Efficient Fixed Base Exponentiation and Scalar Multiplication based on a Multiplicative Splitting Exponent Recoding

Jean-Marc Robert, Christophe Negre, Thomas Plantard

Efficient Fixed Base Exponentiation and Scalar Multiplication based on a Multiplicative Splitting Exponent Recoding

Jean-Marc Robert[2,3], Christophe Negre[2,3] and Thomas Plantard[1]

[1] CCISR, SCIT, University of Wollongong, Australia

[2] Team DALI, Université de Perpignan Via Domitia, France

[3] LIRMM, UMR 5506, Université de Montpellier and CNRS, France

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.

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

I. INTRODUCTION

In the DSS (Digital Signature Standard), DSA (Digital Signature Algorithm) is a popular authentication protocol. According to the NIST standard (see [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 $g^k \bmod p$ (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 $(E(\mathbb{F}_p), +)$ 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 $P \in E(\mathbb{F}_p)$. 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 [8] and the *Fixed-base Comb* presented by Lim and Lee in [14]. The *Fixed-base Radix R* method of [8] precomputes g^{aR^i} 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 $k = \sum_{i=0}^{\ell-1} k_i R^i$ 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 II, we review the best approaches of the literature for fixed-base exponentiation and we give their complexities and storage requirements. In Section III, 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 IV, we present a multiplicative splitting recoding for R prime and the corresponding exponentiation algorithm. In Section V, 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 VI, we give some concluding remarks and perspectives.

II. 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 $E(\mathbb{F}_p)$ [16], [13]. 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

$$s_1 \leftarrow H_1(g^k),$$
$$s_2 \leftarrow (H_2(m) + s_1 x)k^{-1} \mod q.$$

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 $\cong q$. 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 $t - 1$ 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.

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

---

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

Require: Let an integer $k = (k_{t-1}, \ldots, k_0)_2$, and g an element of G.

Ensure: $X = g^k$

1: $X \leftarrow 1$

2: **for** i **from** $t-1$ **downto** 0 **do**

3:     $X \leftarrow X^2$

4:     **if** $k_i = 1$ **then**

5:        $X \leftarrow X \cdot g$

6: **return** (X)

---

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^i}), then for a given computation of g^k, one has to multiply the g^{2^i} 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 $k = \sum_{i=0}^{\ell-1} k_i R^i$. This consists in the memorization of the values $g^{a \cdot R^j}$, with $a \in [0, ..., R-1]$ and $0 \le j < \ell$ where ℓ is the length of the exponent in radix R representation. If we denote $w = \lceil \log_2(R) \rceil$ then we have $\ell = \lceil t/w \rceil$. In this case, the online computation consists of $\ell - 1$ multiplications, for a storage amount of $\ell \cdot R$ 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 $k_i = 0$.

Comb method. Another classical method is the so called *Fixed-base Comb* method which was initially proposed by Lim and Lee in [14]. 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

$$k = \sum_{j=0}^{d-1} (\underbrace{\sum_{i=0}^{w-1} k_{id+j} 2^{id}}_{K_j}) \; 2^j \text{ where } d = \lceil t/w \rceil. \tag{1}$$

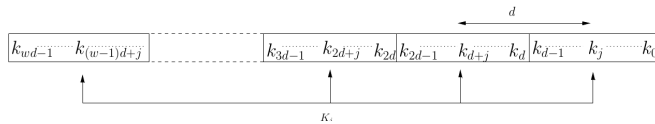Each integer K_j can be seen as a comb as described in the following diagram.

Algorithm 2 Fixed-Base Radix-R Exponentiation

Require: $k = (k_{\ell-1}, \ldots, k_0)_R$, g a generator of G.

Ensure: $X = g^k$

1: *Offline precomputation.* Store $T[a][j] \leftarrow g^{a \cdot R^j}$, with $a \in [0, ..., R-1]$ and $0 \le j < \ell$.

2: $X \leftarrow 1$

3: **for** i **from** $\ell - 1$ **downto** 0 **do**

4: $\quad X \leftarrow X \cdot T[k_i][i]$

5: **return** (X)

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

$$I_{K_j} = [k_{(w-1)d+j} k_{(w-2)d+j} \ldots, k_j]_2,$$

one can compute g^k with a 2^w size look-up table, $\lceil t/w \rceil - 1$ multiplications and $\lceil t/w \rceil - 1$ 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 $\ne 0$.

Algorithm 3 Fixed-Base Comb Exponentiation [14]

Require: $k = (k_{t-1}, \ldots, k_1, k_0)_2$, a generator g of G, a window width 2^w and $d = \lceil t/w \rceil$.

Ensure: $X = g^k \bmod p$

1: *Offline precomputation.* For all $(a_{w-1}, \ldots, a_0) \in \{0,1\}^w$ we set $a = a_{w-1} 2^{(w-1)d} + \cdots + a_1 2^d + a_0$ and $T[(a_{w-1}, \ldots, a_0)_2] = g^a$.

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: $\quad X \leftarrow X^2$

6: $\quad X \leftarrow X \cdot T[K_j]$

7: **return** (X)

Fixed base exponentiation with NAF_w. 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 a elliptic curves where computing the inverse of a point is really cheap. They

use a NAF_w 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_w representation of the exponent k

$$k = k'_{t-1}2^{t-1} + k'_{t-2}2^{t-2} + \cdots + k'_0$$

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

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

In [17] the authors noticed that, in each K_i, there is at most one non-zero coefficient k'_{iw+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 $0 \le j < w$. They then reorder the terms in expression (2) by splitting the parameter i into two parts $i = i_1e + i_0$ for some fixed integer e:

$$\begin{aligned} k &= \sum_{i_0=0}^{e-1} \sum_{i_1=0}^{d-1} K_{i_1e+i_0} 2^{i_1ew+i_0w} \text{ where } d = \lceil \ell/e \rceil \\ &= \sum_{i_0=0}^{e-1} \left(\sum_{i_1=0}^{d-1} K_{i_1e+i_0} 2^{i_1ew} \right) 2^{i_0w}. \end{aligned} \tag{3}$$

For all possible values for $K_{i_1e+i_0} 2^{i_1ew}$ with $K_{i_1e+i_0} = sa2^j$ the term $g^{a2^{j+i_1ew}}$ 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_w [17]

Require: A scalar $k = (k'_{t-1}, \ldots, k'_1, k'_0)_{NAF_w}$ 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.* $T[a][i_1][j] = g^{a2^{j+ewi_1}}$ 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** $e - 1$ **downto** 0 **do**

5:     $X \leftarrow X^{2^w}$

6:     **for** i_1 **from** $d - 1$ **downto** 0 **do**

7:         (s, a, j) s.t. $(k'_{jb+t,w-1} \ldots k'_{jb+t,0})_{NAF_w} = s \cdot a \cdot 2^j$

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 $dw2^{w-2} \cong \lceil \frac{t}{e} \rceil 2^{w-2}$. The online computation consists of $w(e-1)$ squarings and $ed(1 - (\frac{w}{w+1})^w) \cong \lceil \frac{t}{w} \rceil (1 - (\frac{w}{w+1})^w)$ multiplications/divisions (cf. [17] for details).

III. FIXED-BASE EXPONENTIATION WITH MULTIPLICATIVE SPLITTING WITH $R = m_0 m_1$

We now present our approach of a Fixed-base exponentiation with multiplicative splitting with $R = m_0 m_1$. In this section, we review the method presented in a preliminary work at WAIFI 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.

A. Digit multiplicative splitting for radix $R = m_0 m_1$

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 $\mathcal{B} = \{m_0, m_1\}$. Let $R = m_0 \cdot m_1$ and $x \in \mathbb{Z}$ such that $0 \le x < R$. 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 $|x|_m = x \bmod m$.

