Efficient Fixed-base exponentiation and scalar multiplication based on a multiplicative splitting exponent recoding

Jean-Marc Robert  
*University of Perpignan, University of Montpellier*

Christophe Negre  
*Universite de Perpignan, University of Montpellier*

Thomas Plantard  
*University of Wollongong, thomaspl@uow.edu.au*
Efficient Fixed-base exponentiation and scalar multiplication based on a multiplicative splitting exponent recoding

Abstract
Digital signature algorithm (DSA) (resp. ECDSA) involves modular exponentiation (resp. scalar multiplication) of a public and known base by a random one-time exponent. In order to speed up this operation, well-known methods take advantage of the memorization of base powers (resp. base multiples). Best approaches are the Fixed-base radix-R method and the Fixed-base Comb method. In this paper, we present a new approach for storage/online computation trade-off, by using a multiplicative splitting of the digits of the exponent radix-R representation. We adapt classical algorithms for modular exponentiation and scalar multiplication in order to take advantage of the proposed exponent recoding. An analysis of the complexity for practical size shows that our proposed approach involves a lower storage for a given level of online computation. This is confirmed by implementation results showing significant memory saving, up to 3 times for the largest NIST standardized key sizes, compared to the state-of-the-art approaches.

Disciplines
Engineering | Science and Technology Studies

Publication Details

This journal article is available at Research Online: https://ro.uow.edu.au/eispapers1/2781
Efficient Fixed Base Exponentiation and Scalar Multiplication based on a Multiplicative Splitting Exponent Recoding

Jean-Marc Robert\textsuperscript{2,3}, Christophe Negre\textsuperscript{2,3} and Thomas Plantard\textsuperscript{1}

\textbf{Abstract} Digital Signature Algorithm (DSA) (resp. ECDSA) involves modular exponentiation (resp. scalar multiplication) of a public and known base by a random one-time exponent. In order to speed-up this operation, well-known methods take advantage of the memorization of base powers (resp. base multiples). Best approaches are the Fixed-base Radix-$R$ method and the Fixed-base Comb method. In this paper we present a new approach for storage/online computation trade-off, by using a multiplicative splitting of the digits of the exponent radix-$R$ representation.

We adapt classical algorithms for modular exponentiation and scalar multiplication in order to take advantage of the proposed exponent recoding. An analysis of the complexity for practical size shows that our proposed approach involves a lower storage for a given level of online computation. This is confirmed by implementation results showing significant memory saving, up to 3 times for the largest NIST standardized key sizes, compared to the state of the art approaches.

\textbf{Keywords} RNS, Multiplicative Splitting, Digital Signature, Fixed Base, Modular Exponentiation, Scalar Multiplication, Memory Storage, Efficient Software Implementation.

\section{Introduction}

In the DSS (Digital Signature Standard), DSA (Digital Signature Algorithm) is a popular authentication protocol. According to the NIST standard (see \cite{12}), the public parameters are $p, q$ and $g$. The parameter $g$ is a generator of a multiplicative sub-group of $\mathbb{F}_p^*$ of size $q$. The integers $p$ and $q$ are two primes with sizes corresponding to the required security level; for the recommended security level 80-256 bits, $q$ has to be a 160-512 bit integer. When a server needs to sign a batch of documents, the most costly operations are modular exponentiations $g^k \mod p$ (one per signature), where $g, p$ are fixed and $k$ is a one time random integer.

Another popular standard for electronic signature is ECDSA which uses the group of point on an elliptic curve $(\mathbb{E}(\mathbb{F}_p), +)$ instead of $(\mathbb{F}_p^*, \times)$. The signature algorithm ECDSA is very similar to the DSA and its main operation is a scalar multiplication $k \cdot P$ for $P \in \mathbb{E}(\mathbb{F}_p)$. In order to cover both cases DSA and ECDSA we consider a multiplicative abelian group $(G, \times)$ in which we have to compute $g^k$ for $g \in G$ and $k \in \mathbb{N}$.

In this article we consider the following practical case: a server has to compute a large number of signatures, which involves a large number of exponentiations $g^k$ with the same $g \in G$ and several random $k$. We assume that the server has a large cache and RAM (Random Access Memory) so that we can therefore store a large amount of precomputed data to speed-up these exponentiations. In the sequel, by 'offline computation' we mean the data computed only once and used in every signature generation; by 'online computation' we mean the operations required only in a single exponentiation $g^k$ for a given $k$.

The main known methods of the state of the art which take advantage of large amount of precomputed data are the Fixed-base Radix $R$ presented by Gordon in \cite{8} and the Fixed-base Comb presented by Lim and Lee in \cite{14}. The Fixed-base Radix $R$ method of \cite{8} precomputes $g^{aR}$ for $0 \leq a < R$ and then, using the radix-$R$ expression of $k$, we obtain the exponentiation $g^k$ with $\log_R(k)$ multiplications. The Fixed-base Comb method uses a Comb decomposition of $k$ (instead of a radix-$R$
representation) and requires less precomputed data at the cost of some extra squarings. In [17] the authors provide a variant of the Radix-$R$ approach using the NAF$_w$ recoding resulting in a reduced number of online multiplications than for the radix-$R$ approach but with a penalty of some extra squarings.

**Contributions.** We investigate some new strategies for a better trade-off between storage and online computation in fixed base exponentiation. To reach this goal, we propose to use the representation of the exponent in radix $R$ as $k = \sum_{i=0}^{t-1} k_i R^i$ and then compute a multiplicative splitting of each digit $k_i$. Specifically, we use a radix $R = m_0 m_1$ with pairwise prime $m_0, m_1$. An RNS representation of a digit $k_i \in [0, R]$ in $\{m_0, m_1\}$ leads to a splitting into two parts: one part $k_i^{(0)}$ which value is at most $m_0$ and the other $k_i^{(1)}$ which value is at most $m_1$. We apply this process to all the digits of the radix $R$ representation of the exponent. While processing the exponentiation, the digits $k_i^{(1)}$ are handled with a look-up table and the digits $k_i^{(0)}$ are handled with online computation. This approach was part of a preliminary version of this paper published in the proceedings of the WAIFI 2016 conference [20].

We present a novel approach for the multiplicative splitting of the digits of the exponent: if we choose the radix $R$ as a prime integer, then processing a partial execution of the extended euclidean algorithm, one can re-express a digit $k_i$ as product $k_i = k_i^{(0)}(k_i^{(1)})^{-1} \mod R$ where $|k_i^{(1)}| < c$ and $|k_i^{(0)}| < R/c$ for a fixed $c$. Again, this splitting can be applied to all digits of the radix $R$ representation of the exponent. The exponentiation algorithms can then be computed with memorizations related to the $(k_i^{(1)})^{-1}$ part of the digit splitting and online computation to handle the part $k_i^{(0)}$ of the digit splitting. The main advantage of this version with a prime $R$ is that the resulting exponentiation algorithm is constant time, which means that it is robust against timing attacks.

We study the corresponding complexities and storage amounts, and compare the results with the best approaches of the literature for fixed-base modular exponentiation (resp. scalar multiplication) for NIST recommended fields (resp. curves). The metric chosen for a comparison between the proposed algorithms is the following: for a given level of online computation the best approach is the one which has the lowest amount of precomputed data. Using this metric we show that the proposed approach is the more efficient for a large range of practical case. We also implement these approaches in software and we perform tests in order to validate the complexity analysis. Our approaches provide also some flexibility in terms of required storage amount: one can choose the storage amount according to the device resources available and compatible to the global computation load of the system.

**Organization of the paper.** In Section 2 we review the best approaches of the literature for fixed-base exponentiation and we give their complexities and storage requirements. In Section 3 we present a multiplicative splitting recoding of the exponent in radix $R = m_0 m_1$ and a fixed-base exponentiation using this recoding. In Section 4 we present a multiplicative splitting recoding for $R$ prime and the corresponding exponentiation algorithm. In Section 5 we compare the complexity results and software implementations of the proposed approach to the best approaches of the literature for modular exponentiation and scalar multiplication. Finally, in Section 6 we give some concluding remarks and perspectives.

## 2 State of the Art of Fixed-Base Exponentiation

We consider digital signature algorithms based on discrete logarithm in a finite group. The main ones are DSA where the considered group is a subgroup of prime order $q$ in the multiplicative group $\mathbb{F}_p^*$ and ECDSA where the group is the set of point on an elliptic curve $E(\mathbb{F}_p)$. For the sake of simplicity, in the sequel, we use a generic abelian multiplicative group $(G, \times)$ of order $q$. The algorithms presented later in this paper extend directly to abelian groups with additive group law like $E(\mathbb{F}_p)$. Generating a digital signature consists in computing $(s_1, s_2)$ from a message $m \in \{0, 1\}^*$, a secret integer $x$ and a random integer $k$ as follows

\[
\begin{align*}
s_1 &\leftarrow H_1(g^k), \\
s_2 &\leftarrow (H_2(m) + s_1 x)k^{-1} \mod q.
\end{align*}
\]

Here, $H_1$ is a function $G \to \mathbb{Z}/q\mathbb{Z}$ and $H_2$ is a cryptographic hash function $\{0, 1\}^* \to \mathbb{Z}/q\mathbb{Z}$. One can see that the most costly operation in a signature generation is the exponentiation $g^k$ of a fixed $g \in G$ and where $k$ is a one-time random exponent of size $\equiv q$. This exponentiation can be done with the classical Square-and-multiply algorithm.

**Square-and-multiply exponentiation.** The left-to-right version of the square-and-multiply exponentiation scans the bits $k_i$ of $k$ from left to right and performs a squaring followed by a multiplication when $k_i = 1$. In terms of complexity, given the bit length $t$ of $k$, the number of squarings is $t$ and the number of multiplications to be computed is $t/2$ on average for a randomly chosen exponent. There is no storage in this case.
Ensure:

X = g^k

1: X ← 1
2: for i from t − 1 downto 0 do
3: X ← X^2
4: if k_i = 1 then
5: X ← X · g
6: return (X)

**Algorithm 1** Left-to-Right Square-and-multiply Exponentiation

**Require:** Let an integer k = (k_{t−1}, ..., k_0)_2, and g an element of G.

**Ensure:** X = g^k

*Side channel analysis.* The above method is threatened by side-channel analysis. These attacks extract part of the exponent by monitoring and analyzing the computation time, the power consumption or the electromagnetic emanations. In this paper, we focus on servers which generate large amounts of signature very quickly and are physically not accessible to an attacker. The main threat in this case is the timing attack. This attack attempts to find the sequence of operations (multiplication and squaring) of an exponentiation by a statistical analysis of several timings of an exponentiation. If the assumed sequence of operations is correct, the attacker can deduce the key bits of the exponent since each multiplication corresponds to a bit equal to 1, otherwise the bit is 0. A general solution to thwart this attack is to render the sequence of operations not correlated to key bits, which means that we need to remove any if test on the key bits or digits in the exponentiation algorithm.

**Fixed base exponentiation.** When the base g is fixed, one can precompute in advance some data in order to reduce the number of operations in the online computation of the exponentiation. This is the case when a server has to intensively compute a number of signatures with the same g. For example, the method presented by Gordon in [8] is a modified square-and-multiply algorithm: one first stores the t successive squarings of g (that is the sequence of g^{2^j}), then for a given computation of g^k, one has to multiply the g^{2^j} corresponding to k_i = 1. In terms of complexity, given the bit length t of the exponent, one has now no squarings and the number of multiplications is t/2, in average. As counterpart, one has to store t elements of G. We can even further reduce the amount of online computation by increasing the precomputed data. This is the strategy followed by the main approaches of the literature.

