J Cryptogr Eng (2017) 7:245–253
DOI 10.1007/s13389-016-0134-5

SHORT COMMUNICATION

Efficient regular modular exponentiation using multiplicative half-size splitting

Christophe Negre[1,2] · **Thomas Plantard**[3,4]

Received: 14 August 2015 / Accepted: 23 June 2016 / Published online: 13 July 2016

Abstract In this paper, we consider efficient RSA modular exponentiations $x^K \mod N$ which are regular and constant time. We first review the multiplicative splitting of an 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 by more than 12 % compared approaches which are both regular and constant time.

Keywords RSA · Regular exponentiation · Constant time exponentiation · Multiplicative splitting

1 Introduction

Currently, RSA [1] is 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 security level, N and K are chosen large enough to render the factorization of N infeasible: they are typically 2048-bit integers. The basic approach to efficiently perform 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 sufficiently, 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 exponentiations 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 noticed 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 with the Hamming weight of the exponent, which is then leaked out.

✉ Christophe Negre
christophe.negre@univ-perp.fr

Thomas Plantard
thomaspl@uow.edu.au

1 DALI, Universite de Perpignan, Perpignan, France

2 LIRMM, Université de Montpellier and National Center for Scientific Research (CNRS), 161 Rue Ada, 34090 Montpellier, France

3 Centre for Computer and Information Security Research (CCISR), University of Wollongong, Wollongong, Australia

4 School of Computer Science and Software Engineering (SCSSE), University of Wollongong, Wollongong, Australia

Table 1 Complexity in terms of word operations per iteration of loop body for a modular exponentiation

Algorithm	Regular	Constant time	Complexity per loop body for t-word integers	
			# 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	✓	✓	$5t^2 + O(t)$	$\frac{5}{2}t^2 + O(t)$

A prerequisite to be SPA resistant is then to be regular and constant time. A first method which satisfies both of these properties 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 a half-size integer. The half-size multiplications and squarings modulo N are computed with the method of Montgomery [8], and we then also provide a version of the proposed exponentiation with Montgomery modular multiplications adapted to the size of the operands. We provide a complexity analysis when modular operations are computed with word-level algorithm: computer words are w-bit long and considered integers modulo N have a size of t computer words. The complexity of the proposed approach is given in Table 1 which contains the costs of the loop bodies of the considered exponentiation algorithms. We notice that the proposed approach always reaches the best complexity while having the higher security level compared to best known methods of the literature.

The remainder of the paper is organized as follows. Section 2 summarizes state of the art methods for regular modular exponentiation. In Sect. 2.1 we review techniques to compute a multiplicative splitting of an integer modulo N. In Sect. 3 we present a new modular exponentiation algorithm which uses this splitting to render regular the square-and-multiply exponentiation. In Sect. 4, we present a version of the proposed exponentiation which incorporates Montgomery modular multiplications. Finally, in Sect. 5, we evaluate the complexity of the proposed algorithm and provide implementation results.

2 Review of regular modular exponentiation

We review in this section several methods for performing an exponentiation $x^K \mod N$. The simplest and 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 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 with the key bits. In the literature the following strategies were proposed to prevent SPA:

- *Multiply-always* [5]. This approach performs all the squarings of the square-and-multiply exponentiation as

multiplications. This leads to a sequence of $\frac{3\ell}{2}$ multiplications on average. Unfortunately, this multiply-always approach can be threatened by the attack of [4]: this attack differentiates a power trace of a multiplication $r \times r$ (i.e. a hidden squaring) by a multiplication $r \times x$ with $x \neq r$ based on a difference of the Hamming weight of the output bits.

- *Square-always* [5]. This approach is an improvement of the Multiply-always and prevents the attack of [4]. 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 re-express all the multiplications of the square-and-multiply exponentiation in order to get a square-always exponentiation. This square-always exponentiation requires 2ℓ squarings in average. Unfortunately, both multiply-always and square-always approaches suffer from a weakness: they do not process the exponentiation in a constant time, and then the Hamming weight of the key can be leaked out by the computation time.

- *Square-and-multiply-always* [6]. The first method which is regular and constant time 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 that 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.
- *Montgomery-ladder* [7]. 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 safe error fault injection attack [11,12]. This problem was fixed by the Montgomery-ladder approach for modular exponentiation [7]. The Montgomery-ladder is regular and constant time and any error injected during the computation will affect the final results, yielding a natural resilience to safe error fault injection attack.

Both square-and-multiply-always and Montgomery-ladder exponentiations have a complexity of ℓ squarings and ℓ multiplications for an ℓ-bit exponent K.