One represents x with the residues

$$\begin{cases} x^{(0)} = |x|_{m_0}, \\ x^{(1)} = |x|_{m_1}, \end{cases}$$

and x can be retrieved using the Chinese Remainder Theorem as follows:

$$x = \left| x^{(0)} \cdot m_1 \cdot |m_1^{-1}|_{m_0} + x^{(1)} \cdot m_0 \cdot |m_0^{-1}|_{m_1} \right|_R. \tag{4}$$

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

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

We represent every radix-R digit in RNS with the RNS base $\mathcal{B} = \{m_0, m_1\}$: 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 $\mathcal{B}$

$$\begin{cases} k_i^{(0)} = |k_i|_{m_0}, \\ k_i^{(1)} = |k_i|_{m_1}. \end{cases}$$

Let us denote

$$m_0' = m_1 \cdot |m_1^{-1}|_{m_0},$$
$$m_1' = m_0 \cdot |m_0^{-1}|_{m_1}.$$

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

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

$$
\begin{cases}
k_i'^{(0)} & = & |k_i^{(0)} \cdot (k_i^{(1)})^{-1}|_{m_0}, \\
k_i'^{(1)} & = & k_i^{(1)}.
\end{cases}
$$

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 $k_i = |k_i'|_R$ with (4). When modifying the digits of k as above, one needs to take into account the correcting term due to the reduction modulo R:

$$
\begin{aligned}
k_i = k_i'^{(1)} |k_i'^{(0)} \cdot m_0' + m_1'|_R \\
- \lfloor k_i'^{(1)} \cdot |k_i'^{(0)} \cdot m_0' + m_1'|_R / R \rfloor \cdot R.
\end{aligned}
$$

Let us denote $C = \lfloor k_i'^{(1)} \cdot (k_i'^{(0)} \cdot m_0' + m_1')/R \rfloor$ 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**

      $k_{i+1} \leftarrow k_{i+1} - C$

      $C \leftarrow 0$

   else

      $k_{i+1} \leftarrow k_{i+1} + R - C,$

      $C \leftarrow 1$

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

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

$$
k_i' = \underbrace{\left| |k_i^{(0)} + 1|_{m_0} \cdot m_0' + m_1' \right|_R}_{(*)} - \underbrace{|m_0' + m_1'|_R}_{=1}. \tag{6}
$$

and k_i' satisfies $|k_i'|_R = k_i$ This expression is meant to have the part $(*)$ as in (5): 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:

$$
\begin{cases}
k_i'^{(0)} & = & |k_i^{(0)} + 1|_{m_0}, \\
k_i'^{(1)} & = & 0.
\end{cases}
$$

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

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

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 $k = \sum_{i=0}^{\ell-1} k_i R^i$.

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

1: $C \leftarrow 0$

2: **for** i **from** 0 **to** $\ell - 1$ **do**

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:      $k_i^{(0)} \leftarrow |k_i|_{m_0}, k_i^{(1)} \leftarrow |k_i|_{m_1}$.

7:      **if** $k_i^{(1)} = 0$ **then**

8:          $(k_i'^{(0)}, k_i'^{(1)}) \leftarrow (|k_i^{(0)} + 1|_{m_0}, 0)$

9:          $C \leftarrow C + \left\lfloor \left(|k_i'^{(0)} \cdot m_0' + m_1'|_R - 1 \right) / R \right\rfloor$

10:     **else**

11:          $k_i'^{(0)} \leftarrow |k_i^{(0)} \cdot (k_i^{(1)})^{-1}|_{m_0}$

12:          $k_i'^{(1)} \leftarrow k_i^{(1)}$

13:          $C \leftarrow C + \lfloor k_i'^{(1)} \cdot |k_i'^{(0)} \cdot m_0' + m_1'|_R / R \rfloor$

14: **return** $\{(k_i'^{(0)}, k_i'^{(1)}), 0 \le i < \ell, k_\ell' = -C\}$

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 $k_i'^{(1)} \times |k_i'^{(0)} \cdot m_0' + m_1'|_R$ 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 $m_0 m_1$ recoding with an exponent size t of 20 bits ($0 < k < 2^{20}$), and $\mathcal{B} = \{11, 8\}$ (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*

$$m_0' = 8 \cdot |8^{-1}|_{11} = 56,$$
$$m_1' = 11 \cdot |11^{-1}|_8 = 33.$$

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 5, 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`*-test steps 4 to 5 since $k_0 \ge 0$.*

  - *Step 6, one computes the RNS representation in base $\mathcal{B}$ of $k_0 = 48$:*

    $$k_0^{(0)} = |k_0|_{11} = 4, k_0^{(1)} = |k_0|_8 = 0.$$

  - *Steps 7 to 9, since $k_0^{(1)} = 0$, one sets*

    $$(k_0'^{(0)}, k_0'^{(1)}) \leftarrow (|k_0^{(0)} + 1|_{11}, 0) = (5, 0).$$

and the carry

$$C \leftarrow C + \left\lfloor \left(\left| k_0'^{(0)} \cdot 56 + 33 \right|_{88} - 1 \right) / 88 \right\rfloor = 0$$

- *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 $\mathcal{B}$ of $k_1 = 78$:*

  $$k_1^{(0)} = |k_1|_{11} = 1, k_1^{(1)} = |k_1|_8 = 6.$$

  - *Steps 10 to 13, since $k_1^{(1)} \neq 0$, one has*

  $$
  \begin{aligned}
  |(k_1^{(1)})^{-1}|_{11} &\quad&\leftarrow\quad& 2 \\
  k_1'^{(0)} = |k_1^{(0)} \cdot (k_1^{(1)})^{-1}|_{11} &\quad&\leftarrow\quad& 2 \\
  k_1'^{(1)} = k_1^{(1)} &\quad&\leftarrow\quad& 6 \\
  C \leftarrow \lfloor (k_1'^{(1)} \cdot |k_1'^{(0)} \cdot 56 + 33|_{88})/88 \rfloor &\quad&\leftarrow\quad& 3
  \end{aligned}
  $$

- *In the third iteration ($i = 2$), one has now $k_2 \leftarrow k_2 - C = 29$.*

  - *The RNS representation in base $\mathcal{B}$ 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:

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

B. Exponentiation with a multiplicative splitting recoding in radix $R = m_0 m_1$

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

$$
\begin{aligned}
g^k \bmod p &= g^{\sum_{i=0}^{\ell} k_i' \cdot R^i} \\
&= g^{k_\ell' \cdot R^\ell} \cdot \prod_{i=0}^{\ell-1} g^{k_i' \cdot R^i}
\end{aligned}
\tag{7}
$$

where each term $g^{k_i' \cdot R^i}$ satisfy one of the following three cases:

- When $k_i'^{(1)} \neq 0$ and $i < \ell$:

$$g^{k_i' \cdot R^i} = g^{k_i'^{(1)} \cdot R^i \cdot |k_i'^{(0)} \cdot m_0' + m_1'|_R}$$

- When $k_i'^{(1)} = 0$ and $i < \ell$:

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

- 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:

$$T[i][j] = g^{R^i \cdot |j \cdot m_0' + m_1'|_R}, \text{ with } \begin{cases} 0 \leq i \leq \ell - 1, \\ 0 \leq j < m_0. \end{cases}$$

and one also stores the following inverses:

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

We use Y_j to denote the product of $g^{R^i \cdot |k_i'^{(0)} \cdot m_0' + m_1'|_R}$ for each i such that $k_i'^{(1)} = j$. In other words for $j \ne 0$

$$Y_j = \begin{cases} \left(\prod_{\text{for } k_i'^{(1)} = j, i < \ell} T[i][k_i'^{(0)}] \right) \cdot T[\ell][-1] \text{ if } |k_\ell'| = j, \\ \left(\prod_{\text{for } k_i'^{(1)} = j, i < \ell} T[i][k_i'^{(0)}] \right), \end{cases}$$

and

$$Y_0 = \prod_{\text{for all } k_i'^{(1)} = 0, i < \ell} T[i][k_i'^{(0)}] \times T[i][-1].$$

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 6.

One important drawback of the above algorithm is that it is not constant time, due to the **if** branching attached to the condition $k_i'^{(1)} = 0$.

Example 2. *We present the computation of $g^k \mod p$ using Algorithm 6, we take $\mathcal{B} = \{11, 8\}$ (i.e. $m_0 = 11, m_1 = 8$). In terms of storage, one computes the values*

$$T[i][j] = g^{R^i \cdot |j \cdot m_0' + m_1'|_R} \mod p \text{ with } 0 \le i \le \ell - 1.$$

One has the values $\{33, 1, 57, 25, 81, 49, 17, 73, 41, 9, 65\}$ for $|j \cdot m_0' + m_1'|_R$ when $0 \le j < 11$. This leads to

$$T[i][0..10] = \{g^{88^i \cdot 33}, g^{88^i}, g^{88^i \cdot 57}, g^{88^i \cdot 25}, g^{88^i \cdot 81},$$
$$g^{88^i \cdot 49}, g^{88^i \cdot 17}, g^{88^i \cdot 73}, g^{88^i \cdot 41}, g^{88^i \cdot 9}, g^{88^i \cdot 65}\}.$$

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 I.

C. 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 III for the average case. The number of multiplications (M) is evaluated as follows:

- The costs of Steps 6 to 15 follow directly from Algorithm 6 and are detailed in Table III.
- 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 $Y_j \ne 1$. This means that in the worst case we save the first multiplication which is an affectation : this is the case considered in Table III.

