new approach for regular modular exponentiation. We first introduced a multiplicative splitting of an integer $x$ modulo $N$. We showed that this splitting can be used to modify the square-and-multiply algorithm in order to have a regular sequence of squarings always followed by a multiplication with a half-size integer. We then modified this algorithm in order to perform modular multiplication with the Montgomery's method. Compared to the usual regular and constant time modular exponentiations, the proposed method involves only multiplication by half-size integer instead of a full multiplication. This leads to a reduction of the complexity by 16%.

**Aknowledgements.** This work was supported by PAVOIS ANR 12 BS02 002 02.

## REFERENCES


[1] R. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, pp. 120–126, 1978.

[2] P. C. Kocher, J. Jaffe, and B. Jun, "Differential Power Analysis," in *Advances in Cryptology, CRYPTO'99*, ser. LNCS, vol. 1666. Springer, 1999, pp. 388–397.

[3] S. Mangard, "Exploiting Radiated Emissions - EM Attacks on Cryptographic ICs," in *Austrochip 2003, Linz, Austria, October 1st*, 2003, pp. 13–16.

[4] F. Amiel, B. Feix, M. Tunstall, C. Whelan, and W. Marnane, "Distinguishing Multiplications from Squaring Operations," in *SAC 2008*, ser. LNCS, vol. 5381. Springer, 2009, pp. 346–360.

[5] C. Clavier, B. Feix, G. Gagnerot, M. Roussellet, and V. Verneuil, "Square Always Exponentiation," in *Progress in Cryptology - INDOCRYPT 2011*, ser. LNCS, vol. 7107. Springer, 2011, pp. 40–57.

[6] J.-S. Coron, "Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems," in *CHES*, 1999, pp. 292–302.

[7] M. Joye and S. Yen, "The Montgomery Powering Ladder," in *CHES 2002*, ser. LNCS, vol. 2523. Springer, 2002, pp. 291–302.

[8] P. Montgomery, "Modular Multiplication Without Trial Division," *Math. Computation*, vol. 44, pp. 519–521, 1985.

[9] C. Negre, T. Plantard, and J. Robert, "Efficient Modular Exponentiation Based on Multiple Multiplications by a Common Operand," in *IEEE Symposium on Computer Arithmetic 2013*, to appear.

[10] A. Menezes, P. van Oorschot, and S. Vanstone, *Handbook of Applied Cryptography*. CRC Press, 1996.

[11] S.-M. Yen and M. Joye, "Checking Before Output May Not Be Enough Against Fault-Based Cryptanalysis," *IEEE Trans. Computers*, vol. 49, no. 9, pp. 967–970, 2000.

[12] S.-M. Yen, S. Kim, S. Lim, and S.-J. Moon, "A Countermeasure against One Physical Cryptanalysis May Benefit Another Attack," in *ICISC 2001*, ser. LNCS, vol. 2288. Springer, 2001, pp. 414–427.

[13] M. Joye and M. Tunstall, "Exponent Recoding and Regular Exponentiation Algorithms," in *Progress in Cryptology - AFRICACRYPT 2009*, ser. LNCS, vol. 5580. Springer, 2009, pp. 334–349.

[14] R. Gallant, R. Lambert, and S. Vanstone, "Faster point multiplication on elliptic curves with efficient endomorphisms," in *Advances in Cryptology - CRYPTO 2001*, ser. LNCS, vol. 2139. Springer, 2001, pp. 190–200.

[15] J. von zur Gathen and J. Gerhard, *Modern Computer Algebra (3. ed.)*. Cambridge University Press, 2013.

[16] A. Bosselaers, R. Govaerts, and J. Vandewalle, "Comparison of Three Modular Reduction Functions," in *Advances in Cryptology - CRYPTO'93*, ser. LNCS, vol. 773. Springer, 1993, pp. 175–186.

[17] M. Ciet and M. Joye, "(Virtually) Free Randomization Techniques for Elliptic Curve Cryptography," in *ICICS 2003*, ser. LNCS, vol. 2836. Springer, 2003, pp. 348–359.

[18] M. Tunstall and M. Joye, "Coordinate blinding over large prime fields," in *CHES 2010*, ser. LNCS, vol. 6225. Springer, 2010, pp. 443–455.

[19] L. Goubin, "A Refined Power-Analysis Attack on Elliptic Curve Cryptosystems," in *PKC 2003*, ser. LNCS, vol. 2567. Springer, 2003, pp. 199–210.