Remark 1 In this paper we focus on methods that require at most two intermediate variables. But the reader might be aware that there are some alternative methods in the literature ensuring a regularity of the operations while reducing the number of multiplications. These methods use a larger number of intermediate variable. This is for example the case of the methods reported in [13] which use a regular windowing recoding of the exponent K.

2.1 Multiplicative splitting of an integer x modulo N

We consider an RSA modulus N and an integer $x \in [0, N]$ that 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

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

The idea to split multiplicatively is not new, we can find it in a number of references of the literature: for example in [14] the authors use it to randomize an RSA exponent.

In order to get a multiplicative splitting of x modulo N, we use the method presented in [15] 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 [16].

In order to obtain a multiplicative splitting of x, the authors in [15] 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 computing the splitting of an integer x is reviewed in Algorithm 2.

Algorithm 2 Multiplicative splitting modulo N [15]

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: $a_0 \leftarrow 0; a_1 \leftarrow 1; r_0 \leftarrow N; r_1 \leftarrow x, i \leftarrow 1$
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

In order to prove the correctness of Algorithm 2, we need to recall some properties of the extended Euclidean algorithm. These properties are well known, but, since we could not find references presenting them we recall them for the sake of completeness and readability.

Lemma 1 *Let a_i and r_i be the two sequences of coefficients computed in Algorithm 2. They satisfy the following properties:*

(i) $(-1)^{i-1}a_i \geq 1$ *for all* $i \geq 1$.
(ii) $a_{i+1}r_i - a_i r_i = (-1)^i N$ *for all* $i \geq 1$.

The proof of Lemma 1 is reviewed in the Appendix.

The following lemma asserts that Algorithm 2 outputs a pair a_{i_c} and r_{i_c} which satisfy $|a_{i_c}|, |r_{i_c}| < c$.

Lemma 2 *Let $c \in \mathbb{N}$ such that $c > N^{1/2}$ and let $a_0, a_1, \ldots, a_{i_c}$ and $r_0, r_1, \ldots, r_{i_c}$ be the sequences computed in Algorithm 2. Then Algorithm 2 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 consequence of Lemma 1: statements (i) and (ii) imply that for $i \geq 1$

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

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

A direct consequence of Lemma 2 is the following. If $N^{1/2}$ is not an integer and if Algorithm 2 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 multiplicative splitting.

Complexity. For the sake of simplicity, we will only give an upper bound on the cost of the multiplicative splitting. Specifically, since computing a multiplicative splitting consists in a partial execution of the extended Euclidean algorithm, we can bound its cost above with an upper bound of the complexity of the extended Euclidean algorithm. We use the following lemma inspired from [16].

Lemma 3 (Complexity of the extended Euclidean algorithm) *The extended Euclidean algorithm (i.e. Algorithm 2 with $c = 1$), with two positive integers $x \leq N$ of w-bit word length t as input, requires at most $4wt^2$ word additions.*

Proof We will consider a modified version of Algorithm 2: 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. The cost of this modified algorithm is equal to

(number of iterations) × (cost of one loop body)

We have:

- *Cost of one loop body.* If we assume that the integers a_i and r_i in Algorithm 2 are stored on t words, each loop body requires $2t$ word subtractions.
- *Number of iterations.* At each iteration we remove the most significant bit of r_i or r_{i-1} by at least one bit. This reduces the bit length $\lceil \log_2(r_i) \rceil + \lceil \log_2(r_{i-1}) \rceil$ by one. This implies that the number of iterations before we get $r_i = 0$ is at most $\lceil \log_2(x) \rceil + \lceil \log_2(N) \rceil \leq 2tw$.

At the end the total number of operations is at most $2tw \times 2t = 4t^2w$ word subtractions. □

3 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 3. This algorithm reaches our goal since it is regular: each iteration of the loop body is a squaring followed by a half-size multiplication. It is also robust against safe error fault injection attack: each error in one half-size multiplication will affect the final result.

Algorithm 3 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: $\quad temp \leftarrow k_i x_1 + (1 - k_i)x_0$
5: $\quad r \leftarrow r^2 \times temp \mod N$
6: **end for**
7: $r \leftarrow r \times x_0 \mod N$
8: **return** r

The following lemma establishes the validity of Algorithm 3, i.e., that it correctly computes $r = x^K \mod N$.