**Radix-R method.** Gordon in [8] mentions the generalization of his first idea to radix R = 2^w representation of the exponent k = \sum_{i=0}^{t−1} k_i R^i. This consists in the memorization of the values g^{a_R^j}, with a \in [0, ..., R − 1] and 0 \leq j < t where t is the length of the exponent in radix R representation. If we denote w = \lfloor \log_2(R) \rfloor then we have \ell = \lfloor t/w \rfloor. In this case, the online computation consists of \ell − 1 multiplications, for a storage amount of t · R values in G. In the sequel, we will call this approach the *Fixed-base Radix-R* exponentiation method (see Algorithm [2]). This algorithm is constant time as soon as the multiplications by 1 (i.e., when k_i = 0) are performed as any other multiplication or, alternatively, by using the radix R recoding of [11] which avoids k_i = 0.

**Algorithm 2** Fixed-Base Radix-R Exponentiation

**Require:** k = (k_{t−1}, ..., k_0)_R, g a generator of G.

**Ensure:** X = g^k

1: Offline precomputation. Store T[a][j] ← g^{a_R^j}, with a \in [0, ..., R − 1] and 0 \leq j < t.
2: X ← 1
3: for i from t − 1 downto 0 do
4: X ← X · (T[k_i][i]
5: return (X)

*Comb method.* Another classical method is the so called *Fixed-base Comb* method which was initially proposed by Lim and Lee in [13]. This method attempts to trade some of the storage of Algorithm [2] with a few online computed squarings. It is based on the following decomposition of the exponent k

\[ k = \sum_{j=0}^{d−1} \left( \sum_{i=0}^{w−1} k_{id+j}2^i \right) 2^j \]  \tag{1}

Each integer K_j can be seen as a comb as described in the following diagram.

The integer w is the number of comb-teeth in each K_j and d = \lceil t/w \rceil is the distance in bits between two consecutive teeth. When all the possible values g^{K_j} are precomputed and stored in table indexed by

\[ I_{K_j} = [k_{(w−1)d+j} k_{(w−2)d+j} ⋯ k_{d+j}]_2, \]

one can compute g^k with a 2^w size look-up table, [t/w] − 1 multiplications and [t/w] − 1 squarings using [1]. This method is shown in Algorithm [3] As in the case of Radix-R method, this approach can be implemented in constant time if the multiplications by 1 (which occurs K_j = 0) are computed as an arbitrary multiplication or by using the recoding of [10] which renders all comb coefficients \neq 0.
**Algorithm 3** Fixed-Base Comb Exponentiation \[14\]

**Require:** \( k = (k_{-1}, \ldots, k_1, k_0) \), a generator \( g \) of \( G \), a window width \( 2^w \) and \( d = \lfloor t/w \rfloor \).

**Ensure:** \( X = g^{\ell} \mod p \)

1: **Offline precomputation.** For all \((a_{w-1}, \ldots, a_0) \in \{0,1\}^w\) we set \( a = a_{w-1}2^{w-1}d + \cdots + a_12^d + a_0 \) and \( T[(a_{w-1}, \ldots, a_0)] = g^a \).

2: Split \( k = \sum_{j=0}^{d-1} K_j 2^j \) as in \([1]\).

3: \( X \leftarrow 1 \)

4: for \( j \) from \( d-1 \) downto \( 0 \) do

5: \( X \leftarrow X^2 \)

6: \( X \leftarrow X \cdot T[K_j] \)

7: return \((X)\)

---

**Fixed base exponentiation with NAF.** In \([17]\), the authors proposed an alternative approach when inverting an element in the group \( G \) is almost free of computation and multi-squarings can be computed efficiently. Their main application is the group of points on an elliptic curve. To compute the inverse of a point is really cheap. They use a NAF representation of \( k \) in order to reduce the number of multiplications (this generalizes the approach of [21] which uses a NAF representation of \( k \)). Specifically, they start by computing the NAF representation of the exponent \( k \)

\[
k = k'_12^{t-1} + k'_22^{t-2} + \cdots + k'_0
\]

where \( k'_j \in \{\pm 1, \pm 3, \ldots, \pm 2^{w-1} - 1\} \) and there are at least \( w \) zeros between two non-zero coefficients. In [21] one can find more details on NAF and the reader may refer to [9]. Then they rewrite this NAF representation \( k \) into \( \ell = \lceil t/w \rceil \) consecutive windows of coefficients:

\[
k = \sum_{i=0}^{\ell} \left( \sum_{j=0}^{w-1} k'_{i+w+j} 2^j \right) 2^{iw}.
\]

In \([17]\) the authors noticed that, in each \( K_i \), there is at most one non-zero coefficient \( k'_{i+w+j} \), which means that \( K_i = s \times a \times 2^j \) for some \( s \in \{-1, 1\}, a \in \{1, 3, \ldots, 2^{w-1}-1\} \) and \( 0 \leq j < w \). They then reorder the terms in expression \([2]\) by splitting the parameter \( i \) into two parts \( i = i_1 + i_0 \) for some fixed integer \( e \):

\[
k = \sum_{i_0=0}^{e-1} \sum_{i_1=0}^{d-1} K_{i_1} 2^{i_1} 2^{i_0} w \text{ where } d = \lceil t/e \rceil
\]

\[
= \sum_{i_0=0}^{\ell} \left( \sum_{i_1=0}^{d-1} K_{i_1} 2^{i_1+e} 2^{i_0} w \right)
\]

For all possible values for \( K_{i_1} 2^{i_1} 2^{i_0} w \) with \( K_{i_1} = s \times a \times 2^j \) the term \( g^{2^{i_1} 2^{i_0} w} \) is stored in a Table \( T[a][i_1][j] \). Then Algorithm \([4]\) computes \( g^k \) based on \([3]\) as a sequence of multiplications/divisions (in Step 9 depending on \( s = 1 \) or \( s = -1 \)) and \( w \) consecutive squarings (in Step 5).

**Algorithm 4** Fixed-Base Exponentiation with NAF \([17]\)

**Require:** A scalar \( k = (k'_{-1}, \ldots, k'_1, k'_0)_{NAF_w} \) and \( g \) in an abelian group \( G \), and positive integers \( c, w \).

**Ensure:** \( X = g^k \)

1: \( \ell = \lceil \frac{t}{w} \rceil \) and \( d = \lceil \frac{\ell}{e} \rceil \)

2: **Offline precomputation.** \( T[a][i_1][j] = g^{2^{i_1} 2^{i_0} w} \) for all \( a \in \{1, 3, \ldots, 2^{w-1}-1\}, i_1 \in \{0, \ldots, d-1\} \) and \( j \in \{0, \ldots, w-1\} \).

3: \( X \leftarrow 1 \)

4: for \( i_0 \) from \( c-1 \) downto \( 0 \) do

5: \( X \leftarrow X^2 \)

6: for \( i_1 \) from \( d-1 \) downto \( 0 \) do

7: \((s, a, j)\) s.t. \((k'_{i_1+i_0}, k'_0)_{NAF_w} = s \times a \times 2^j\)

8: if \( a \neq 0 \) then

9: \( X \leftarrow X \times (T[a][i_1][j])^s \)

10: return \((X)\)

In Algorithm \([4]\) the number of precomputed elements is equal to \( dw2^{w-2} \cong \lceil \frac{d}{w} \rceil 2^{w-2} \). The online computation consists of \( w(\ell - 1) \) squarings and \( ed(1 - (\frac{w}{\ell + 1})^w) \cong \lceil \frac{d}{w} \rceil (1 - (\frac{w}{w+1})^w) \) multiplications/divisions (cf. \([17]\) for details).

---

**3 Fixed-base exponentiation with multiplicative splitting with \( R = m_0m_1 \)**

We now present our approach of a Fixed-base exponentiation with multiplicative splitting with \( R = m_0m_1 \). In this section, we review the method presented in a preliminary work at WAFI 2016 \([20]\). The goal is to use a multiplicative splitting of the digits of \( k \) in order to provide a better trade-off between storage and online computation in the exponentiation.

3.1 Digit multiplicative splitting for radix \( R = m_0m_1 \)

A natural way to get a splitting of the digits is to use the RNS representation in radix \( R = m_0 \cdot m_1 \) which splits any digit into two parts. When all the digits of an exponent are split we can process the exponentiation as follows: the first part of the digits will be used to select the precomputed values and the second part will be processed by online computation.

We first remind the RNS representation in a base \( B = \{m_0, m_1\} \). Let \( R = m_0 \cdot m_1 \) and \( x \in \mathbb{Z} \) such that \( 0 \leq x < R \). Let us also assume \( m_0 \) is prime, since this allows us to invert all non-zero integers \( < m_0 \) modulo \( m_0 \), and we choose \( m_1 < m_0 \). In the sequel, we denote \( |x|_{m_0} = x \mod m_0 \).

One represents \( x \) with the residues:

\[
\begin{align*}
x^{(0)} &= |x|_{m_0}, \\
x^{(1)} &= |x|_{m_1},
\end{align*}
\]
We recode the digits of base $R$ as follows:

$$x = \left\lfloor x^{(0)} \cdot m_1 \cdot m_0^{-1} \right\rfloor_m + x^{(1)} \cdot m_0 = \left\lfloor x^{(0)} \cdot m_0^{-1} \right\rfloor_m R. \tag{4}$$

We now present our recoding approach. We consider an exponent $k$ expressed in radix $R = m_0 \cdot m_1$

$$k = \sum_{i=0}^{\ell-1} k_i R^i \quad \text{with} \quad \ell = \lceil t / \log_2(R) \rceil.$$

We represent every radix-R digit in RNS with the RNS base $B = \{m_0, m_1\}$: if $k_i$ is the $i$-th digit of $k$ in radix-$R$, we denote by $(k_i^{(0)}, k_i^{(1)})$ its RNS representation in base $B$.

$$\begin{cases} k_i^{(0)} = \lfloor k_i/m_0 \rfloor, \\ k_i^{(1)} = \lfloor k_i/m_1 \rfloor. \end{cases}$$

Let us denote

$$m_0' = m_1 \cdot m_1^{-1} \lfloor m_0 \rfloor, \quad m_1' = m_0 \cdot m_0^{-1} \lfloor m_1 \rfloor.$$

We recode the digits of $k$ in $B = \{m_0, m_1\}$ as follows

- If $k_i^{(1)} \neq 0$: we denote

$$\begin{cases} k_i^{(0)} = \lfloor k_i^{(0)} \cdot (k_i^{(1)})^{-1} \rfloor_m, \\ k_i^{(1)} = k_i^{(1)} \cdot (0)^{m_1}. \end{cases}$$

One keeps

$$k_i' = k_i^{(1)} | k_i^{(0)} \cdot m_0' + m_1' |_R \tag{5}$$

as a representation of $k_i$ in a multiplicative splitting form and we have $k_i = \lfloor k_i' \rfloor_R$ with $\lfloor \rceil$. When modifying the digits of $k$ as above, one needs to take into account the correcting term due to the reduction modulo $R$:

$$k_i = k_i^{(1)} | k_i^{(0)} \cdot m_0' + m_1' |_R - [k_i^{(0)} \cdot k_i^{(1)} \cdot m_0' + m_1' |_R \cdot R].$$

Let us denote $C = |k_i^{(1)} \cdot (k_i^{(0)} \cdot m_0' + m_1' |_R]$, which satisfies $0 \leq C < m_1$. We consider $C$ as a carry that one can subtract to $k_{i+1}$. This leads to the following computation:

- If $k_{i+1} \geq C$ then
  $$k_{i+1} \leftarrow k_{i+1} - C, \quad C \leftarrow 0$$
- else
  $$k_{i+1} \leftarrow k_{i+1} + R - C, \quad C \leftarrow 1$$

and one gets $k_{i+1} \geq 0$.