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

Algorithm 6 Fixed-base exponentiation with multiplicative splitting with radix $R = m_0 m_1$

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

Ensure: $A = g^k$

1: *Offline precomputation.* Store $T[i][j] \leftarrow g^{R^i \cdot |j \cdot m_0' + m_1'|_R}$ with $0 \leq i < \ell, 0 \leq j < m_0, T[i][-1] \leftarrow g^{-R^i}, 0 \leq i \leq \ell$

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** $k_i'^{(1)} = 0$ **then**

5:         **if** $Y_0 = 1$ **then**

6:             $Y_0 \leftarrow T[i][k_i'^{(0)}] \times T[i][-1]$

7:         **else**

8:             $Y_0 \leftarrow Y_0 \times T[i][k_i'^{(0)}] \times T[i][-1]$

9:     **else**

10:         **if** $Y_{k_i'^{(1)}} = 1$ **then**

11:             $Y_{k_i'^{(1)}} \leftarrow T[i][k_i'^{(0)}]$

12:         **else**

13:             $Y_{k_i'^{(1)}} \leftarrow Y_{k_i'^{(1)}} \times T[i][k_i'^{(0)}]$

14: **if** $k_\ell' \neq 0$ **then**

15:     $Y_{|k_\ell'|} \leftarrow Y_{|k_\ell'|} \times T[\ell][-1]$

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** 1 **do**

20:         **if** bit i of j is non zero **then**

21:             $X \leftarrow X \times Y_j$

22: **return** $(X \times Y_0)$

IV. 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.

A. 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

$$k_i = k_i^{(0)}(k_i^{(1)})^{-1} \mod R \text{ with } \begin{cases} |k_i^{(0)}| < c \\ |k_i^{(1)}| \leq R/c \end{cases} \tag{8}$$

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

Table I

EXAMPLE OF AN EXECUTION TRACE FOR AN EXPONENTIATION BASED ON MULTIPLICATIVE SPLITTING RECODING WITH $R = m_0 m_1$

Iter. (loop 3:)	Exp. coef.	Step	Value
$i = 0$	$k_0'^{(0)} = 5$ $k_0'^{(1)} = 0$	6:	$Y_0 \leftarrow \quad T[0][k_0'^{(0)}] \times T[0][-1]$ $= g^{49} \times g^{-1} = g^{48}$
$i = 1$	$k_1'^{(0)} = 2$ $k_1'^{(1)} = 6$	11:	$Y_6 \leftarrow T[1][k_1'^{(0)}] \quad = \quad g^{88 \cdot 57}$ $= \quad g^{5016}$
$i = 2$	$k_2'^{(0)} = 8$ $k_2'^{(1)} = 5$	11:	$Y_5 \leftarrow T[2][k_2'^{(0)}] \quad = \quad g^{88^2 \cdot 41}$ $= \quad g^{317504}$
$i = 3$	$k_3'^{(0)} = 3$ $k_3'^{(1)} = 7$	11:	$Y_7 \leftarrow T[3][k_3'^{(0)}] \quad = \quad g^{88^3 \cdot 25}$ $= \quad g^{17036800}$
-	-	15:	$T_2 \leftarrow T[4][-1] \quad = \quad g^{88^4 \cdot (-1)}$ $= \quad g^{-59969536}$
-	-	17: to 22:	$g^k \quad = \quad Y_0 \times \prod_{j=1}^{m_1 - 1} Y_j^j$ $= \quad g^{48} g^{2 \cdot (-59969536)}$ $\times g^{5 \cdot 317504}$ $\times g^{6 \cdot 5016} g^{7 \cdot 17036800}$ $= \quad g^{936192}$

Table II

HAMMING WEIGHTS ACCOUNT FOR $0 \le j < m_1$

m_1	2	3	4	5	6	7	8	9
$\mathcal{H}$	1	2	4	5	7	9	12	13

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:

$$r_0 = R, r_1 = k, r_2 = r_0 \bmod r_1, \dots$$

$$\dots, r_{j+1} = r_{j-1} \bmod r_j, \dots$$

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 $r_\ell \ne 0$ satisfies $r_\ell = \gcd(k, R) = 1$. The extended Euclidean algorithm computes a Bezout relation

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

TABLE III

COMPLEXITY OF EXPONENTIATION BASED ON MULTIPLICATIVE SPLITTING RECODING WITH $R = m_0 m_1$

Complexity		
Step	Operation	Cost
$1\times$ Step 6	$T[i][k_i'^{(0)}] \times T[i][-1]$	1 M
$(\ell/m_1 - 1)\times$ Step 8	$Y_0 \times T[i][k_i'^{(0)}] \times T[i][-1]$	2 M
$(m_1 - 1)\times$ Step 11	-	-
$(\ell\frac{m_1-1}{m_1} - (m_1 - 1))$ $\times$ Step 13	$Y_{k_i'^{(1)}} \times T[i][k_i'^{(0)}]$	1 M
$1\times$ Step 15	$Y_{|k_\ell'|} \times T[\ell][-1]$	1 M
$(W - 1)\times$ Step 18	$X \leftarrow X^2$	1 S
$(\mathcal{H} - 1)\times$ Step 21	$X \times Y_j$	1 M
$1\times$ Step 22	$(X \times Y_0)$	1 M
TOTAL	$(\ell\frac{m_1+1}{m_1} - m_1 + \mathcal{H} + 1)$ M $+(W - 1)$ S	
TOTAL STORAGE	$(m_0 + 1) \times \ell + 1$ elements of G	

by maintaining two sequences of integers u_j and v_j satisfying:

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

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.

2) Recoding the exponent: We now present our recoding approach for an integer k given in radix-R representation:

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

We choose a splitting bound c and we consider a digit $k_i \neq 0$. Using Algorithm 7 we get $s_i, k_i^{(0)}$ and $k_i^{(1)}$ such that

$$k_i = s_i k_i^{(0)} (k_i^{(1)})^{-1} \mod R \text{ with } \begin{cases} s_i \in \{-1, 1\} \\ k_i^{(0)} \in [0, c[, \\ k_i^{(1)} \in [0, R/c]. \end{cases} \tag{10}$$

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: $(s, k^{(0)}, k^{(1)})$, such as $k = |s \times k^{(0)} \times (k^{(1)})^{-1}|_R$ with $0 \le k^{(0)} < c$ and $0 \le k^{(1)} \le \lceil R/c \rceil$ and $s \in \{-1, 1\}$ when
   $\gcd(k, R) = 1$.

1: **if** $\gcd(k, R) = R$ **then**

2:     **return** $(1, 0, 0)$

3: **else**

4:     $u_0 \leftarrow 1, v_0 \leftarrow 0, r_0 \leftarrow R, u_1 \leftarrow 0, v_1 \leftarrow 1, r_1 \leftarrow |k|_R$

5:     **while** $(r_1 \ge c)$ **do**

6:         $q \leftarrow \lfloor r_0/r_1 \rfloor, r_2 \leftarrow |r_0|_{r_1}$

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:    $s \leftarrow \text{sign}(v_1), k^{(0)} \leftarrow r_1, k^{(1)} \leftarrow |v_1|$

11:    **return** $(s, k^{(0)}, k^{(1)})$

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 $-c \le C < c$. We then consider C as a carry that we subtract to k_{i+1}.

We obtain an expression $k = \sum_{i=0}^{\ell} k_i' R^i$ 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 $k_\ell' = -C$ 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, $k = \sum_{i=0}^{\ell-1} k_i R^i$, and c the splitting bound.

Ensure: $\{(s_i, k_i^{(0)}, k_i^{(1)}), 0 \le i < \ell, (k_\ell')\}$ 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:     $C \leftarrow (s_i k_i^{(0)} |(k_i^{(1)})^{-1}|_R - k_i)/R$ //exact quotient

6: **return** $\{(s_i, k_i^{(0)}, k_i^{(1)}), 0 \le i < \ell, (k_\ell' = -C)\}$

Example 3. *We present an example of multiplicative splitting recoding for a prime radix $R = 89$ with an exponent
size t of 20 bits ($0 < k < 2^{20}$). In this case, one has $\ell = \lceil 20/\log_2(89) \rceil = 4$. One also sets $c = 2^3 = 8$, and then,
$\lceil R/c \rceil = 12$. Let us take $k = 901644_{10}$, 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 IV.