Lemma 4 *Let $K = (k_{\ell-1}, \ldots, k_0)_2$ with $k_i \in \{0, 1\}$ be an ℓ-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 the iteration 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 $i-1$. We denote r_i the value of r after the execution of iteration i in Algorithm 3 and we assume that it satisfies $r_i = x^{K_i} \times x_0^{-1}$. Then if $k_{i-1} = 1$ the execution of iteration $i-1$ gives:

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

since $K_{i-1} = 2K_i + 1$. And, if $k_{i-1} = 0$, the execution of iteration $i-1$ gives:

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

since in this case $K_{i-1} = 2K_i$. □

4 Exponentiation with half-size splitting and Montgomery multiplication

An RSA modulus N looks like a random integer: the binary representation is not sparse and has no other underlying structure which can be used to speed-up a reduction modulo N. The most commonly used method to perform a multiplication modulo a random integer is the Montgomery method [8]. We modify Algorithm 3 in order to use the Montgomery multiplication for the squarings and multiplications modulo N. A squaring in Algorithm 3 involves integers of size $\lceil \log_2(N) \rceil$ bits, while a multiplication involves two kinds of multiplicand: one integer of size $\lceil \log_2(N) \rceil$ bits and one integer of size $\cong \lceil \log_2(N)/2 \rceil$ bits. This compels 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 integers of size $\lceil \log_2(N) \rceil$ bits. 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 4.

Algorithm 4 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: $\quad r \leftarrow FMS(r, r)$
5: $\quad temp \leftarrow k_i x_1 + (1 - k_i) x_0$
6: $\quad r \leftarrow HMM(r, temp)$
7: **end for**
8: $r \leftarrow \left(r \times x_0 \times m^{-1} \times M^{-1}\right) \mod N$
9: **return** r

Lemma 5 *Let $K = (k_{\ell-1}, \ldots, k_0)_2$ with $k_i \in \{0, 1\}$ be an ℓ-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 iteration i in Algorithm 4 satisfies:*

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

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

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

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

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

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

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

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

5 Complexity comparison and implementation results

In this section we first briefly 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 Sect. 2.

5.1 Word-level Montgomery multiplication and squaring

The proposed exponentiation in Algorithm 4 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 5.

The complexity of Algorithm 5 is evaluated step by step in Table 2. 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:

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

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 6.

Algorithm 5 Word-level Montgomery multiplication [17]

Require: $N < 2^{wt-1}$ 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 **from** 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 2 Step by step complexity evaluation of word-level Montgomery multiplication (Algorithm 5)

	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		$s(4t+2)+t$	$s(2t+1)$

Algorithm 6 Word-level Montgomery squaring [9]

Require: $N < 2^{wt-1}$ 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 **from** 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

The complexity of Algorithm 6 is evaluated step by step in Table 3. 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 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 3.

5.2 Complexity comparison

Now, we can deduce the cost of a FMM, a FMS and a HMM from the complexity of the word-level Montgomery multiplication and squaring. Specifically, the cost of a FMS with $M = 2^{tw}$ is the same as the one shown in Table 3. To obtain the complexity of FMM with $M = 2^{tw}$, we take $s = t$ in the formula of Table 2 and to get the complexity of a HMM with $m = 2^{tw/2}$ we take $s = t/2$ in the formula of Table 2. This leads to the complexities shown in the upper part of Table 4.

Table 3 Step by Step complexity evaluation of a word-level Montgomery squaring (Algorithm 5)

	Operations	# Word add.	# Word mul.
Step 1	$x + x$	t	0
t Step 4	$\lvert 2x_{i+1}\rvert_{2^w}$	t	0
t Step 5	$z + 2^{wi}\widetilde{x}_i x_i$	$t^2 + 2t$	$\frac{t(t+1)}{2}$
t Step 6	$\lvert z\rvert_{2^w} \cdot N'$	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		$3t^2 + 6t$	$\frac{3t^2}{2} + \frac{3t}{2}$

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

- The square-and-multiplication exponentiation requires ℓ FMS and $\ell/2$ FMM in average.
- The multiply-always exponentiation necessitates $3\ell/2$ FMM in average.
- The square-always exponentiation necessitates 2ℓ FMS in average.
- The square-and-multiply-always and Montgomery-ladder exponentiation require ℓ FMS and ℓ FMM.
- The Montgomery-ladder exponentiation with common multiplicand [9]: this necessitates ℓ word-level combined Montgomery multiplications AB, AC, which have a reduced complexity by sharing some computations involved in reductions modulo N (cf. [9] for details).

The complexities of these approaches in terms of the number of word additions and multiplications are given in Table 4.

For the proposed regular exponentiation with half-size Montgomery multiplication (Algorithm 4), we need ℓ FMS and ℓ HMM in the ℓ iterations of the loop body. For the computation of the multiplicative splitting x_0 with Algorithm 2, the cost is, using Lemma 3, 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 4.