- If $k_i^{(1)} = 0$: we define $k_i'$ as follows

$$k_i' = \left\lfloor k_i^{(0)} + 1 \lfloor m_0' \cdot m_0' + m_1' |_R \cdot m_0' + m_1' |_R \right\rfloor \tag{6}$$

and $k_i'$ satisfies $|k_i'|_R = k_i$. This expression is meant to have the part $\left(\ast\right)$ as in the theorem: the goal is to use the same precomputed data in the exponentiation algorithm. The term $- |m_0' + m_1' |_R - 1$ is meant to get back to $k_i$ while reducing $k_i'$ modulo $R$. We then set the following coefficients:

$$\begin{cases} k_i^{(0)} = |k_i^{(0)} + 1 \lfloor m_0 \rfloor, \\ k_i^{(1)} = 0. \end{cases}$$

Setting $k_i^{(1)} = 0$ tells us that this is a special case and we get $k_i$ from $k_i^{(0)}$ as

$$k_i = |k_i^{(0)} \cdot m_0' + m_1' |_R - 1 |_R.$$  

We deal with the carry as it was done when $k_i^{(1)} \neq 0$, this is detailed in the algorithm.

One notices it might be necessary to handle the last carry $C$ generated by the recoding of $k_{\ell-1}$ with a final correction. This gives a final coefficient $k'_\ell = -C$ which satisfies $|k'_\ell| < m_1$. Finally, this leads to the recoding algorithm shown in Algorithm 5.

**Algorithm 5** Multiplicative Splitting Recoding with $R = m_0 m_1$

**Require:** An RNS base $\{m_0, m_1\}$, a radix $R = m_0 \cdot m_1$ and an exponent $k = \sum_{i=0}^{\ell} k_i R^i$.

**Ensure:** $(k_i^{(0)}, k_i^{(1)}), 0 \leq i < \ell, (C)$ the multiplicative splitting recoding of $k$ in radix $R = m_0 m_1$.

1. $C \leftarrow 0$
2. for $i$ from 0 to $\ell - 1$
3. $k_i \leftarrow k_i - C, C \leftarrow 0$
4. if $k_i < 0$ then
5. $k_i \leftarrow k_i + R, C \leftarrow 1$
6. $k_i^{(0)} \leftarrow |k_i|_m, k_i^{(1)} \leftarrow |k_i|m_1$.
7. if $k_i^{(1)} = 0$ then
8. \(k_i^{(0)}, k_i^{(1)} \leftarrow (|k_i^{(0)} + 1|_m, 0)\)
9. $C \leftarrow C + \left( |k_i^{(0)} \cdot m_0' + m_1' |_R - 1 \right) / R$
10. else
11. $k_i^{(0)} \leftarrow |k_i^{(0)} \cdot (k_i^{(1)})^{-1} |_m$
12. $k_i^{(1)} \leftarrow k_i^{(1)}$
13. $C \leftarrow C + |k_i^{(1)} \cdot (k_i^{(0)} \cdot m_0' + m_1' |_R / R$
14. return $(k_i^{(0)}, k_i^{(1)}), 0 \leq i < \ell, k'_\ell = -C$.

At the end the recoded exponent $k = \sum_{i=0}^{\ell} k_i R^i$ has most of its digits $k_i'$ expressed as a product $k_i^{(1)} \cdot |k_i^{(0)} \cdot m_0' + m_1' |_R$ and $k_i^{(1)}$ is of size $m_1$ while $|k_i^{(0)} \cdot m_0' + m_1' |_R$ is indexed with $k_i^{(0)}$ which is of size $m_0$. 
Example 1 We present here an example of the $m_0m_1$ recoding with an exponent size of 20 bits ($0 < k < 2^{20}$), and $B = \{11, 8\}$ (i.e. $m_0 = 11, m_1 = 8$). Thus, in this case, one has the radix $R = m_0 \cdot m_1 = 88$, $\ell = \lceil 20/\log_2(88) \rceil = 4$, and also

\begin{align*}
m_0 &= 8 \cdot 8^{-1}_{11} = 56, \\
m_1 &= 11 \cdot 11^{-1}_8 = 33.
\end{align*}

Let us take $k = 936192_{10}$, the random exponent. By rewriting $k$ in radix-$R$, one has

\[ k = 48 + 78 \cdot 88 + 32 \cdot 88^2 + 1 \cdot 88^3. \]

We now use Algorithm 1, which consists of a for loop (Steps 2 to 13).

- In the first iteration ($i = 0$), one has $k_0 = 48$.
  - One has $C \leftarrow 0$ and one skips the if-tests steps 4 to 5 since $k_0 \geq 0$.
  - Step 6, one computes the RNS representation in base $B$ of $k_0 = 48$:
    \[ (k_0^{(0)}, k_0^{(1)}) \leftarrow (k_0^{(0)} \cdot 11, k_0^{(1)}) = (5, 0). \]
    and the carry
    \[ C \leftarrow C + \left( \lceil (k_0^{(0)} \cdot 56 + 33)_{88} - 1 \rceil / 88 \right) = 0 \]
- In the second iteration ($i = 1$), one has $k_1 = 78$.
  - One has $C \leftarrow 0$ and one skips the if-test of Steps 4 to 5 since $k_1 \geq 0$.
  - Step 6, one computes the RNS representation in base $B$ of $k_1 = 78$:
    \[ (k_1^{(0)}, k_1^{(1)}) \leftarrow (k_1^{(0)} \cdot 11, k_1^{(1)}) = (6, 1). \]
    and the carry
    \[ C \leftarrow C + \left( \lceil (k_1^{(1)} \cdot 56 + 33)_{88} / 88 \rceil \right) = 3 \]
- In the third iteration ($i = 2$), one has now $k_2 \leftarrow k_2 - C = 29$.
  - The RNS representation in base $B$ of $k_2$ is $k_2^{(0)} = 7, k_2^{(1)} = 5$.
  - The steps 10-13 give $C \leftarrow 2$, and
    \[ (k_2^{(0)}, k_2^{(1)}) \leftarrow (8, 5). \]
Without providing all the remaining details, one finally obtains the values returned by the algorithm:

\[ ((5, 0), (2, 6), (8, 5), (3, 7)), \text{ and } k_4' = -C = -2. \]

3.2 Exponentiation with a multiplicative splitting recoding in radix $R = m_0m_1$

We first rewrite the exponentiation using the recoding of $k = \sum_{i=0}^\ell k'_i R^i$ of the previous subsection as follows:

\[ g^k \text{ mod } p = g^{\sum_{i=0}^\ell k'_i R^i} = g^{k'_i R^i} \cdot \prod_{i=0}^{\ell-1} g^{k'_i R^i}, \quad (7) \]

where each term $g^{k'_i R^i}$ satisfy one of the following three cases:

- When $k'_i \neq 0$ and $i < \ell$:
  \[ g^{k'_i R^i} = g^{k'_i (R^i | k'_{i0} m_0 + m_1 | n)} \]
- When $k'_i = 0$ and $i < \ell$:
  \[ g^{k'_i R^i} = g^{R^i | k'_{i0} m_0 + m_1 | n} \cdot g^{-R^i}. \]
- When $i = \ell$ we have $k'_\ell \leq 0$ which implies that $g^{k'_\ell R^\ell} = (g^{-R^\ell})^{k'_\ell}$.

In order to compute the fixed-base exponentiation $g^k$, one stores the following values:

\[ T[i][j] = g^{R^i | j \cdot m_0 + m_1 | n}, \quad \text{with } \begin{cases} 0 \leq i \leq \ell - 1, \\ 0 \leq j < m_0. \end{cases} \]

and one also stores the following inverses:

\[ T[i][-1] = g^{-R^i} \quad \text{with } 0 \leq i \leq \ell. \]

We use $Y_j$ to denote the product of $g^{R^i | k'_{i0} m_0 + m_1 | n}$ for each $i$ such that $k'_i = j$. In other words for $j \neq 0$

\[ Y_j = \left\{ \begin{array}{ll}
(\prod_{i=0}^{\ell \cdot j} T[i][k'_{i0}]) \cdot T[i][-1] & \text{if } k'_i = j, \\
(\prod_{i=0}^{\ell \cdot j} T[i][k'_{i0}]) & \text{otherwise,}
\end{array} \right. \]

and

\[ Y_0 = \prod_{k'_i = 0, i \leq \ell} T[i][k'_{i0}] \times T[i][-1]. \]

We can then rewrite the expression of $g^k$ in (7) in terms of $Y_j$ for $j = 0, \ldots, m_1 - 1$ as follows:

\[ g^k = Y_0 \times \prod_{j=1}^{m_1-1} Y_j^j. \]

Each individual exponentiation $Y_j^j$ is performed with a square-and-multiply approach, which is more efficient than performing $j - 1$ multiplications, even for small $m_1$. This approach is depicted in Algorithm 2.

One important drawback of the above algorithm is that it is not constant time, due to the if branching attached to the condition $k'_i = 0$. 


Algorithm 6 Fixed-base exponentiation with multiplicative splitting with radix $R = m_0m_1$

**Require**: An RNS base $\{m_0, m_1\}$, a radix $R = m_0m_1$, the exponent $k = \sum_{i=0}^{\ell-1} k_R \cdot r_i$ and $\{(k^{(0)}_i, k^{(1)}_i), 0 \leq i < \ell, (k^{(1)}_i)\}$ the $m_0m_1$ recoding of $k$ and $g \in G$.

**Ensure**: $A = g^k$

1: **Offline precomputation.** Store $T[i][j] \leftarrow g^{R^j \cdot m_0 \cdot m_1 \cdot i} \mod p$ with $0 \leq i < \ell, 0 \leq j < m_0, T[i][\ell] \leftarrow g^{-R^i}, 0 \leq i < \ell$
2: $X \leftarrow 1, Y_j \leftarrow 1$ for $0 \leq j < m_1$
3: **for** $i$ from 0 to $\ell - 1$ **do**
4: **if** $k^{(1)}_i = 0$ **then**
5: **else**
6: $Y_0 \leftarrow T[i][k^{(0)}_i] \times T[i][\ell] \leftarrow 1$
7: **else**
8: $Y_0 \leftarrow Y_0 \times T[i][k^{(0)}_i] \times T[i][\ell] \leftarrow 1$
9: **else**
10: **if** $Y_{k^{(1)}_i} = 1$ **then**
11: $Y_{k^{(1)}_i} \leftarrow T[i][k^{(0)}_i]$
12: **else**
13: $Y_{k^{(1)}_i} \leftarrow Y_{k^{(1)}_i} \times T[i][k^{(0)}_i]$
14: **if** $k^{(1)}_i \neq 0$ **then**
15: $Y_{k^{(1)}_i} \leftarrow Y_{k^{(1)}_i} \times T[i][\ell] \leftarrow 1$
16: $W \leftarrow$ size of $m_1$ in bits
17: **for** $i$ from $W - 1$ downto 0 **do**
18: $X \leftarrow X^2$
19: **for** $j$ from $m_1 - 1$ downto 0 **do**
20: **if** bit $i$ of $j$ is non zero **then**
21: $X \leftarrow X \times Y_j$
22: **return** $(X \times Y_0)$

**Example 2** We present the computation of $g^k \mod p$ using Algorithm 6 we take $B = \{11, 8\}$ (i.e. $m_0 = 11, m_1 = 8$). In terms of storage, one computes the values $T[i][j] = g^{R^j \cdot m_0 \cdot m_1 \cdot i} \mod p$ with $0 \leq i < \ell - 1$. One has the values $\{33, 1, 57, 25, 81, 49, 17, 73, 41, 9, 65\}$ for $j = m_0 + m_1 \mod R$ when $0 \leq j < 11$. This leads to $T[i][0..10] = \{g^{59833}, g^{8981}, g^{88857}, g^{88825}, g^{88881}, g^{88849}, g^{88817}, g^{88873}, g^{88841}, g^{88889}, g^{88865}\}$. The trace of Algorithm 6 for the computation of $g^k$ and $k = 936192$ using the recoding obtained in Example 1 is provided in Table 1.