Table IV

EXAMPLE OF AN EXECUTION TRACE OF ALGORITHM 8

Iter.	Step	Value
	3:	$k_0 = 74$ does not change since $C = 0$
$i = 0$	4:	$s_0 = -1, k_0^{(0)} = 1, k_0^{(1)} = 6.$
	5:	$C \leftarrow (s_0 \cdot k_0^{(0)} \cdot |(k_0^{(1)})^{-1}|_R - k_0)/R = -1$
	3:	$k_1 \leftarrow 73 + 1 = 74$ since $C = -1$
$i = 1$	4:	$s_1 = -1, k_1^{(0)} = 1, k_1^{(1)} = 6.$
	5:	$C \leftarrow (s_1 \cdot k_1^{(0)} \cdot |(k_1^{(1)})^{-1}|_R - k_1)/R = -1$
	3:	$k_2 \leftarrow 24 + 1 = 25$ since $C = -1$
$i = 2$	4:	$s_2 = -1, k_2^{(0)} = 3, k_2^{(1)} = 7.$
	5:	$C \leftarrow (s_2 \cdot k_2^{(0)} \cdot |(k_2^{(1)})^{-1}|_R - k_2)/R = -2$
	3:	$k_3 \leftarrow 1 + 2 = 3$ since $C = -2$
$i = 3$	4:	$s_3 = 1, k_3^{(0)} = 3, k_3^{(1)} = 1.$
	5:	$C \leftarrow (s_3 \cdot k_3^{(0)} \cdot |(k_3^{(1)})^{-1}|_R - k_3)/R = 0$
$((-1,1,6),(-1,1,6),(-1,3,7),(1,3,1))$ and $k_4' = C = 0$		

B. Exponentiation Algorithm with multiplicative splitting recoding in a prime radix R

We now present an exponentiation algorithm which takes advantage of the exponent recoding given in Section IV-A2. One wants to compute

$$
\begin{aligned}
g^k &= g^{\sum_{i=0}^{\ell} k_i' \cdot R^i} \\
&= g^{k_\ell' \cdot R^\ell} \cdot \prod_{i=0}^{\ell-1} g^{k_i' \cdot R^i}
\end{aligned}
\tag{11}
$$

with

$$
\begin{aligned}
g^{k_i' \cdot R^i} &= g^{s_i \cdot k_i^{(0)} \cdot |(k_i^{(1)})^{-1}|_R \cdot R^i}, \ \text{if } k_i^{(1)} \neq 0, \\
g^{k_i' \cdot R^i} &= 1, \ \text{if } k_i^{(1)} = 0 (\text{this corresponds to } k_i = 0).
\end{aligned}
$$

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

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

$$
\begin{aligned}
T[i][s][0] &= 1 \text{ with } s \in \{-1, 1\}. \\
T[\ell][s] &= g^{sR^\ell} \text{ with } s \in \{-1, 1\}.
\end{aligned}
$$

One denotes Y_j the product of the terms $g^{s_i \cdot |(k_i^{(1)})^{-1}|_R \cdot R^i}$ such that of $k_i^{(0)} = j$. This means that for $j \neq |k'_\ell|$

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

and for $j = |k'_\ell|$ one has

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

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

$$g^k = \prod_{j \in \{1,\dots,c-1\}} Y_j^j.$$

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.

Algorithm 9 Fixed-base exponentiation with multiplicative splitting for prime radix R

Require: R a prime integer, an exponent $k = \sum_{i=0}^{\ell-1} k_i R^i$ and $\{(s_i, k_i^{(0)}, k_i^{(1)}), 0 \leq i < \ell, k'_\ell\}$ the multiplicative splitting recoding in radix R of k and $g \in G$.

Ensure: $X = g^k$

1: *Offline precomputation.* For $0 \leq i \leq \ell-1, 1 \leq j \leq \lceil R/c \rceil, s \in \{-1, 1\}$ store $T[i][s][j] \leftarrow g^{R^i \cdot s \cdot |j^{-1}|_R}$ and $T[i][s][0] \leftarrow 1$ for $0 \leq i \leq \ell-1, s \in \{-1, 1\}$ and $T[\ell][s] \leftarrow g^{sR^\ell}$ for $s \in \{-1, 1\}$.

2: $X \leftarrow 1, Y_j \leftarrow 1$ for $0 \leq j \leq c$

3: **for** i from 0 to $\ell-1$ **do**

4: $\quad Y_{k_i^{(0)}} \leftarrow Y_{k_i^{(0)}} \times T[i][s_i][k_i^{(1)}]$

5: $Y_{|k'_\ell|} \leftarrow Y_{|k'_\ell|} \times T[\ell][\text{sign}(k'_\ell)]$

6: $W \leftarrow$ size of c in bits

7: **for** i from $W-1$ **downto** 0 **do**

8: $\quad X \leftarrow X^2$

9: $\quad$ **for** j from $c-1$ **downto** 1 **do**

10: $\quad\quad$ **if** bit i of j is non zero **then**

11: $\quad\quad\quad X \leftarrow X \times Y_j$

12: **return** (X)

The above algorithm can be implemented in a constant time fashion. Indeed there is no **if** 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.

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

$$((-1, 1, 6), (-1, 1, 6), (-1, 3, 7), (1, 3, 1)) \text{ and } k'_4 = 0. \tag{12}$$

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

$$T[i][s][j] = g^{R^i \cdot s \cdot |j^{-1}|_R} \ with \ \begin{cases} 0 \le i \le \ell - 1, \\ 1 \le j \le \lceil R/c \rceil = 12, \\ s \in \{-1, 1\}. \end{cases}$$

One has the following values of $\left|j^{-1}\right|_R$ for $1 \le j \le 12$

$$\{1, 45, 30, 67, 18, 15, 51, 78, 10, 9, 81, 52\}.$$

This brings us to store the following values in G:

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

The execution of Algorithm 9 is shown step by step in Table V

Table V

EXAMPLE OF AN EXECUTION TRACE FOR AN EXPONENTIATION BASED ON MULTIPLICATIVE SPLITTING RECODING WITH R PRIME

Iter.	Step	Coeff			Value
$i = 0$	4:	s_0	$=$	-1	$Y_1 \leftarrow Y_1 \times T[0][s_0][k_0^{(1)}]$
		$k_0^{(0)}$	$=$	1	$= 1 \times g^{-15}$
		$k_0^{(0)}$	$=$	6	
$i = 1$	4:	s_1	$=$	-1	$Y_1 \leftarrow Y_1 \times T[1][s_1][k_1^{(1)}]$
		$k_1^{(0)}$	$=$	1	$= g^{-15} \times g^{-89 \cdot 15}$
		$k_1^{(1)}$	$=$	6	$= g^{-1350}$
$i = 2$	4:	s_2	$=$	-1	$Y_3 \leftarrow Y_3 \times T[2][s_3][k_2^{(1)}]$
		$k_2^{(0)}$	$=$	3	$= 1 \times g^{-89^2 \cdot 51}$
		$k_2^{(1)}$	$=$	7	$= g^{-403971}$
$i = 3$	4:	s_3	$=$	1	$Y_3 \leftarrow Y_3 \times T[3][s_3][k_3^{(1)}]$
		$k_3^{(0)}$	$=$	3	$= g^{-403971} \times g^{89^3 \cdot 1}$
		$k_3^{(1)}$	$=$	1	$= g^{300998}$
-	5:	$k_4' = 0$			$Y_0 \leftarrow Y_0 \times T[\ell][\text{sign}(k_4')] = g^{59969536}$
	7:				$g^k = \prod_{j=1}^{c-1} Y_j^j$
-	to	-			$= g^{3 \cdot 300998 - 1350}$
	11:				$= g^{901644}$

C. 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 $Y_{k_i^{(0)}} = 1$, 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 affectation. The other multiplications in Step 11 are performed even in case of $Y_j = 1$, again to ensure a constant computation time.

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

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

Table VI

EXPONENTIATION COMPLEXITY AND STORAGE FOR THE PROPOSED APPROACH WITH A PRIME RADIX R RECODING.

Complexity		
Step	Operation	Complexity
$\ell \times$ Step 4	$Y_{k_i^{(0)}} \times T[i][s_i][k_i^{(1)}]$	1 M
$1 \times$ Step 5	$Y_{|k_\ell'|} \times T[\ell][\text{sign}(k_\ell')]$	1 M
$(W - 1) \times$ Step 12	X^2	1 S
$(\mathcal{H} - 1) \times$ Step 15	$X \times Y_j$	1 M
TOTAL	$(\ell + \mathcal{H})$ M $+(W - 1)$ S	
TOTAL STORAGE	$2(\lceil R/c \rceil + 1)\ell + 2$ elements of G	

V. COMPLEXITY AND EXPERIMENTATION COMPARISON

A. Complexity comparison

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

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^{2w_R} while the Radix-R method needs $\frac{t}{w_R}2^{w_R}$ elements of G. In other words, for a fixed number

Table VII

COMPLEXITIES AND STORAGE AMOUNTS OF EXPONENTIATION ALGORITHM, AVERAGE CASE, BINARY EXPONENT LENGTH t.