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 all regular approaches: the square-always and multiply-always exponentiations and also the Montgomery-ladder and square-and-multiply always approaches.

5.3 Implementation results

We implemented in C language the different approaches and compiled them on an Intel Core i7 Broadwell 5600U with gcc-4.8.4 and on a quad-core ARMv7 Cortex-A7 with gcc-4.9.2. For modular multiplication and modular squaring, we implemented Algorithm 6 and Algorithm 5 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 was implemented using the low-level function of GMP for Euclidean division. The timings obtained for a number of

Table 4 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 4)	$\ell(5t^2+8t)+10t^2+8t$	$\ell(\frac{5t^2}{2}+2t)+8wt^2+5t^2+\frac{5t}{2}$

Table 5 Timings in 10^3 clock-cycles of modular exponentiation

	Algorithm	Timings on a Core i7			Timings on an ARMv7		
		2048 bits	3072 bits	4096 bits	2048 bits	3072 bits	4096 bits
Exponentiation without side channel protection	Square-and-multiply	12,811	40,207	93,094	155,005	502,142	1,175,073
Non constant time regular exponentiations	Multiply-always [5]	13,896	45,407	106,177	175,946	575,859	1,373,075
	Square-always [5]	16,751	52,120	118,744	193,493	620,804	1,450,404
Regular and constant time exponentiations	Montgomery-ladder [7]	17,669	56,449	130,436	214,077	702,270	1,633,957
	Montgomery-ladder with CM [9]	15,478	48,963	113,133	183,707	598,020	1,398,734
	Square-and-multiply-always [6]	17,619	56,137	130,043	214,249	697,046	1,634,550
	Proposed (Algorithm 4)	13,616	42,139	96,547	158,805	509,769	1,188,119

practical bit lengths of N (i.e., 2048, 3072 and 4096) are reported in Table 5. We used Papi library [18] to get cycle counts on both platforms. These timings are the average of 1000 timings obtained with random input messages x and random exponents K.

We notice that the reported timings relate to the complexity results shown in Table 4. Indeed, the fastest approach is the square-and-multiply exponentiation which is not protected against simple side channel analysis. Our approach is less than 6.3 slower than square-and-multiply for any key size. Our approach is better than all other approaches: by 1–13.4 % compared to the multiply-always approach, which is not entirely secure against SPA, and more than 12 % compared to the other regular approaches.

6 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 method. Compared to the usual regular and constant time modular exponentiations, the proposed method involves only multiplications by half-size integers instead of full multiplications. This leads to a reduction of the complexity by 16 % and an improvement of the timing by 12 % compared to other approach which are both regular and constant time.

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

Appendix

Proof of Lemma 1 • *Proof of* (i). We prove by induction on i that $(-1)^{i-1}a_i \geq 1$ for all $i \geq 1$. For $i = 1$ we have $a_1 = 1$ which implies $(-1)^{i-1}a_i = 1$ as required. For $i = 2$ we have $a_2 = -q_1 a_1$ which implies $(-1)^1 a_2 = q_1 a_1 \geq 1$. Now, we suppose that the inequality holds for $i - 1$ and i , i.e.,

$$(-1)^{i-2}a_{i-1} \geq 1 \quad \text{and} \quad (-1)^{i-1}a_i \geq 1, \tag{9}$$

and we prove that the inequality is also true for $i + 1$. We starts with $(-1)^i a_{i+1}$ and replace a_{i+1} by its expression in terms of a_i , a_{i-1}, r_i and r_{i-1} in Algorithm 2. We obtain the following:

$$\begin{aligned}(-1)^i a_{i+1} &= (-1)^i \left(a_{i-1} - \lfloor r_{i-1}/r_i \rfloor a_i\right)\\ &= (-1)^i a_{i-1} - \lfloor r_{i-1}/r_i \rfloor (-1)^i a_i\\ &= (-1)^{i-2} a_{i-1} + \lfloor r_{i-1}/r_i \rfloor (-1)^{i-1} a_i\\ &\geq 1 + \lfloor r_{i-1}/r_i \rfloor \qquad \text{(Using (9))}\end{aligned}$$

Therefore, we have proven by induction that $(-1)^i a_i \geq 1$ for all i.