3.3 Complexity

For the amount of precomputed data, one can notice that it is equal to $(m_0 + 1) \times \ell + 1$ elements.

The complexity of online computation in Algorithm 6 is evaluated step by step in Table 3 for the average case. The number of multiplications ($M$) is evaluated as follows.

<table>
<thead>
<tr>
<th>Iter. (loop 3)</th>
<th>Exp. coeff</th>
<th>Step</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i = 0$</td>
<td>$k^{(0)}_0 = 5$</td>
<td>$i = 1$</td>
<td>$Y_0 \leftarrow T[0][k^{(0)}_0] \times T[0][\ell] = g^{598} \times g^{1} = g^{598}$</td>
</tr>
<tr>
<td>$i = 1$</td>
<td>$k^{(0)}_1 = 2$</td>
<td>$i = 2$</td>
<td>$Y_0 \leftarrow T[1][k^{(0)}_1] = g^{888} \cdot 57 = 56016$</td>
</tr>
<tr>
<td>$i = 2$</td>
<td>$k^{(0)}_2 = 8$</td>
<td>$i = 3$</td>
<td>$Y_0 \leftarrow T[2][k^{(0)}_2] = g^{888} \cdot 41 = 317504$</td>
</tr>
<tr>
<td>$i = 3$</td>
<td>$k^{(0)}_3 = 3$</td>
<td></td>
<td>$Y_0 \leftarrow T[3][k^{(0)}_3] = g^{888} \cdot 25 = 170368000$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$i = 15$</td>
<td>$T_2 \leftarrow T[4][\ell] = g^{88} \cdot 1 = g^{88}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$i = 17$</td>
<td>$g^k = Y_0 \times \prod_{j=1}^{m_1 - 1} Y_j$ $= 59969536$</td>
</tr>
</tbody>
</table>

The costs of Steps 6 to 15 follow directly from Algorithm 6 and are detailed in Table 3.

The first squaring in Step 18 skipped since $X = 1$, leading to a cost of $W - 1$ squarings.

The multiplications in Steps 21 and 22 are performed only in case of $Y_j \neq 1$. This means that in the worst case we save the first multiplication which is an affectation : this is the case considered in Table 3.

For the sake of simplicity, we denote by $H$ the sum of the $j$ Hamming weights for each $j$ from $m_1 - 1$ downto 1 (for loop in Step 1c). The value of $H$ is shown in Table 2 for different practical values of $m_1$.

<table>
<thead>
<tr>
<th>$m_1$</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>$H$</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>12</td>
<td>13</td>
</tr>
</tbody>
</table>

4 Fixed base exponentiation with multiplicative splitting with $R$ prime

In this section we present a novel recoding algorithm based on multiplicative splitting modulo $R$ prime. We will show that the resulting exponentiation algorithm can be made constant time.

Table 1 Example of an execution trace for an exponentiation based on multiplicative splitting recoding with $R = m_0m_1$
Table 3 Complexity of exponentiation based on multiplicative splitting recoding with $R = m_0 m_1$

<table>
<thead>
<tr>
<th>Step</th>
<th>Operation</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1x Step 6</td>
<td>$T[i]</td>
<td>k_i^{(0)} \times T[i]</td>
</tr>
<tr>
<td>$(t/m - 1) \times$ Step 8</td>
<td>$Y_0 \times T[i]</td>
<td>k_i^{(0)} \times T[i]</td>
</tr>
<tr>
<td>$(m_1 - 1) \times$ Step 11</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$(t/m - 1) \times$ Step 13</td>
<td>$Y(i+1)</td>
<td>k_i^{(1)}</td>
</tr>
<tr>
<td>$(W - 1) \times$ Step 18</td>
<td>$X \leftarrow X' \times$</td>
<td>1 S</td>
</tr>
<tr>
<td>$(N - 1) \times$ Step 21</td>
<td>$X \times Y_j$</td>
<td>1 M</td>
</tr>
<tr>
<td>1x Step 22</td>
<td>$(X \times Y_0)$</td>
<td>1 M</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td>$(t/m - 1)m_1 + n + 1) M + (W - 1) S$</td>
<td></td>
</tr>
<tr>
<td><strong>TOTAL STORAGE</strong></td>
<td>$(m_0 + 1) \times \ell + 1$ elements of $G$</td>
<td></td>
</tr>
</tbody>
</table>

4.1 Digit multiplicative splitting for prime radix $R$

We present in this subsection a variant of the multiplicative splitting to the case of a prime radix $R$. When $R$ is a prime we can use a multiplicative splitting modulo $R$ based on an extension of the half-size multiplicative splitting of [19]. Our goal is to get the following splitting

$$k_i = k_i^{(0)}(k_i^{(1)})^{-1} \mod R$$

for a fixed bound $0 < c < R$.

4.1.1 Multiplicative splitting modulo a prime $R$

The multiplicative splitting modulo a prime radix $R$ is based on the extended Euclidean algorithm. We briefly review this algorithm. We consider a prime integer $R$ and $0 < k < R$. Then $k$ and $R$ are pairwise prime $\gcd(k, R) = 1$. The Euclidean algorithm computes $\gcd(k, R)$ through a sequence of modular reductions:

$$r_0 = R, r_1 = k, r_2 = r_0 \mod r_1, \ldots, r_{j+1} = r_{j-1} \mod r_j, \ldots$$

The sequence of remainders $r_j$ satisfies

$$\gcd(r_j, r_{j+1}) = \gcd(R, k)$$

and is strictly decreasing and thus reaches 0 after some iterations. The last $r_t \neq 0$ satisfies $r_t = \gcd(k, R) = 1$. The extended Euclidean algorithm computes a Bezout relation

$$uR + vk = \gcd(k, R)$$

by maintaining two sequences of integers $u_j$ and $v_j$ satisfying:

$$u_j R + v_j k = r_j, \text{ for } j = 0, 1, \ldots, \ell. \quad (9)$$

The sequence $v_j$ is an increasing sequence in magnitude starting from $v_0 = 0$ and $v_1 = 1$. The multiplicative splitting of [8] can then be obtained from [9] where we take $j$ such that $r_j \in [0, c]$ and $v_j \in [0, R/c]$ and by taking $k_i^{(0)} = r_j$ and $k_i^{(1)} = v_j$. The following lemma establishes this property.

**Lemma 1** If one chooses $c \in [0, R]$, there exists $j$ such that $|r_j| \geq c$ and $r_{j+1} < c$ and at the same time $|v_j| \leq R/c$ and $|v_{j+1}| \geq R/c$.

The proof of the lemma is given in the appendix.

This leads to the method shown in Algorithm 7 for multiplicative splitting modulo a prime radix $R$. In this algorithm a third variable $s$ is used for the sign of the multiplicative splitting.

**Algorithm 7** Truncated Extended Euclidean Algorithm (TruncatedEEA($k, R, c$))

**Require:** $k \in \mathbb{Z}$, the prime radix $R$, and $c$, the upper bound for $k_i^{(1)}$.

**Ensure:** $(s, k_0^{(0)}, k_1^{(1)})$, such as $k = |s| k_0^{(0)} \times (k_1^{(1)})^{-1} R$ with $0 \leq k_0^{(0)} < c$ and $0 \leq k_1^{(1)} \leq R/c$ and $c \in [-1, 1]$ when $\gcd(k, R) = 1$.

1. if $\gcd(k, R) = R$ then
2. return $(1, 0, 0)$
3. else
4. $u_0 \leftarrow 1, v_0 \leftarrow 0, u_1 \leftarrow 0, v_1 \leftarrow 1, r_1 \leftarrow |k|_R$
5. while $(r_1 \geq c)$ do
6. $q \leftarrow |r_0|/r_1$, $r_2 \leftarrow |r_0| r_2$
7. $u_2 \leftarrow u_0 - q \cdot u_1, v_2 \leftarrow v_0 - q \cdot v_1$
8. $(u_0, v_0, r_0) \leftarrow (u_1, v_1, r_1)$
9. $(u_1, v_1, r_1) \leftarrow (u_2, v_2, r_2)$
10. $s \leftarrow \text{sign}(v_1), k_0^{(0)} \leftarrow r_1, k_1^{(1)} \leftarrow |v_1|$
11. return $(s, k_0^{(0)}, k_1^{(1)})$

4.1.2 Recoding the exponent

We now present our recoding approach for an integer $k$ given in radix-$R$ representation:

$$k = \sum_{i=0}^{\ell-1} k_i R^i, \text{ with } \ell = \lceil \log_2(R) \rceil.$$

We choose a splitting bound $c$ and we consider a digit $k_i \neq 0$. Using Algorithm 3, we get $s_i, k_i^{(0)}$ and $k_i^{(1)}$ such that

$$k_i = s_i k_i^{(0)}(k_i^{(1)})^{-1} \mod R$$

with

$$k_i^{(0)} \in [0, c], \quad \{k_i^{(1)} \in [0, R/c]. \quad (10)$$
We put apart the case \( k_i = 0 \) which is recoded as \((1, 0, 0)\) (cf. Step 2 of Algorithm \[7\]). We handle the reduction modulo \( R \) as follows:

\[
C = (s_i k_i^{(0)} | (k_i^{(1)})^{-1} | R - k_i) / R \text{ (exact quotient),}
\]

\[
k_i = s_i k_i^{(0)} | (k_i^{(1)})^{-1} | R - CR.
\]

One notices that \( C \) satisfies \(-c \leq C < c\). We then consider \( C \) as a carry that we subtract to \( k_{i+1} \).

We obtain an expression \( k = \sum_{i=0}^{t-1} k_i R^i \) of \( k \) in radix \( R \) such that each digit \( k_i' = s_i k_i^{(0)} | (k_i^{(1)})^{-1} | R \) is given in a multiplicative splitting form. The last coefficient \( k_t' = -C \) is necessary to handle the last carry. The resulting recoding algorithm is shown in Algorithm \[8\].

**Algorithm 8** Multiplicative Splitting Recoding for \( R \) Prime

**Require:** \( R \) prime, \( k = \sum_{i=0}^{t-1} k_i R^i \), and \( c \) the splitting bound.

**Ensure:** \( \{(s_i, k_i^{(0)}, k_i^{(1)}) \mid 0 \leq i < \ell, (k_i') \} \) the multiplicative splitting recoding of \( k \).

1. \( C \leftarrow 0 \)
2. for \( i \) from 0 to \( \ell - 1 \) do
   3. \( k_i \leftarrow k_i - C \)
   4. \( s_i, k_i^{(0)}, k_i^{(1)} \leftarrow \text{TruncatedEEA}(k_i, R, c) \)
   5. \( C \leftarrow (s_i k_i^{(0)} | (k_i^{(1)})^{-1} | R - k_i) / R \) \( \text{//exact quotient} \)
3. return \( \{(s_i, k_i^{(0)}, k_i^{(1)}) \mid 0 \leq i < \ell, (k_i' = -C)\} \)

*Example 3* We present an example of multiplicative splitting recoding for a prime radix \( R = 89 \) with an exponent size \( t = 20 \) bits (0 < \( k < 2^{20} \)). In this case, one has \( t = [20 / \log_2(89)] = 4 \). One also sets \( c = 2^3 = 8 \), and then, \( |R/c| = 12 \). Let us take \( k = 90164410 \), the random exponent. By rewriting \( k \) in radix-\( R \), one has

\[
k = 74 + 73 \cdot 89 + 24 \cdot 89^2 + 1 \cdot 89^3.
\]

The execution trace of Algorithm \[8\] is provided in Table \[4\].