	Constant time	#Mul	#Squ.	Storage (#values in G)
Square-and-mult. (Algo. 1)	no	$\frac{t}{2}$	$t-1$	0
Fixed-base Radix-$R^{(*)}$ (Algo. 2)	yes	$\frac{t}{w}-1$	0	$\frac{t}{w}2^w$
Fixed-base Comb (Algo. 3)	yes	$\frac{t}{w}-1$	$\frac{t}{w}-1$	2^w
Fixed base NAF$_w$ (Algo. 3)	no	$\frac{t}{w}\left(1-\left(\frac{w}{w+1}\right)^w\right)$	$(e-1)w$	$\frac{t}{e}2^{w-2}$
Proposed$^{(*)}$ with $R=m_0 m_1$ (Algo. 6)	no	$\frac{t}{w}\frac{m_1+1}{m_1}-m_1+\mathcal{H}+1$	$W-1$	$(2^w/m_1+1)\frac{t}{w}+1$
Proposed$^{(*)}$ with R prime (Algo. 9)	yes	$\frac{t}{w}+\mathcal{H}$	$W-1$	$(2^{w+1}/c+1)\frac{t}{w}+1$

$^{(*)}$ We assume that R is a w bit integer

t/w_R of online computation, the Comb method is better than the Radix-R as soon as $2^{w_R} < \frac{t}{w_R}$ 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 ($\mathcal{H}$ and W). This means that the proposed approaches can be competive 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.

B. Complexities and timings for modular exponentiation

In this subsection we focus on exponentiation in $((\mathbb{Z}/p\mathbb{Z})^*, \times)$ 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 recoding of the exponent (Algorithm 6 for $R = m_0 m_1$ 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 MS = 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:

Table VIII

NIST RECOMMENDED KEY AND FIELD SIZES

Security level	80	112	128	192	256
Key size (bits)	160	224	256	384	512
Field size (bits)	1024	2048	3072	7680	15360

Fig. 1 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 [4]) and the behavior is roughly the same for all sizes, although the benefit of our approach with $R = m_0 m_1$ is lower for smaller sizes. One can see that the Fixed-base Comb method is the best for small storage amount. Our $m_0 m_1$ 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 9) is less efficient than the $R = m_0 m_1$ for small storage amounts. The reason is that this requires some additional computations to get a constant time execution, while the $m_0 m_1$ 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 IX shows numerical application of the complexity comparison between the Fixed-base Comb (Algorithm 3), the Fixed-base Radix-R (Algorithm 2) and the approaches based on our multiplicative splitting recodings (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: $R = m_0 m_1$ 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

Figure 1. Complexity comparison, Fixed base modular exponentiation NIST DSA, key size 256, 384 and 512 bits (field size 3072, 7360 and 15360 bits).
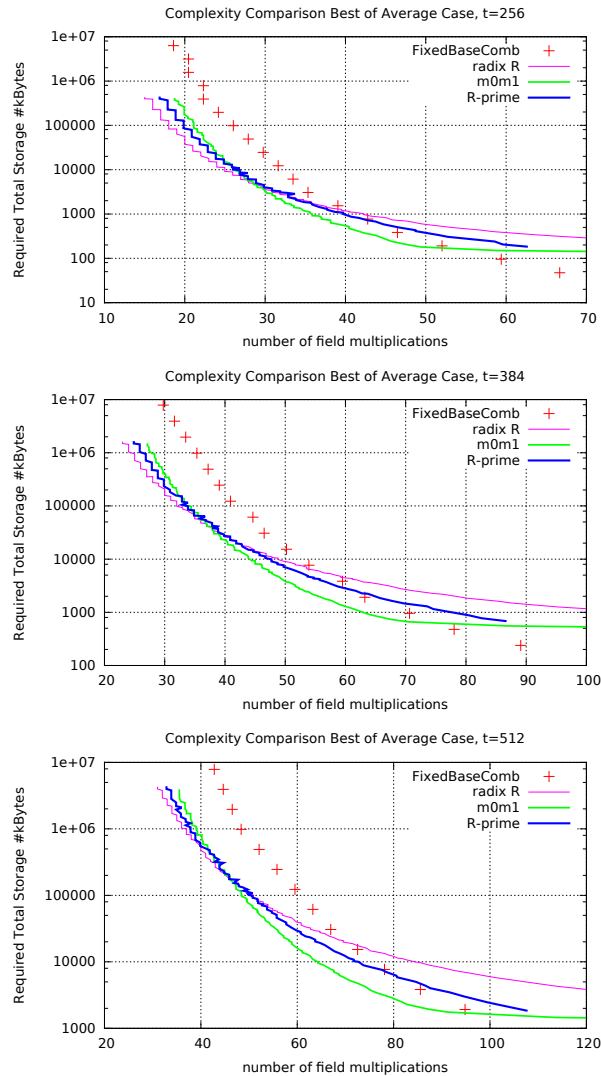
WAIFI 2016. The results in Table IX show that the exponentiation with multiplicative splitting with $R = m_0 m_1$ and R prime are close from each other. But the approach with $R = m_0 m_1$ is generally slightly better than the one with R prime. But, as noticed earlier, this is the price to pay to get 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 IX, 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.*

Table IX

STORAGE AMOUNT COMPARISON FOR FIXED-BASE COMB, FIXED-BASE RADIX-R AND MODULAR EXPONENTIATION WITH MULTIPLICATIVE SPLITTING RECODING FOR NIST RECOMMENDED EXPONENT SIZES

Key size $t = 224$ bits

#MM	Fixed-base Comb	Fixed-base Radix-R	Multiplicative splitting	
			$R = m_0 m_1$	R-prime
45	127.5 kB	345 kB	**108** kB	240 kB
	$w = 9$	$R = 31$	$(m_0, m_1) = (11, 9)$	$(R, c) = (97, 7)$
37	511.5 kB	594 kB	**242** kB	541 kB
	$w = 11$	$R = 61$	$(31, 7)$	$(179, 5)$
30	4095.5 kB	1386 kB	**770** kB	**1205** kB
	$w = 14$	$R = 179$	$(127, 7)$	$(179, 5)$
24	32767.5 kB	4230 kB	**4173** kB	4489 kB
	$w = 17$	$R = 677$	$(877, 7)$	$(1223, 3)$
19	524287.5 kB	27084 kB	50409 kB	27954 kB
	$w = 21$	$R = 5417$	$(13441, 5)$	$(6211, 2)$

Key size $t = 256$ bits

#MM	Fixed-base Comb	Fixed-base Radix-R	Multiplicative splitting	
			$R = m_0 m_1$	R-prime
46	383 kB	845 kB	**241** kB	494 kB
	$w = 10$	$R = 47$	$(m_0, m_1) = (17, 11)$	$(R, c) = (97, 5)$
39	1535 kB	1454 kB	**579** kB	**1116** kB
	$w = 12$	$R = 97$	$47; 7$	$223; 5$
32	12287 kB	3179 kB	**2070** kB	**3084** kB
	$w = 15$	$R = 257$	$211; 6$	$409; 3$
26	98303 kB	9846 kB	**9642** kB	10207 kB
	$w = 18$	$R = 937$	$1223; 6$	$1699; 3$
20	1572863 kB	**66676** kB	225482 kB	85558 kB
	$w = 22$	$R = 8467$	$37579; 5$	$12007; 2$

Key size $t = 384$ bits

#MM	Fixed-base Comb	Fixed-base Radix-R	Multiplicative splitting	
			$R = m_0 m_1$	R-prime
63	1918 kB	4081 kB	**969** kB	2274 kB
	$w = 11$	$R = 67$	$(m_0, m_1) = (19, 11)$	$(R, c) = (127, 6)$
50	15358 kB	10087 kB	**3742** kB	**7182** kB
	$w = 14$	$R = 191$	$101; 11$	$433; 5$
41	122878 kB	26655 kB	**17284** kB	**22891** kB
	$w = 17$	$R = 677$	$541; 6$	$937; 3$
35	983038 kB	80357 kB	**64768** kB	**65837** kB
	$w = 20$	$R = 2381$	$2381; 6$	$3191; 3$
30	7864318 kB	246070 kB	315053 kB	**235255** kB
	$w = 23$	$R = 8467$	$13441; 5$	$13441; 3$
26	62914558 kB	**951217** kB	3256278 kB	1030642 kB
	$w = 26$	$R = 37579$	$165397; 5$	$43973; 2$
24	503316478 kB	**1750756** kB	- kB	- kB
	$w = 29$	$R = 74699$	$-$	$-$