- *Proof of* (ii). We follow the proof of [16]: we express the inductive expression of a_i and r_i as a 2×2 matrix product:

$$\begin{pmatrix} a_{i+1} & r_{i+1} \\ a_i & r_i \end{pmatrix} = \begin{pmatrix} -\lfloor r_{i-1}/r_i \rfloor & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} a_i & r_i \\ a_{i-1} & r_{i-1} \end{pmatrix}.$$

Now since for all i we have $\det \begin{pmatrix} -\lfloor r_{i-1}/r_i \rfloor & 1 \\ 1 & 0 \end{pmatrix} = -1$, we obtain by induction that

$$\begin{aligned}\det \begin{pmatrix} a_{i+1} & r_{i+1} \\ a_i & r_i \end{pmatrix} &= (-1)^i \det \begin{pmatrix} a_1 & r_1 \\ a_0 & r_0 \end{pmatrix}\\ &= (-1)^i \det \begin{pmatrix} 1 & x \\ 0 & N \end{pmatrix}\\ &= (-1)^i N.\end{aligned}$$

Finally we obtain that

$$\forall i \geq 0, a_{i+1} r_i - a_i r_{i+1} = (-1)^i N.$$

References

1. Rivest, R., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM **21**, 120–126 (1978)
2. Kocher, P.C., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener, M.J. (ed.): Advances in Cryptology–CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15–19, 1999, Proceedings, Lecture Notes in Computer Science, vol. 1666, pp. 388–397. Springer, Berlin (1999)
3. Mangard, S.: Exploiting Radiated Emissions - EM Attacks on Cryptographic ICs. In: *Austrochip 2003, Linz, Austria, October 1st*, pp. 13–16 (2003)
4. Amiel, F., Feix, B., Tunstall, M., Whelan, C., Marnane, W.: Distinguishing Multiplications from Squaring Operations. In: SAC 2008, ser. LNCS, vol. 5381, pp. 346–360. Springer (2009)
5. Clavier, C., Feix, B., Gagnerot, G., Roussellet, M., Verneuil, V.: Square Always Exponentiation. In: Progress in Cryptology - INDOCRYPT, 2011 ser. LNCS, vol. 7107, pp. 40–57. Springer (2011)
6. Coron, J.-S.: Resistance against differential power analysis for elliptic curve cryptosystems. In: Koç, Ç.K., Paar, C. (eds.): Cryptographic Hardware and Embedded Systems. First International-Workshop, CHES'99 Worcester, MA, USA, August 12–13, 1999, Proceedings, Lecture Notes in Computer Science, vol. 1717, pp. 292–302. Springer, Berlin (1999)
7. Joye, M., Yen, S.: The Montgomery Powering Ladder. In: CHES, 20002 ser. LNCS, vol. 2523, pp. 291–302. Springer (2002)
8. Montgomery, P.: Modular multiplication without trial division. Math. Comput. **44**, 519–521 (1985)
9. Negre, C., Plantard, T., Robert, J.: Efficient Modular Exponentiation Based on Multiple Multiplications by a Common Operand. In: 22nd IEEE Symposium on Computer Arithmetic 2015, pp. 144–151 (2015)
10. Menezes, A., van Oorschot, P., Vanstone, S.: Handbook of Applied Cryptography. CRC Press, Boca Raton (1996)
11. Yen, S.-M., Joye, M.: Checking before output may not be enough against fault-based cryptanalysis. IEEE Trans. Comput. **49**(9), 967–970 (2000)
12. Yen, S.-M., Kim, S., Lim, S., Moon, S.-J.: A Countermeasure against One Physical Cryptanalysis May Benefit Another Attack. In: ICISC, 2001 ser. LNCS, vol. 2288, pp. 414–427. Springer (2001)
13. Joye, M., Tunstall, M.: Exponent Recoding and Regular Exponentiation Algorithms. In: Progress in Cryptology - AFRICACRYPT, 2009 ser. LNCS, vol. 5580, pp. 334–349. Springer (2009)
14. Bryant, E., Rambhia, A., Atallah, M. and Rice, J.: Software Trusted Platform Module and Application Security Wrapper," Jan 2011, US Patent 7,870,399. [Online]. https://www.google.ch/patents/US7870399
15. Gallant, R., Lambert, R., Vanstone, S.: Faster Point Multiplication on Elliptic Curves with Efficient Endomorphisms. In: Advances in Cryptology-CRYPTO, 2001 ser. LNCS, vol. 2139, pp. 190–200 Springer (2001)
16. von zur Gathen, J.: Modern Computer Algebra, 3rd edn. Cambridge University Press, Cambridge (2013)
17. Bosselaers, A., Govaerts, R. and Vandewalle, J.: "Comparison of Three Modular Reduction Functions," in *Advances in Cryptology-CRYPTO'93*, ser. LNCS, vol. 773. Springer, pp. 175–186 (1993)
18. Papi, M.: "Performance Application Programming Interface (PAPI)." [Online]. Available: http://icl.cs.utk.edu/papi/