### 4.2 Exponentiation Algorithm with multiplicative splitting recoding in a prime radix \( R \)

We now present an exponentiation algorithm which takes advantage of the multiplicative recoding given in Section 4.1.2. One wants to compute

\[
g^k = g^{\sum_{i=0}^{t-1} k_i R^i} = g^{k_t' R^t} \cdot \prod_{i=0}^{t-1} g^{k_i' R^i} \quad (11)
\]

with

\[
g^{k_t' R^t} = g^{s_t k_t^{(0)} | (k_t^{(1)})^{-1} | R R^t}, \text{ if } k_t^{(1)} \neq 0,
\]

\[
g^{k_i' R^i} = 1, \text{ if } k_i^{(1)} = 0(\text{this corresponds to } k_i = 0).
\]

Table 4 Example of an execution trace of Algorithm \[8\]

<table>
<thead>
<tr>
<th>Iter.</th>
<th>Step</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3:</td>
<td>( k_0 = 74 ) does not change since ( C = 0 )</td>
</tr>
<tr>
<td>1</td>
<td>3:</td>
<td>( k_1 = 73 + 1 = 74 ) since ( C = -1 )</td>
</tr>
<tr>
<td>1</td>
<td>4:</td>
<td>( s_1 = -1, k_1^{(0)} = 1, k_1^{(1)} = 6 )</td>
</tr>
<tr>
<td>1</td>
<td>5:</td>
<td>( C \leftarrow (s_1 k_1^{(0)}</td>
</tr>
<tr>
<td>2</td>
<td>3:</td>
<td>( k_2 = 24 + 1 = 25 ) since ( C = -1 )</td>
</tr>
<tr>
<td>2</td>
<td>4:</td>
<td>( s_2 = -1, k_2^{(0)} = 3, k_2^{(1)} = 7 )</td>
</tr>
<tr>
<td>2</td>
<td>5:</td>
<td>( C \leftarrow (s_2 k_2^{(0)}</td>
</tr>
<tr>
<td>3</td>
<td>3:</td>
<td>( k_3 = 1 + 2 = 3 ) since ( C = -2 )</td>
</tr>
<tr>
<td>3</td>
<td>4:</td>
<td>( s_3 = 1, k_3^{(0)} = 3, k_3^{(1)} = 1 )</td>
</tr>
<tr>
<td>3</td>
<td>5:</td>
<td>( C \leftarrow (s_3 k_3^{(0)}</td>
</tr>
</tbody>
</table>

\(((-1, 1, 6), (-1, 1, 6), (-1, 3, 7), (1, 3, 1)) \) and \( k'_4 = C = 0 \)

In order to compute the fixed-base exponentiation \( g^k \text{ mod } p \), one stores the following values:

\[
T[i][s][j] = g^{R^{-s} \cdot |j|-1 \cdot n}, \text{ with } \begin{cases} \{0 \leq i \leq \ell - 1, \ 1 \leq j \leq |R/c|, \\ s \in \{-1, 1\}, \end{cases}
\]

\[
T[i][s][0] = 1 \text{ with } s \in \{-1, 1\},
\]

\[
T[i][s] = g^{R^s} \text{ with } s \in \{-1, 1\}.
\]

One denotes \( Y_j \) the product of the terms \( g^{s_i |(k_i^{(1)})^{-1} | n \cdot R^i} \) such that \( k_i^{(1)} = j \). This means that for \( j \neq |k'_i| \)

\[
Y_j = \left( \prod_{k_i^{(0)} = j} T[i][s_i][k_i^{(1)}] \right) .
\]

and for \( j = |k'_i| \) one has

\[
Y_j = \left( \prod_{k_i^{(0)} = j} T[i][s_i][k_i^{(1)}] \right) \times T[i][\text{sign}(k'_i)].
\]

We can then rewrite the products in \( (11) \) in terms of \( Y_j \) as follows:

\[
g^k = \prod_{j \in \{1, \ldots, c-1\}} Y_j^j.
\]

Every individual exponentiation \( Y_j^j \) is performed with a square-and-multiply approach, which is more efficient than performing \( j - 1 \) multiplications, even for small \( c \). This finally leads to the exponentiation shown in Algorithm \[9\].

The above algorithm can be implemented in a constant time fashion. Indeed there is no control attached to the digits of the exponent. Then, the algorithm consists in a constant and regular sequence of multiplications and squarings as soon as a multiplication with a 1 is computed as any other multiplication.
This brings us to store the following values in $G$.

Example 4 We consider the exponent $k = 901644_{10}$ along with the multiplicative splitting recoding computed in Example 3.

\[(−1,1,6), (−1,1,6), (−1,3,7), (1,3,1)\] and $k'_4 = 0$.

We present the computation of $g^k$ using Algorithm 9. In terms of storage, one computes the values:

$$T[i][s][j] = g^{R^j \times s \times j^{-1} \times n}$$

with $0 \leq i \leq \ell -1$, $1 \leq j \leq \lceil R/c \rceil$, $s \in \{-1,1\}$.

One has the following values of $j^{-1} \times n$ for $1 \leq j \leq 12$

\[1, 45, 30, 67, 18, 15, 51, 78, 10, 9, 81, 52].

This brings us to store the following values in $G$:

$$T[i][1] = \{g^{89}, g^{89 \times 45}, g^{89 \times 30}, g^{89 \times 67}, g^{89 \times 18}, g^{89 \times 15}, g^{89 \times 51}, g^{89 \times 78}, g^{89 \times 10}, g^{89 \times 9}, g^{89 \times 81}, g^{89 \times 52}\}$$

$$T[i][-1] = \{g^{89}, g^{89 \times 45}, g^{89 \times 30}, g^{89 \times 67}, g^{89 \times 18}, g^{89 \times 15}, g^{89 \times 51}, g^{89 \times 78}, g^{89 \times 10}, g^{89 \times 9}, g^{89 \times 81}, g^{89 \times 52}\}.$$}

Table 5 Example of an execution trace for an exponentiation based on multiplicative splitting recoding with $R$ prime

<table>
<thead>
<tr>
<th>Iter.</th>
<th>Step</th>
<th>Coeff</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>i = 0</td>
<td>4</td>
<td>$k_0^{(1)} = 1$</td>
<td>$y_1 = y_1 \times T[0][c][k_0^{(1)}]_{10}$</td>
</tr>
<tr>
<td>i = 1</td>
<td>4</td>
<td>$k_1^{(1)} = 1$</td>
<td>$y_1 = y_1 \times T[1][c][k_1^{(1)}]_{10}$</td>
</tr>
<tr>
<td>i = 2</td>
<td>4</td>
<td>$k_2^{(1)} = 1$</td>
<td>$y_1 = y_1 \times T[2][c][k_2^{(1)}]_{10}$</td>
</tr>
<tr>
<td>i = 3</td>
<td>4</td>
<td>$k_3^{(1)} = 1$</td>
<td>$y_1 = y_1 \times T[3][c][k_3^{(1)}]_{10}$</td>
</tr>
</tbody>
</table>

4.3 Complexity

Let us now evaluate the complexity of Algorithm 9. Concerning the amount of storage it consists in $2(\lceil R/c \rceil \times +1) \ell + 2$ elements of $G$.

For the online complexity, we evaluate the cost of each step of Algorithm 9 based on the following:

- the multiplications in Step 4 are performed even in case of $y_i^{(0)} = 1$, in order to ensure the constant time of the computation;
- the same applies for Step 5.

The number of operations in the final reconstruction is evaluated as follows:

- the squaring in Step 8 is not performed in the first loop iteration ($X = 1$);
- this first multiplication in Step 11 is skipped since it is an affecation. The other multiplications in Step 11 are performed even in case of $y_i = 1$, again to ensure a constant computation time.

We denote by $H$ the sum of the $j$ Hamming weights for each $j$ from $c - 1$ downto 1 (for loop in Step 7). The value of $H$ is as follows for the different values of $c$ can be found in Table 2.

The contribution of each step is given in Table 6 along with the total complexity.

5 Complexity and experiment comparison

5.1 Complexity comparison

In Table 4 we give the complexities in terms of the number of online operations and storage amount of the state of the art approaches (Section 2) and the two proposed approaches in Section 3 and 4. All the approaches presented in the above table can be implemented in constant time except the Square-and-multiply, Fixed base NAF and the proposed approach with $R = m_0 m_1$. 

\[\text{Iter.} \quad \text{Step} \quad \text{Coeff} \quad \text{Value} \]

<table>
<thead>
<tr>
<th>Iter.</th>
<th>Step</th>
<th>Coeff</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>i = 0</td>
<td>4</td>
<td>$k_0^{(1)} = 1$</td>
<td>$y_1 = y_1 \times T[0][c][k_0^{(1)}]_{10}$</td>
</tr>
<tr>
<td>i = 1</td>
<td>4</td>
<td>$k_1^{(1)} = 1$</td>
<td>$y_1 = y_1 \times T[1][c][k_1^{(1)}]_{10}$</td>
</tr>
<tr>
<td>i = 2</td>
<td>4</td>
<td>$k_2^{(1)} = 1$</td>
<td>$y_1 = y_1 \times T[2][c][k_2^{(1)}]_{10}$</td>
</tr>
<tr>
<td>i = 3</td>
<td>4</td>
<td>$k_3^{(1)} = 1$</td>
<td>$y_1 = y_1 \times T[3][c][k_3^{(1)}]_{10}$</td>
</tr>
</tbody>
</table>

- 5: $k_4 = 0$ 

| - 7 | $g_i = \prod_{j=1}^{i} Y_j$ |

| - to | $g_i = g_i^{300998} \times 0.05$ |

| 11: | $g_i = g_i^{300998} \times 0.05$ |
Let us first see when the Fixed-base Comb method is better than the Fixed-base Radix-$R$ exponentiation. We denote $w_C$ the window size of the Comb method and $w_R$ the one of the Radix-$R$ method. In order to have both methods with the same number of online operations in $G$, we take $w_C = 2w_R$: in this case, both methods require $t/w_R$ online operations in $G$. Then, considering the storage amount when $w_C = 2w_R$, one can see that the Comb method requires $2^{wR}$ while the Radix-$R$ method needs $\frac{t}{w_R} 2^{wR}$ elements of $G$. In other words, for a fixed number $t/w_R$ of online computation, the Comb method is better than the Radix-$R$ as soon as $2^{wR} < \frac{t}{w_R}$ which is the case for small $w_R$, i.e., for small amount of storage.

If we now consider the Fixed base NAF$_w$, we can notice that it does not compare favorably with the radix-$R$ approach. Indeed for $e = 1$ we would have almost the same number of online multiplications whereas the amount of data in the NAF$_w$ is larger by a factor of $w$. For larger value of $e$ the number of squarings would increase quickly rendering the approach not competitive. Moreover the Fixed base NAF$_w$ has the major drawback to not be constant time.

It is more difficult to formally compare the proposed approaches with the Comb and Radix-$R$ approaches. Indeed, they involve a third parameter ($c$ or $m_1$), which means that for a fixed number of online operations, we would have to find the proper parameter which minimizes the amount of storage. We can still notice that for a given $c$ (resp. $m_1$) we divide by $c$ (resp. $m_1$) the amount of storage compared to the Radix-$R$ approach while having an increase of online computation ($H$ and $W$). This means that the proposed approaches can be competitive only for small $c$ and $m_1$.

To have a clearer idea of the impact of the proposed approach so we follow the strategy used in [17]. Indeed, for practical sizes of group and exponent and for different level of online operations, we evaluate the best choice of parameters which minimizes the amount of precomputation. In the sequel we give the results for DSA and ECDSA, for the fields and curves recommended by the NIST.

5.2 Complexities and timings for modular exponentiation

In this subsection we focus on exponentiation in $(\mathbb{Z}/p\mathbb{Z})^*$ used in DSA. We evaluate and compare the complexities of the best method of the literature, i.e., Fixed-base Comb (Algorithm 3 and Fixed-base Radix-$R$ (Algorithm 2), with the complexity of our proposed approaches based on a multiplicative splitting recording of the exponent (Algorithm 6 for $R = m_0m_1$ and Algorithm 9 for $R$ prime).