Key size $t = 512$ bits

#MM	Fixed-base Comb	Fixed-base Radix-R	Multiplicative splitting	
			$R = m_0 m_1$	R-prime
86	3836 kB	9841 kB	**1940** kB	5004 kB
	$w = 11$	$R = 59$	$(m_0, m_1) = (13, 11)$	$(R, c) = (163, 9)$
73	15356 kB	17855 kB	**4747** kB	**10005** kB
	$w = 13$	$R = 127$	$(41, 10)$	$(241, 6)$
60	122876 kB	46775 kB	**16224** kB	**29979** kB
	$w = 16$	$R = 409$	$(179, 11)$	$(739, 5)$
52	491516 kB	93110 kB	**54680** kB	**76505** kB
	$w = 18$	$R = 937$	$(677, 7)$	$(1223, 3)$
48	983036 kB	156091 kB	**106185** kB	**136971** kB
	$w = 19$	$R = 1699$	$(1489, 10)$	$(2381, 3)$
41	7864316 kB	489112 kB	**355573** kB	477551 kB
	$w = 22$	$R = 6211$	$(5417, 7)$	$(6211, 2)$
35	62914556 kB	2048419 kB	2113890 kB	**1949934** kB
	$w = 25$	$R = 30347$	$(37579, 7)$	$(47269, 3)$

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 [18]. 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 $R = m_0 m_1$ 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 VIII). The timings given in the next subsection include this recoding.

- *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 corresponding exponentiation for each dataset and keep the minimum number of clock cycles. This avoids the cold-cache effect and system issues. The timings are obtained by averaging the timings of all datasets.

Tests results and comparison. The four considered exponentiation 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 operating system is CENTOS 7.0.1406. On this platform, the Random Access Memory is 12.6 GBytes. One notices 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 X, we provide the most significant results. The gains shown are roughly in the same order of magnitude as the one of the complexity evaluation. In particular, for the largest key size (512 bits), the storage of our approach with $R = m_0 m_1$ 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,

Table X

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.

Security level	Level of clock--cycles	Scalar multiplication											
		State of the art methods						Proposed approach					
		Fixed-base Comb			Radix R			$R = m_0 m_1$			R prime		
		Time (#CC)	Storage (kB)	w	Time (#CC)	Storage (kB)	R	Time (#CC)	Storage (kB)	(m_0, m_1)	Time (#CC)	Storage (kB)	(R, c)
112 bits	220000	221108	1023.5	12	227838	829	91	219864	**580**	(89,6)	217104	1191	(257,3)
(key 224 bits,	207000	210074	2047.5	13	206888	1324	163	207072	**766**	(127,7)	206813	1553	(347,3)
field 2048 bits)	148000	149690	65535	18	147877	7289	1223	146156	21599	(5417,6)	149490	17661	(3719,2)
128 bits	505000	524539	1535	12	502981	1411	91	501466	897	(79,6)	509581	1420	(307, 5)
(key 256 bits,	450000	449397	6143	14	445871	2251	163	446444	2056	(211,6)	458936	2372	(307, 3)
field 3072 bits)	354000	356892	98303	18	354640	6414	571	354071	12843	(1721,7)	353662	15283	(1699, 2)
192 bits	444000	4442590	1918	11	4492191	3430	53	4409584	1134	(23, 10)	4494471	2171	(127, 6)
(key 384 bits,	353000	3554339	15358	14	3524896	8290	163	3551437	4164	(113, 10)	3534620	7100	(433, 5)
field 7680 bits)	270000	2736341	245758	18	2543480	45221	1223	2743399	29961	(1031, 7)	2795363	31915	(1381, 3)
256 bits	1860000	18632429	15536	13	19260731	13765	91	18550238	4745	(41, 10)	18683547	8653	(257, 7)
(key 512 bits,	1500000	14848261	122876	16	15401002	34418	163	14812616	22111	(257, 11)	15541482	27853	(641, 5)
field 15360 bits)	1240000	12477816	983036	19	12193232	119061	1223	12499600	102820	(1381, 7)	12802926	101886	(1699, 3)

our approach with R prime gives equivalent results for low levels of storage, and better results for higher levels of storage.

### C. Complexities and timings for scalar multiplication

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

1) Complexity comparison: In the fixed-base elliptic curve scalar multiplication case, the main difference with the modular exponentiation 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 A 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 $R = m_0 m_1$ was never better than the one with R prime. In addition, the approach with $R = m_0 m_1$ does not provide a constant time computation. That is why we do not consider the approach with $R = m_0 m_1$ 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 situations, 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 $\mathbb{F}_p$ of point addition and

Figure 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).
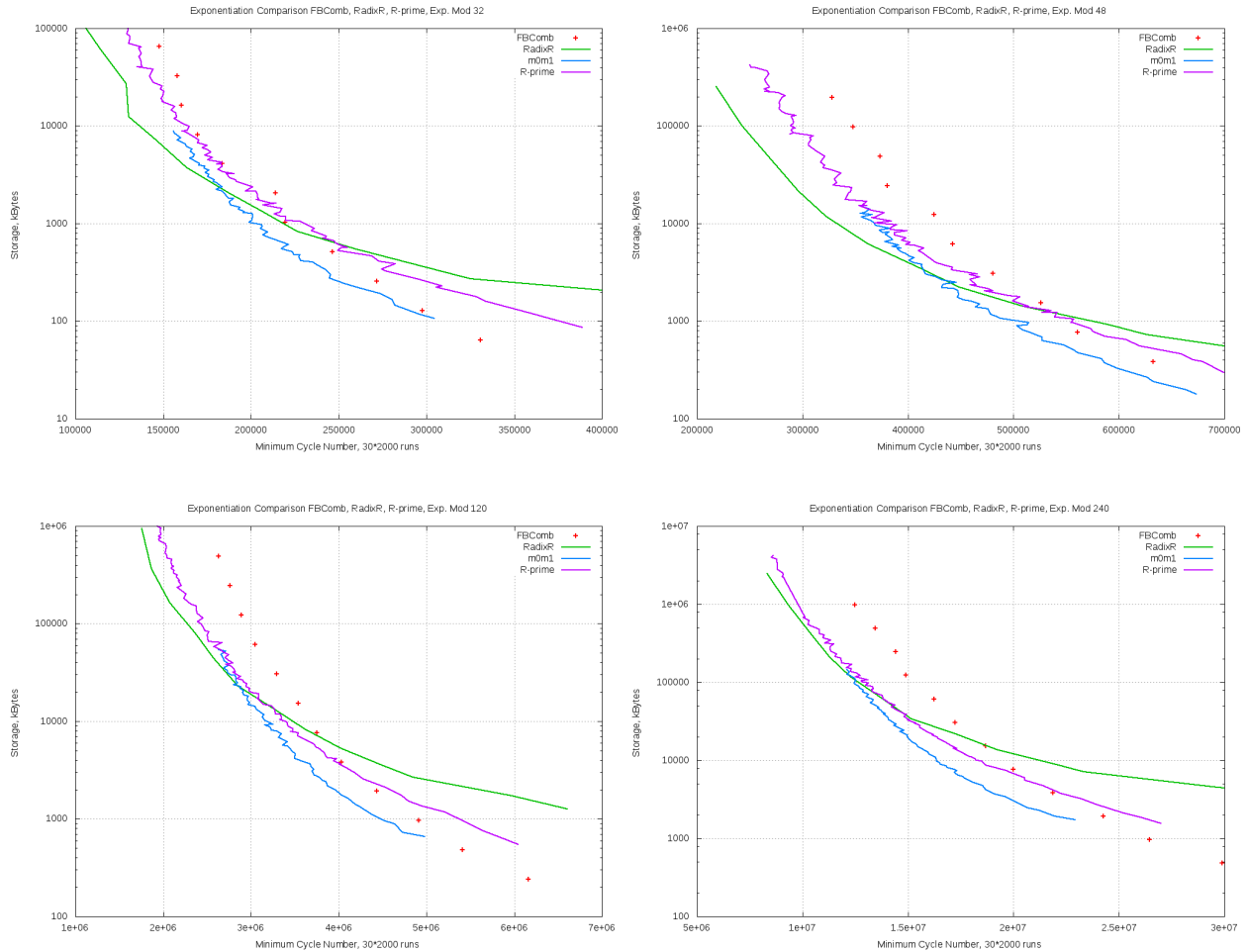
doubling for a Weierstrass curves of [2] to derive the complexity of the considered fixed-base scalar multiplication. The complexities of the curve operations in terms of the number of modular multiplications (MM) and squarings (MS) are as follows:

$$\text{Addition: } 12MM + 4MS,$$
$$\text{Doubling: } 4MM + 4MS,$$
$$\text{Mixed-Addition: } 7MM + 4MS.$$

The resulting complexities of the considered scalar multiplication approaches are reported in Table XI 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 XI, one notices that for the four considered methods, one has a slight difference in comparison with the modular exponentiation case. Indeed, for all key sizes and for most of the levels of online computation the proposed approach shows the lowest amount of storage. This is due to the relative cost of the doubling of point and the mixed and full Jacobian addition of points:

- Since the doubling is much cheaper than the additions, 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.