In the sequel of this subsection, we provide complexity evaluations in terms of modular multiplications MM, under the assumption of modular squaring $M = 0.86$ MM, which is the average value of our implementations for the NIST DSA recommended field sizes. We warn the reader to keep in mind that the Fixed-base Comb, Radix-$R$ and Algorithm 9 are constant time, and that Algorithm 6 is not, i.e., the only one weak against timing attacks.

The NIST provides recommended key sizes and corresponding field sizes (respectively the size of the primes $q$ and $p$, see NIST SP800-57 [4]). This standardized sizes are as follows:

Fig. 4 gives the general behavior of the four algorithms in terms of storage ($y$ axis) with respect to the number of online operations ($x$ axis). In the figure, we present three of the field sizes recommended in the NIST standards (see [3]) and the behavior is roughly the same for all sizes, although the benefit of our approach with $R = m_0m_1$ is lower for smaller sizes. One can see that the Fixed-base Comb method is the best for small storage amount. Our $m_0m_1$ approach (Algorithm 6) is better for larger amount of storage, however, the Fixed-base Radix-$R$ method is the best when the storage is increasing. One can see that the $R$ prime multiplicative splitting approach (Algorithm 3) is less efficient than the $R = m_0m_1$ for small storage amounts. The reason is that this requires some additional computations to get a constant time execution, while the $m_0m_1$ approach is not constant time and is thus slightly more efficient. Nevertheless, one can see a range of storage/complexity trades-off where the $R$ prime multiplicative splitting approach is the best of the constant-time ones.

Table 9 shows numerical application of the complexity comparison between the Fixed-base Comb (Algorithm 3), the Fixed-base Radix-$R$ (Algorithm 2) and
Table 7 Complexities and storage amounts of exponentiation algorithm, average case, binary exponent length t.

<table>
<thead>
<tr>
<th>Method</th>
<th>Constant time</th>
<th>#Mul</th>
<th>#Squ.</th>
<th>Storage (#values in G)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Square-and-mult. (Algo. 1)</td>
<td>no</td>
<td>$\frac{1}{2}$</td>
<td>$t - 1$</td>
<td>0</td>
</tr>
<tr>
<td>Fixed-base Radix-$R^{(1)}$ (Algo. 2)</td>
<td>yes</td>
<td>$\frac{1}{2} - 1$</td>
<td>0</td>
<td>$\frac{1}{2^{2^w}}$</td>
</tr>
<tr>
<td>Fixed-base Comb (Algo. 3)</td>
<td>yes</td>
<td>$\frac{1}{2} - 1$</td>
<td>$\frac{1}{2} - 1$</td>
<td>$2^w$</td>
</tr>
<tr>
<td>Fixed base NAF$_w$ (Algo. 4)</td>
<td>no</td>
<td>$\frac{1}{2}(1 - (\frac{1}{w+1}))$</td>
<td>$(e - 1)w$</td>
<td>$\frac{1}{2^{2w-2}}$</td>
</tr>
<tr>
<td>Proposed(*) with $R = m_0m_1$ (Algo. 6)</td>
<td>no</td>
<td>$\frac{1}{w}m_0 + 1 - m_1 + H + 1$</td>
<td>$W - 1$</td>
<td>$(2^w/m_1 + 1)\frac{1}{2} + 1$</td>
</tr>
<tr>
<td>Proposed(*) with $R$ prime (Algo. 9)</td>
<td>yes</td>
<td>$\frac{1}{2} + H$</td>
<td>$W - 1$</td>
<td>$(2^{w+1/c} + 1)\frac{1}{2} + 1$</td>
</tr>
</tbody>
</table>

(*) We assume that $R$ is a $w$ bit integer.

Fig. 1 Complexity comparison, Fixed base modular exponentiation NIST DSA, key size 256, 384 and 512 bits (field size 3072, 7360 and 15360 bits).

Table 8 NIST recommended key and field sizes

<table>
<thead>
<tr>
<th>Security level</th>
<th>80</th>
<th>112</th>
<th>128</th>
<th>192</th>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key size (bits)</td>
<td>1024</td>
<td>2048</td>
<td>3072</td>
<td>7680</td>
<td>15360</td>
</tr>
<tr>
<td>Field size (bits)</td>
<td>1024</td>
<td>2048</td>
<td>3072</td>
<td>7680</td>
<td>15360</td>
</tr>
</tbody>
</table>

the approaches based on our multiplicative splitting recordings (Algorithm 6 and Algorithm 9). For an equivalent number of MMs, we provide the minimum amount of storage. We can notice the following:

- For all key sizes, we do not provide the results for small amount of storage (values for $w < 8$). For such storage, the Fixed-base Comb method is the best. One may notice that the Fixed-base Radix-$R$ approach involves the largest storage amount at this complexity level.

- Comparison of the two proposed approaches: $R = m_0m_1$ vs $R$ prime. We would like to evaluate the improvements provided by the new approach (Algorithm 9) compared to (Algorithm 6) which was presented at WAIFI 2016. The results in Table 9 show that the exponentiation with multiplicative splitting with $R = m_0m_1$ and $R$ prime are close from each other. But the approach with $R = m_0m_1$ is generally slightly better than the one with $R$ prime. But, as noticed earlier, this is the price to pay to get a constant-time algorithm.

- Comparison of constant time approaches. We consider the Fixed-base Comb, Radix-$R$ and multiplicative splitting with $R$ prime approaches. A thorough analysis of the complexities shows that the proposed approach is interesting for intermediate level of online computation. Specifically from Table 9 for a 224 bit key size, one notices that there are not many cases where the proposed multiplicative splitting approach is interesting. However, for the other key sizes $t = 256, 384$ and 512, one can see a lot of cases where the amount of storage is reduced by 50% compared to Comb and Radix-$R$ approaches.
Remark 1 One may notice that the largest memory storage sizes exceed the common values of Random Access Memory, and in some cases, the maximum allowed for the malloc function of the standard C library for memory allocation. Nevertheless, the storage savings proposed by our method and Fixed-base Radix-R ones allow to keep the level of storage under the limit for lower complexities.

5.2.1 Implementation results

Implementation strategies. We review hereafter the main implementation strategies and test process for modular exponentiation for NIST recommended sizes. This applies for the four considered exponentiation algorithms. The algorithms were coded in C, compiled with gcc 4.8.3 and run on the same platform.

- Multi-precision multiplication and squaring. We used the low level functions performing multi-precision multiplication and squaring of the GMP library as building blocks of our codes (GMP 6.0.0, see GMP library [1]). According to the GMP documentation, the classical schoolbook algorithm is used for small sizes, and Karatsuba and Toom-Cook subquadratic methods for size ≥ 2048 bits.

- Modular reduction. This operation implements the Montgomery representation and modular reduction method, which avoid multi-precision division in the computation of the modular reduction. This approach was presented by Montgomery in [13]. We use the block Montgomery algorithm suggested by Bosselaers et al. in [5]. In this algorithm, the multi-precision operations combine full size operand with one word operand and are also available in the GMP library [1].

- Multiplicative splitting recoding with $R = m_0m_1$ and $R$ prime. The conversion in radix-$R$ needs multi-precision divisions. These operations are implemented using the GMP library [1]. The size of these operations is decreasing along the algorithm, and this is managed through GMP. The other operations are classical long integer operations. At Step 11 in Algorithm 5 (resp. Step 5 in Algorithm 8), an inversion modulo $m_0$ (resp. $R$) is required. This operation is performed using the Extended Euclidean Algorithm, over long integer data. For the considered exponent sizes, the cost of the recoding is negligible. This is explained by the small size of the exponent in comparison with the size of the data processed during the modular exponentiation (see Table 3). The timings given in the next subsection include this recoding.

Table 9 Storage amount comparison for Fixed-base Comb, Fixed-base Radix-$R$ and modular exponentiation with multiplicative splitting recoding for NIST recommended exponent sizes

<table>
<thead>
<tr>
<th>Key size $t = 224$ bits</th>
<th>#MM</th>
<th>Fixed-base Comb</th>
<th>Fixed-base Radix-$R$</th>
<th>Multiplicative splitting</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$R = m_0m_1$</td>
<td>$R = m_0m_1$</td>
<td>$R$-prime</td>
</tr>
<tr>
<td>45</td>
<td>127.5 kB</td>
<td>345 kB</td>
<td>108 kB</td>
<td>240 kB</td>
</tr>
<tr>
<td>w = 9</td>
<td>$R = 31$</td>
<td>$(m_0, m_1) = (11, 9)$</td>
<td>$(R, c) = (97, 7)$</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>511.5 kB</td>
<td>594 kB</td>
<td>242 kB</td>
<td>541 kB</td>
</tr>
<tr>
<td>w = 11</td>
<td>$R = 64$</td>
<td>$(16, 7)$</td>
<td>$(179, 3)$</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>409575.6 kB</td>
<td>1386 kB</td>
<td>792 kB</td>
<td>2425.92 kB</td>
</tr>
<tr>
<td>w = 14</td>
<td>$R = 117$</td>
<td>$(127, 7)$</td>
<td>$(179, 3)$</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>32407.65 kB</td>
<td>4249 kg</td>
<td>4174 kg</td>
<td>4489 kg</td>
</tr>
<tr>
<td>w = 17</td>
<td>$R = 677$</td>
<td>$(87, 7)$</td>
<td>$(1223, 3)$</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>222457.6 kg</td>
<td>27054 kg</td>
<td>50409 kg</td>
<td>27954 kg</td>
</tr>
<tr>
<td>w = 21</td>
<td>$R = 941$</td>
<td>$(13441.5)$</td>
<td>$(6241.2)$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key size $t = 256$ bits</th>
<th>#MM</th>
<th>Fixed-base Comb</th>
<th>Fixed-base Radix-$R$</th>
<th>Multiplicative splitting</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$R = m_0m_1$</td>
<td>$R = m_0m_1$</td>
<td>$R$-prime</td>
</tr>
<tr>
<td>46</td>
<td>383 kB</td>
<td>845 kB</td>
<td>241 kB</td>
<td>494 kB</td>
</tr>
<tr>
<td>w = 10</td>
<td>$R = 41$</td>
<td>$(m_0, m_1) = (17, 17)$</td>
<td>$(R, c) = (97, 5)$</td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>1535 kB</td>
<td>1454 kB</td>
<td>579 kB</td>
<td>1116 kB</td>
</tr>
<tr>
<td>w = 12</td>
<td>$R = 91$</td>
<td>$(71, 11)$</td>
<td>$(224, 9)$</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>1227.5 kB</td>
<td>2179.9 kB</td>
<td>2070 kB</td>
<td>3954 kB</td>
</tr>
<tr>
<td>w = 15</td>
<td>$R = 257$</td>
<td>211.6</td>
<td>409.3</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>98023  kB</td>
<td>9816 kB</td>
<td>9642 kg</td>
<td>10207 kg</td>
</tr>
<tr>
<td>w = 18</td>
<td>$R = 961$</td>
<td>1225.6</td>
<td>10399.7</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>1572665 kB</td>
<td>666676 kB</td>
<td>224582 kg</td>
<td>80558 kg</td>
</tr>
<tr>
<td>w = 22</td>
<td>$R = 8467$</td>
<td>35705.3</td>
<td>12907.2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key size $t = 384$ bits</th>
<th>#MM</th>
<th>Fixed-base Comb</th>
<th>Fixed-base Radix-$R$</th>
<th>Multiplicative splitting</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$R = m_0m_1$</td>
<td>$R = m_0m_1$</td>
<td>$R$-prime</td>
</tr>
<tr>
<td>63</td>
<td>1918 kB</td>
<td>4081 kB</td>
<td>969 kB</td>
<td>2274 kB</td>
</tr>
<tr>
<td>w = 11</td>
<td>$R = 11$</td>
<td>$(m_0, m_1) = (11, 11)$</td>
<td>$(R, c) = (127, 6)$</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>15358 kB</td>
<td>10087 kB</td>
<td>3742 kB</td>
<td>7382 kB</td>
</tr>
<tr>
<td>w = 14</td>
<td>$R = 101$</td>
<td>101.11</td>
<td>423.5</td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>12287.5 kB</td>
<td>26655 kB</td>
<td>17284 kB</td>
<td>22381 kB</td>
</tr>
<tr>
<td>w = 17</td>
<td>$R = 677$</td>
<td>3416.6</td>
<td>537.4</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>983038 kB</td>
<td>80357 kB</td>
<td>64708 kB</td>
<td>65837 kB</td>
</tr>
<tr>
<td>w = 20</td>
<td>$R = 257$</td>
<td>2485.6</td>
<td>1010.3</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>7664318 kB</td>
<td>246070 kB</td>
<td>310563.9</td>
<td>2342.55 kB</td>
</tr>
<tr>
<td>w = 23</td>
<td>$R = 8467$</td>
<td>13341.1</td>
<td>19441.3</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>92914338 kB</td>
<td>86712018 kg</td>
<td>56951702 kg</td>
<td>1089342 kg</td>
</tr>
<tr>
<td>w = 26</td>
<td>$R = 37079$</td>
<td>1860397.9</td>
<td>43974.2</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>503316478 kB</td>
<td>17597566 kg</td>
<td>- kg</td>
<td>- kg</td>
</tr>
<tr>
<td>w = 29</td>
<td>$R = 74099$</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key size $t = 512$ bits</th>
<th>#MM</th>
<th>Fixed-base Comb</th>
<th>Fixed-base Radix-$R$</th>
<th>Multiplicative splitting</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$R = m_0m_1$</td>
<td>$R = m_0m_1$</td>
<td>$R$-prime</td>
</tr>
<tr>
<td>86</td>
<td>18386 kB</td>
<td>9814 kB</td>
<td>1940 kB</td>
<td>6084 kB</td>
</tr>
<tr>
<td>w = 11</td>
<td>$R = 59$</td>
<td>$(m_0, m_1) = (13, 13)$</td>
<td>$(R, c) = (163, 9)$</td>
<td></td>
</tr>
<tr>
<td>73</td>
<td>15356 kB</td>
<td>17585 kB</td>
<td>4747 kB</td>
<td>10005 kB</td>
</tr>
<tr>
<td>w = 13</td>
<td>$R = 127$</td>
<td>$(41, 10)$</td>
<td>$(241, 6)$</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>12287.6 kB</td>
<td>40760 kB</td>
<td>16224 kB</td>
<td>29979 kB</td>
</tr>
<tr>
<td>w = 16</td>
<td>$R = 409$</td>
<td>$(19, 11)$</td>
<td>$(293, 5)$</td>
<td></td>
</tr>
<tr>
<td>52</td>
<td>491516 kB</td>
<td>93118 kB</td>
<td>54680 kB</td>
<td>76055 kB</td>
</tr>
<tr>
<td>w = 18</td>
<td>$R = 941$</td>
<td>$(67, 7)$</td>
<td>$(1225, 3)$</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>989846 kB</td>
<td>159909 kB</td>
<td>106185 kB</td>
<td>136971 kB</td>
</tr>
<tr>
<td>w = 19</td>
<td>$R = 2629$</td>
<td>$(13441.1)$</td>
<td>$(2381.3)$</td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>786444864kB</td>
<td>4601172kB</td>
<td>3555743 kB</td>
<td>477551 kB</td>
</tr>
<tr>
<td>w = 22</td>
<td>$R = 6241$</td>
<td>$(431, 7)$</td>
<td>$(6241.2)$</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>62914556 kB</td>
<td>2048419 kg</td>
<td>211380 kg</td>
<td>1949934 kB</td>
</tr>
<tr>
<td>w = 25</td>
<td>$R = 40347$</td>
<td>$(10257.7)$</td>
<td>$(14226.5)$</td>
<td></td>
</tr>
</tbody>
</table>
Test processing. The tests involve a few hundred
datasets, which consist of random exponent inputs
and an exponentiation base with the precomputed
stored values. We compute 2000 times the corre-
sponding exponentiation for each dataset and keep
the minimum number of clock cycles. This avoids the
cold-cache effect and system issues. The tim-
ings are obtained by averaging the timings of all
datasets.

Tests results and comparison. The four considered ex-
ponentiation algorithms were coded in C, compiled with
gcc 4.8.3 and run on the following platform: the CPU
is an Intel XEON® E5-2650 (Ivy bridge), and the op-
erating system is CENTOS 7.0.1406. On this platform,
the Random Access Memory is 12.6 GBytes. One no-
tices that the performance results include the Radix-
R recoding and the multiplicative splitting of the digits
for $R = m_0, m_1$ and $R$ prime.

We show the performance results in Fig. 2 which
gives a global overview. The implementation results
confirm the complexity evaluation, for key sizes of 224,
256, 384, and 512 bits. However, the best results are for
384 and 512 bits.

In Table 10 we provide the most significant results.
The gains shown are roughly in the same order of mag-
nitude as the one of the complexity evaluation. In par-
cular, for the largest key size (512 bits), the storage
of our approach with $R = m_0m_1$ is nearly ten times
less than the one required with the Fixed-base Comb
method, and nearly 14% less than the one required for
the Fixed-base Radix-R method, for the same level of
clock-cycles. In the same time, our approach with $R$
prime gives equivalent results for low levels of storage,
and better results for higher levels of storage.

5.3 Complexities and timings for scalar multiplication

In this subsection, we present complexity results and
timings of the fixed base scalar multiplication over ell-
liptic curves recommended by NIST.

5.3.1 Complexity comparison

In the fixed-base elliptic curve scalar multiplication
case, the main difference with the modular expo-
entiation is the negligible cost of the inversion of a group
element (i.e. an elliptic curve point). This allows to half
the memory requirements, by only storing the points
corresponding of the positive sign $s_i$ in the recoded
coefficients. We provide in appendix C the version of
the scalar multiplication algorithm with multiplicative
splitting with $R$ prime which takes advantage of a cheap
point subtraction.

When computing the complexities, we noticed that
the approach using a multiplicative splitting recoding
with $R = m_0m_1$ was never better than the one with
$R$ prime. In addition, the approach with $R = m_0m_1$
does not provide a constant time computation. That is
why we do not consider the approach with $R = m_0m_1$
in remainder of this subsection. Specifically, we only
deal with constant-time approaches: Fixed-base Comb,
Radix-R and multiplicative splitting with $R$ prime.

We compare explicit complexities for practical situ-
ations, which are the three elliptic curves standardized
by NIST: P256, P384, P521. One can find in [12] the
Weierstrass curve equations of these three NIST curves
which are reviewed in the appendix. For the arithmetic
on these curves, we use the Jacobian coordinate system,
which provides the fastest curve operations. We use
the complexities in terms of operations in $F_p$ of point
addition and doubling for a Weierstrass curves of [2]
to derive the complexity of the considered fixed-base
scalar multiplication. The complexities of the curve op-
erations in terms of the number of modular multiplica-
tions (MM) and squarings (MS) are as follows:

Addition: $12MM + 4MS$,
Doubling: $4MM + 4MS$,
Mixed-Addition: $7MM + 4MS$.

The resulting complexities of the considered scalar mul-
tiplication approaches are reported in Table 11 and
Fig. 3 assuming that $MS = 0.8MM$.

Fig. 3 gives the general behavior of the storage
for a given amount of online computation. This figure
shows that, as expected, for small amount of storage
the Fixed-base Comb is always the best approach. For
larger complexities the proposed approach with a prime
radix $R$ is the best choice.

Considering the results in Table 11 one notices that
for the four considered methods, one has a slight dif-
fERENCE in comparison with the modular exponentia-
tion case. Indeed, for all key sizes and for most of the lev-
els of online computation the proposed approach shows
the lowest amount of storage. This is due to the re-
lative cost of the doubling of point and the mixed and
full Jacobian addition of points:

- Since the doubling is much cheaper than the ad-
  ditions, this is beneficial to the Fixed-base Comb
  method and the proposed approach with $R$ prime.
  Specifically, Fixed-base Comb is the best approach
  for small amount of storage, that is up to several
tens of kilobytes, for the three curves P256, P384
and P521. For larger amounts of storage, the other
methods remain more efficient.
Table 10 Implementation results for modular exponentiation in terms of clock cycles and storage (kB). Test performed on an Intel XEON E5-2650 (Ivy bridge), gcc 4.8.3, CENTOS 7.0.1406.

<table>
<thead>
<tr>
<th>Security level (bits)</th>
<th>Level of clock-cycles</th>
<th>State of the art methods</th>
<th>Proposed approach</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Fixed-base Comb</td>
<td>Radix R</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Time (CC)</td>
<td>Storage (kB)</td>
</tr>
<tr>
<td>112 bits (key 224 bits, field 2048 bits)</td>
<td>220000</td>
<td>221198</td>
<td>1023.5</td>
</tr>
<tr>
<td>128 bits (key 256 bits, field 3072 bits)</td>
<td>148000</td>
<td>140690</td>
<td>65535</td>
</tr>
<tr>
<td>192 bits (key 384 bits, field 7680 bits)</td>
<td>444000</td>
<td>4442590</td>
<td>1918</td>
</tr>
<tr>
<td>256 bits (key 512 bits, field 15360 bits)</td>
<td>150000</td>
<td>15032420</td>
<td>10536</td>
</tr>
</tbody>
</table>

Fig. 2 Performance comparison, Fixed base modular exponentiation NIST DSA, key size 224, 256, 384 and 512 bits (field size 2048, 3072, 7360 and 15360 bits).
• The proposed approach which uses a multiplicative splitting recoding with a prime radix \( R \) is the best for the following levels of online computations: for P256 and \( \#MM \in \{176, ..., 405\} \), for P384 and \( \#MM \in \{264, ..., 475\} \) and for P521 and \( \#MM \in \{352, ..., 651\} \).

Fig. 3 Complexity comparison for constant time Fixed-base scalar multiplication on elliptic Weierstrass curves P256, P384 and P521

Table 11 Storage amount comparison for constant time Fixed-base scalar multiplication over NIST curves P256, P384, P521

<table>
<thead>
<tr>
<th>#MM</th>
<th>Fixed-base Comb</th>
<th>Fixed-base Radix-R</th>
<th>Multiplicative splitting</th>
</tr>
</thead>
<tbody>
<tr>
<td>411</td>
<td>64 kB</td>
<td>162 kB</td>
<td>73 kB</td>
</tr>
<tr>
<td></td>
<td>( w = 10 )</td>
<td>( R = 69 )</td>
<td>( R = 163; C = 5 )</td>
</tr>
<tr>
<td>405</td>
<td>128 kB</td>
<td>222.5 kB</td>
<td>100 kB</td>
</tr>
<tr>
<td></td>
<td>( w = 11 )</td>
<td>( R = 89 )</td>
<td>( 127; r = 4 )</td>
</tr>
<tr>
<td>317</td>
<td>1024 kB</td>
<td>556 kB</td>
<td>334 kB</td>
</tr>
<tr>
<td></td>
<td>( w = 14 )</td>
<td>( R = 283 )</td>
<td>( 571; 3 )</td>
</tr>
<tr>
<td>264</td>
<td>8192 kB</td>
<td>1532.5 kB</td>
<td>1142 kB</td>
</tr>
<tr>
<td></td>
<td>( w = 17 )</td>
<td>( R = 937 )</td>
<td>( 2381; 3 )</td>
</tr>
<tr>
<td>211</td>
<td>65536 kB</td>
<td>6235.6 kB</td>
<td>5581 kB</td>
</tr>
<tr>
<td></td>
<td>( w = 20 )</td>
<td>( R = 4751 )</td>
<td>( 8929; 2 )</td>
</tr>
<tr>
<td>176</td>
<td>1048576 kB</td>
<td>22192 kB</td>
<td>22156 kB</td>
</tr>
<tr>
<td></td>
<td>( w = 24 )</td>
<td>( R = 19747 )</td>
<td>( 66467; 3 )</td>
</tr>
</tbody>
</table>

5.3.2 Implementation results

We now present implementation strategies and results for the constant time fixed-base scalar multiplication over NIST prime curves P256, P384, P521.