- 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 $\#MM \in \in \{352, ..., 651\}$.

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 V-B1 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 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. For the inversion, we use the binary extended Euclidean algorithm, with some specific assembly code portion, which is presented by Brown *et al.* in [6]. The field reductions use also the algorithms presented by Brown *et al.* in [6].

- *Radix R conversion and recoding.* The algorithm and the code is the same as the one previously used for modular exponentiation case. However, the size of the scalar in this case is the same as the one of the field elements representing the elliptic curve point coordinates. The computation time of the recoding is no more negligible in this case, as shown by the tests of the conversion alone, and the multiplicative splitting recoding computation (including the conversion). We provide in Table XIII 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.

  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 then process the

Table XI

STORAGE AMOUNT COMPARISON FOR CONSTANT TIME FIXED-BASE SCALAR MULTIPLICATION OVER NIST CURVES P256, P384, P521

NIST Curve P256

$\#MM$	Fixed-base Comb	Fixed-base Radix-R	Multiplicative splitting Regular R-prime
441	**64** kB	162 kB	73 kB
	$w = 10$	$R = 59$	$R = 163; C = 5$
405	128 kB	222.5 kB	**100** kB
	$w = 11$	$R = 89$	127; 3
317	1024 kB	566 kB	**334** kB
	$w = 14$	$R = 283$	571; 3
264	8192 kB	1522.5 kB	**1142** kB
	$w = 17$	$R = 937$	2381; 3
211	65536 kB	6235 kB	**5581** kB
	$w = 20$	$R = 4751$	8929; 2
176	1048576 kB	22192 kB	**22156** kB
	$w = 24$	$R = 19727$	66467; 3

NIST Curve P384

$\#MM$	Fixed-base Comb	Fixed-base Radix-R	Multiplicative splitting Regular R-prime
669	**96** kB	365 kB	127 kB
	$w = 12$	$R = 59$	$R = 149; C = 6$
475	1536 kB	1352.5 kB	**661** kB
	$w = 14$	$R = 307$	491; 3
370	24576 kB	5734 kB	**2901** kB
	$w = 18$	$R = 1699$	2729; 3
299	393216 kB	26693 kB	**17643** kB
	$w = 22$	$R = 9491$	13441; 2
264	3145728 kB	71250 kB	**51532** kB
	$w = 25$	$R = 29231$	43973; 2

NIST Curve P521

$\#MM$	Fixed-base Comb	Fixed-base Radix-R	Multiplicative splitting Regular R-prime
915	**144** kB	678 kB	234 kB
	$w = 10$	$R = 53$	$R = 157; C = 7$
651	2304 kB	2547 kB	**1146** kB
	$w = 14$	$R = 283$	739; 5
493	36864 kB	12750.5 kB	**6733** kB
	$w = 18$	$R = 1889$	3191; 3
405	589824 kB	47627 kB	**35915** kB
	$w = 22$	$R = 8467$	13441; 2
352	4718592 kB	153675 kB	**133905** kB
	$w = 25$	$R = 31223$	57709; 2

Figure 3. Complexity comparison for constant time Fixed-base scalar multiplication on elliptic Weierstrass curves P256, P384 and P521
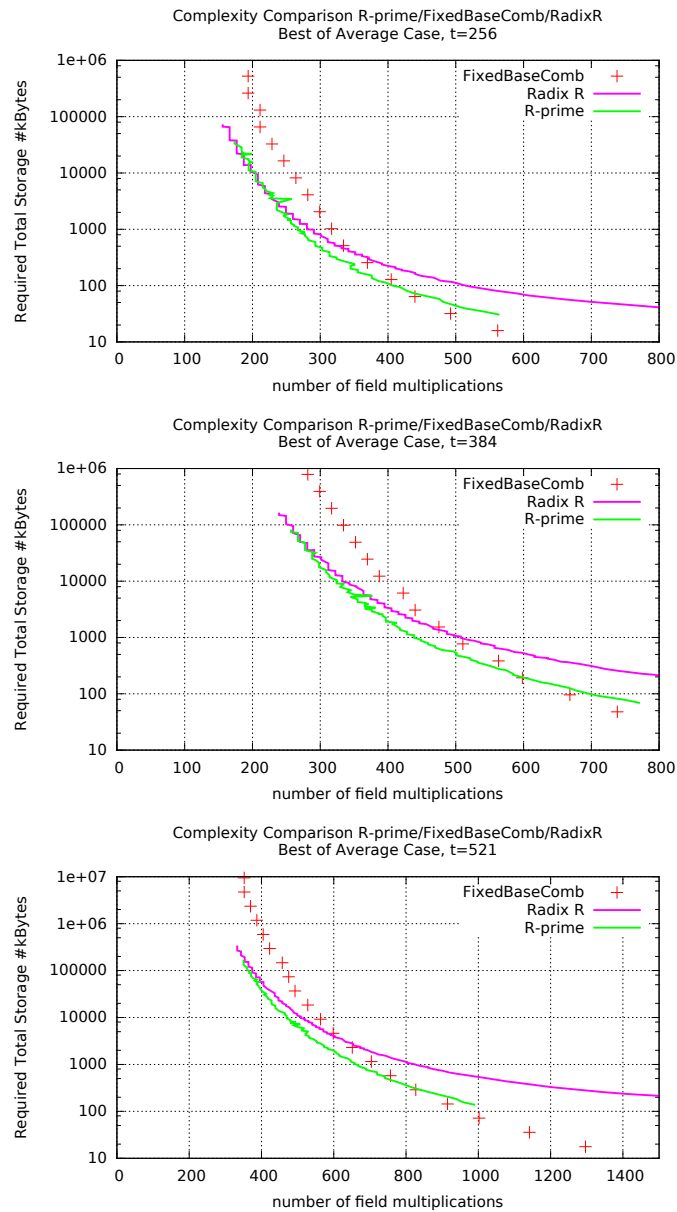
digital signature (this strategy was proposed in [7] to avoid costly scalar recoding in double base representation).

- *Test processing*. The test processing is the same as the one used for the modular exponentiation. This involves a few hundred of datasets, which are random scalars and a fixed base with precomputed data. To get the timings, we perform 2000 times the scalar multiplication and keep the minimal number of clock-cycles. This is meant to avoid the cold-cache effect and system interruptions. The final timings are the average of the dataset timings.

Test results and comparison. The algorithms were coded in C, compiled with `gcc 4.8.3`. The test were performed on a platform which has the following characteristics: an Intel XEON E5-2650 (Ivy bridge), a RAM of 12.6GBytes

Table XII

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.

Security level	Level of Clock-cycles	Scalar multiplication								
		State of the art methods						Proposed approach		
		Fixed-base Comb			radix R			R-splitting rec.		
		Time (#CC)	Storage (kB)	w	Time (#CC)	Storage (kB)	R	Time (#CC)	Storage (kB)	(R, c)
128 bits (NIST curve P256)	370000	378184	64	12	376370	74	19	366057	**37**	(71,5)
	276000	275230	1024	14	276917	231	89	276660	**170**	(257,3)
	205000	207456	32768	19	206777	1120	641	203414	**1012**	(1699,2)
192 bits (NIST curve P384)	575000	575854	192	11	571975	283	41	583590	**86**	(79,5)
	460000	461271	1536	14	470537	547	97	451846	**354**	(233,3)
	375000	376114	24576	18	372952	1861	433	378733	**1214**	(997,3)
	349000	359578	49151	19	360786	2069	491	354919	**1911**	(1699,3)
256 bits (NIST curve P521)	450000	446633	288	11	451280	572	41	449550	**146**	(97,7)
	364000	363615	2304	14	362166	1621	157	367299	**726**	(433,5)
	289000	289085	73728	19	288394	7217	937	290146	**6243**	(2897,3)

Table XIII

RECODING TESTS, FOR SIZES 256, 384 AND 521 BITS: WORST CASE COMPUTATION TIME IN CLOCK CYCLES

| Recoding | | | |
Scalar size	256 bits #CC	384 bits #CC	521 bits #CC
Radix R conversion	1200	1640	2250
R-splitting conversion	14400	21600	27500

and a CENTOS 7.0.1406 operating system.