Implementation strategies. The implementation strategies and test processes are the same as the ones presented in Subsection 5.2.1 for modular exponentiation. We review most of them and provides the specific strategies adapted to the considered elliptic curves.

• Curve operations. We use the curve operations in Jacobian coordinate system, which provides the lowest complexities for the considered NIST curves. The doubling formula is from [3], the mixed addition is from [15] and the full-addition is from [2].

• Field operations We use the low level functions performing multi-precision addition, subtraction, multiplication and squaring of the GMP library as build-
Table 12 Implementation results for fixed base scalar multiplication for constant time algorithms. Test performed on an Intel XEON E5-2650 (Ivy bridge), gcc 4.8.3, CENTOS 7.0.1406.

<table>
<thead>
<tr>
<th>Security level</th>
<th>Level of</th>
<th>State of the art methods</th>
<th>Proposed approach</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Clock-cycles</td>
<td>Fixed-base Comb</td>
<td>radix R</td>
</tr>
<tr>
<td></td>
<td>Time</td>
<td>Storage</td>
<td>w</td>
</tr>
<tr>
<td>128 bits</td>
<td>(NIST curve P256)</td>
<td>370000</td>
<td>378184</td>
</tr>
<tr>
<td>192 bits</td>
<td>(NIST curve P384)</td>
<td>375000</td>
<td>375854</td>
</tr>
<tr>
<td>256 bits</td>
<td>(NIST curve P521)</td>
<td>360000</td>
<td>361271</td>
</tr>
</tbody>
</table>

Table 13 Recoding tests, for sizes 256, 384 and 521 bits: worst case computation time in clock cycles

<table>
<thead>
<tr>
<th>Scalar size</th>
<th>Recoding</th>
<th>256 bits</th>
<th>384 bits</th>
<th>521 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Radix R conversion</td>
<td>#CC</td>
<td>1200</td>
<td>1640</td>
<td>2250</td>
</tr>
<tr>
<td>R-splitting conversion</td>
<td>#CC</td>
<td>14400</td>
<td>21060</td>
<td>27660</td>
</tr>
</tbody>
</table>

Due to the relative cost of the recoding in the multiplicative splitting with R prime, and to the fact that the recoding itself is not regular as implemented, we provide timings without the recoding, considering that in ECDSA, one can directly generate a random scalar k in a multiplicative splitting form and in such case the recoding itself is not regular as implemented. We provide in Table 13 the results of these tests. One sees that the impact of the recoding is at most 8% of the scalar multiplication computation time without recoding, in the worst cases. The most important part of the recoding time is the computation of the multiplicative splittings of the scalar digits, with the repeated use of the Truncated EEA.

6 Conclusion

In this paper, we considered fixed base modular exponentiation and elliptic curve scalar multiplication. These operations are intensively used in NIST standards for digital signature algorithm. In particular, they are used for remote authentication of web server. We
proposed algorithms for modular exponentiation and scalar multiplication based on a multiplicative splitting recoding of the digits of the exponent or scalar. The multiplicative splitting provides more freedom in the distribution of computational cost into storage and online computation. We evaluated the complexities of the proposed approaches for the security levels recommended by the NIST. We could show that, for a fixed level of computation, the proposed approaches reduce the amount of stored data in most of the considered practical cases. Specifically, the storage requirement is reduced by at least 50% in most cases and up to 3 times for the largest NIST standardized key sizes. These complexity results were confirmed by the implementation tests done on an Intel XEON E5-2650.

References

1. The GNU Multiple Precision Arithmetic Library (GMP). http://gmplib.org/

A NIST recommended elliptic curves

We review here the NIST recommended curves (see [12]) used in our implementations. Over a finite field $\mathbb{F}_p$, one has:

- Equation of Weierstrass curve:
  $$ y^2 = x^3 + ax + b $$
  with $a = -3$ and $b \in \mathbb{F}_p$.

  The NIST curves used:

  - **P256**:
    One has $p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$, that is
    $$ p = \underbrace{0xffffffff} \underbrace{0xffffffff} \underbrace{0xffffffff} \underbrace{0xffffffff} $$
    and
    $$ b = \underbrace{0x5ac635d8} \underbrace{aa3a93e7} \underbrace{b3ebbd55} \underbrace{769866b} \underbrace{651d06b} \underbrace{cc53b0f6} \underbrace{3bce3c3e} \underbrace{27a26d0b} $$
  
  Curve subgroup generator:
  $$ \text{G} = \underbrace{0x6b17d1f2} \underbrace{e12c4247} \underbrace{f8be6e65} \underbrace{63a440f2} \underbrace{77037d81} \underbrace{2de3b3a0} \underbrace{f4a13945} \underbrace{d989c296} \underbrace{70e5b315ec} \underbrace{cbb64068} \underbrace{37bf51f5} $$

  - **P384**:
    One has $p = 2^{384} - 2^{128} + 2^{96} + 2^{32} - 1$, that is
    $$ p = \underbrace{0xffffffff} \underbrace{ffffffff} \underbrace{ffffffff} \underbrace{ffffffff} $$
    and
    $$ b = \underbrace{0xb3312fa7} \underbrace{e23ee7e4} \underbrace{988e056b} \underbrace{e3f82d19} \underbrace{181d9c6e} \underbrace{fe814112} \underbrace{0314088f} \underbrace{5013875a} \underbrace{c656398d} \underbrace{8a2ed19d} \underbrace{2a85c8ed} \underbrace{d3e2caef} $$
  
  Curve subgroup generator:
  $$ \text{G} = \underbrace{0xaa87ca22} \underbrace{be8b0537} \underbrace{8e1b7e7f} \underbrace{f320ad74} \underbrace{6e1d3b62} \underbrace{8ba79b98} \underbrace{59f741e0} \underbrace{82542a38} \underbrace{5502f25d} \underbrace{ff55296c} \underbrace{289a147c} \underbrace{e9da3113} \underbrace{b5f0b8c0} \underbrace{0a60b1ce} \underbrace{1d7e819d} \underbrace{7a431d7c} \underbrace{90ea0e5f} $$

  - **P521**:
    One has $p = 2^{521} - 1$, that is
    $$ p = \underbrace{0x01ff} \underbrace{ffffffff} \underbrace{ffffffff} \underbrace{ffffffff} $$
    and
    $$ b = \underbrace{0x0051} \underbrace{953eb961} \underbrace{8e1c9a1f} \underbrace{929a21a0} \underbrace{b68540ee} $$
B Proof of Lemma \[1\]

Before proceeding to the proof of Lemma \[1\] we need to recall the following lemma which states some classical properties of the Extended Euclidean Algorithm. A proof of this lemma can be found in [11].

**Lemma 2** Let \(v_i\) and \(r_i\) be the two sequences of coefficients computed in Algorithm \[4\]. They satisfy the following properties:

\[
\begin{align*}
\text{i)} & \quad (-1)^{i-1} v_i \geq 1 \text{ for all } i \geq 1. \\
\text{ii)} & \quad v_{i+1} r_i - r_{i+1} = (-1)^i R \text{ for all } i \geq 1.
\end{align*}
\]

The proof is a direct consequence of Lemma 2 statements i) and ii) imply that for \(i \geq 1\)

\[
r_{i-1} | v_i | + r_i | v_{i-1} | = R.
\]

So if \(r_{i-1} < c\) is the last remainder such that \(r_{i-1} \geq c\) then we have \(r_i < c\). Then taking \(i = i_c\), in \[13\] leads to \(r_{i-1} | v_{i_c} | + r_{i_c} | v_{i-1} | = R\) then one gets \(|v_{i_c}| \leq R |r_{i-1}| \leq R/c\).

\[C\] Fixed-base scalar multiplication based on a multiplicative splitting recoding with prime \(R\)

We consider an elliptic curve \(E(\mathbb{F}_p)\) a point \(P\) on the curve and a scalar \(k\). The scalar multiplication based on a multiplicative splitting recoding with prime \(R\) is shown in Algorithm 10. Table 14 gives the complexity evaluation.

**Algorithm 10** Fixed-base scalar multiplication based on multiplicative splitting recoding with prime radix \(R\)

**Require:** A prime integer \(R\) a scalar \(k = \sum_{i=0}^{\ell-1} k_i R^i\) with \(\{(s_i, k_i[0], k_i[1])\}, 0 \leq i < \ell, (k_i[0])\) its multiplicative splitting recoding using \(W\)-bit split \(c\) and a fixed point \(P \in E(\mathbb{F}_p)\).

**Ensure:** \(X = k \cdot P\)

1: Precomputation.

   Store \(T[i][j] \leftarrow (\langle j^{-1} \rangle R, R') \cdot P\) for \(i = 0, \ldots, \ell - 1, j = 1, \ldots, \lfloor R/c \rfloor\) and \(T[i][1] \leftarrow R' \cdot P\) and \(T[i][0] \leftarrow \mathcal{O}\) for \(i = 0, \ldots, \ell - 1, j = 1, \ldots, \lfloor R/c \rfloor\).

2: \(X \leftarrow \mathcal{O}, Y_j \leftarrow \mathcal{O}\) for \(1 \leq j \leq c\)

3: for \(i\) from 0 to \(\ell - 1\) do

4: \(Y_{k[i][0]} \leftarrow Y_{k[i][0]} + (s_i) \cdot T[i][k[i][1]]\)

5: \(Y_{k[i][1]} \leftarrow Y_{k[i][1]} + (\text{sign}(k_i')) \cdot T[i][1]\)

6: for \(i\) from \(W\) down to 0 do

7: \(X \leftarrow 2 \times X\)

8: for \(j\) from \(c - 1\) down to 0 do

9: if bit \(i\) of \(j\) is non zero then

10: \(X \leftarrow X + Y_j\)

11: return \((X)\)

**Table 14** Complexity evaluation of scalar multiplication based on multiplicative splitting recoding with \(R\) prime

<table>
<thead>
<tr>
<th>Step</th>
<th>Operation</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\ell\times\text{Step 3})</td>
<td>(Y_{k[i][0]} + s_i \cdot T[i][k[i][1]])</td>
<td>(1\text{MixedAdd})</td>
</tr>
<tr>
<td>(1\times\text{Step 5})</td>
<td>(Y_{k[i][1]} + \text{sign}(k_i') \cdot T[i][1])</td>
<td>(1\text{MixedAdd})</td>
</tr>
<tr>
<td>((W - 1)\times\text{Step 7})</td>
<td>(X \leftarrow 2 \times X)</td>
<td>(1\text{Doubling})</td>
</tr>
<tr>
<td>((\ell - 1)\times\text{Step 10})</td>
<td>(X \leftarrow X + Y_j)</td>
<td>(1\text{Addition})</td>
</tr>
<tr>
<td>TOTAL</td>
<td>((\ell + 1) \times \text{MixedAdd})</td>
<td>((W - 1) \times \text{Doubling} + (\ell - 1) \times \text{Addition})</td>
</tr>
<tr>
<td>TOTAL STORAGE</td>
<td>(\ell \times (\lceil R/c \rceil + 1) + 1) points on (E(\mathbb{F}_p))</td>
<td></td>
</tr>
</tbody>
</table>