In Table XII, we report some of the most significant results obtained for the three following approaches: *Fixed-base Comb*, *Fixed-base radix-R* and the one based on multiplicative splitting recoding with a prime radix R. These results show that, except in the last case (P521 and 290000 clock-cycles), our approach provides the smallest storage amount for each considered level of clock-cycles. This is consistent with the complexity evaluation shown in Table XI and Fig. 3. Specifically, for a fixed amount of online computation the proposed approach reduce by roughly 50% the amount of storage.

VI. 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 the 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/.

[2] Explicit Formula Database, 2014. http://www.hyperelliptic.org/EFD/.

[3] D.J. Bernstein and T. Lange. Faster addition and doubling on elliptic curves. In *Proceedings of ASIACRYPT 2007*, volume 4833 of *LNCS*, pages 29–50. Springer, 2007.

[4] R. Blank and Patrick Gallagher. Recommendation for Key Management, 2012.

[5] A. Bosselaers, R. Govaerts, and J. Vandewalle. Comparison of Three Modular Reduction Functions. In *CRYPTO '93*, pages 175–186, 1993.

[6] M. Brown, D. Hankerson, J. López, and A. Menezes. Software implementation of the NIST elliptic curves over prime fields. In *CT-RSA 2001*, pages 250–265, 2001.

[7] Christophe Doche. On the Enumeration of Double-Base Chains with Applications to Elliptic Curve Cryptography. In *Advances in Cryptology - ASIACRYPT 2014*, volume 8873 of *LNCS*, pages 297–316. Springer, 2014.

[8] D. M. Gordon. A Survey of Fast Exponentiation Methods. *J. Algorithms*, 27(1):129–146, 1998.

[9] D. Hankerson, J. López Hernandez, and A. Menezes. Software Implementation of Elliptic Curve Cryptography over Binary Fields. In *CHES 2000*, volume 1965 of *LNCS*, pages 1–24. Springer, 2000.

[10] M. Hedabou, P. Pinel, and L. Bénéteau. A comb method to render ECC resistant against Side Channel Attacks. Technical report, IACR Cryptology ePrint Archive, 2004.

[11] M. Joye and M. Tunstall. Exponent Recoding and Regular Exponentiation Algorithms. In *Proceedings of Africacrypt 2009*, LNCS, pages 334–349. Springer, 2009.

[12] C. Kerry and P. Gallagher. Digital Signature Standard (DSS), 2013.

[13] N. Koblitz. Elliptic curve cryptosystems, 1987.

[14] C.H. Lim and P.J. Lee. More flexible exponentiation with precomputation. In *Advances in Cryptology - Crypto '94*, volume 839 of *LNCS*, pages 95–107. Springer, 1994.

[15] A. Menezes, D. Hankerson, and S. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer, 2004.

[16] V. Miller. Use of elliptic curves in cryptography, 1985.

[17] N. Mohamed, M. Hashim, and M. Hutter. Improved Fixed-Base Comb Method for Fast Scalar Multiplication. In *Progress in Cryptology - AFRICACRYPT 2012*, volume 7374 of *LNCS*, pages 342–359. Springer, 2012.

[18] P. Montgomery. Modular Multiplication Without Trial Division. *Mathematics of Computation*, 44(170):519–521, 1985.

[19] C. Negre and T. Plantard. Efficient regular modular exponentiation using multiplicative half-size splitting. *J. of Cryptographic Engineering*, to appear.

[20] T. Plantard and J.-M. Robert. Enhanced Digital Signature Using RNS Digit Exponent Representation. In *WAIFI 2016*, pages 177–192, 2016.

[21] W.-J. Tsaur and C.-H. Chou. Efficient algorithms for speeding up the computations of elliptic curve cryptosystems. *Applied Mathematics and Computation*, 168(2):1045–1064, 2005.

APPENDIX

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 = 0xffffffff 00000001 00000000 00000000 00000000 ffffffff ffffffff
  ffffffff
  ```
  and
  ```
  b = 0x5ac635d8 aa3a93e7 b3ebbd55 769886bc 651d06b0 cc53b0f6 3bce3c3e
  27d2604b
  ```

  Curve subgroup generator:
  ```
  XG = 0x6b17d1f2 e12c4247 f8bce6e5 63a440f2 77037d81 2deb33a0 f4a13945
  d898c296
  YG = 0x4fe342e2 fe1a7f9b 8ee7eb4a 7c0f9e16 2bce3357 6b315ece cbb64068
  37bf51f5
  ```

- **P384:**

  One has $p = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$, that is

  ```
  p = 0xffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff
  fffffffe ffffffff 00000000 00000000 ffffffff
  ```
  and
  ```
  b = 0xb3312fa7 e23ee7e4 988e056b e3f82d19 181d9c6e fe814112 0314088f
  5013875a c656398d 8a2ed19d 2a85c8ed d3ec2aef
  ```

  Curve subgroup generator:
  ```
  XG = 0xaa87ca22 be8b0537 8eb1c71e f320ad74 6e1d3b62 8ba79b98 59f741e0
  82542a38 5502f25d bf55296c 3a545e38 72760ab7
  YG = 0x3617de4a 96262c6f 5d9e98bf 9292dc29 f8f41dbd 289a147c e9da3113
  b5f0b8c0 0a60b1ce 1d7e819d 7a431d7c 90ea0e5f
  ```

- **P521:**

  One has $p = 2^{521} - 1$, that is
  ```
  p = 0x01ff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff
  ```

```
ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff

ffffffff
```

and

```
b = 0x0051 953eb961 8e1c9a1f 929a21a0 b68540ee a2da725b 99b315f3 b8b48991

8ef109e1 56193951 ec7e937b 1652c0bd 3bb1bf07 3573df88 3d2c34f1 ef451fd4

6b503f00
```


Curve subgroup generator:

```
XG = 0x00c6 858e06b7 0404e9cd 9e3ecb66 2395b442 9c648139 053fb521 f828af60

6b4d3dba a14b5e77 efe75928 fe1dc127 a2ffa8de 3348b3c1 856a429b f97e7e31

c2e5bd66
```

```
YG = 0x0118 39296a78 9a3bc004 5c8a5fb4 2c7d1bd9 98f54449 579b4468 17afbd17

273e662c 97ee7299 5ef42640 c550b901 3fad0761 353c7086 a272c240 88be9476

9fd16650
```

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 [19].

Lemma 2. *Let v_i and r_i be the two sequences of coefficients computed in Algorithm 7. They satisfy the following properties:*

i) $(-1)^{i-1} v_i \geq 1$ *for all* $i \geq 1$.

ii) $v_{i+1} r_i - v_i r_i = (-1)^i R$ *for all* $i \geq 1$.

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

$$r_{i-1}|v_i| + r_i|v_{i-1}| = R. \tag{13}$$

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

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 XIV 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 $= \{(s_i, k_i^{(0)}, k_i^{(1)}), 0 \leq i < \ell, (k'_\ell)\}$ 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 $T[i][j] \leftarrow (\left|j^{-1}\right|_R \cdot R^i) \cdot P$ for $i = 0, \ldots, \ell-1, j = 1, \ldots, \lceil R/c \rceil$ and $T[\ell][1] \leftarrow R^\ell \cdot P$ and $T[i][0] \leftarrow \mathcal{O}$

for $i = 0, \ldots, \ell-1$.

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: $\quad Y_{k_i^{(0)}} \leftarrow Y_{k_i^{(0)}} + (s_i) \cdot T[i][k_i^{(1)}]$

5: $Y_{|k'_\ell|} \leftarrow Y_{|k'_\ell|} + (\text{sign}(k'_\ell)) \cdot T[\ell][1]$

6: **for** i **from** W **downto** 0 **do**

7: $\quad X \leftarrow 2 \times X$

8: $\quad$ **for** j **from** $c - 1$ **downto** 1 **do**

9: $\quad\quad$ **if** bit i of j is non zero **then**

10: $\quad\quad\quad X \leftarrow X + Y_j$

11: **return** (X)

Table XIV

COMPLEXITY EVALUATION OF SCALAR MULTIPLICATION BASED ON MULTIPLICATIVE SPLITTING RECODING WITH R PRIME

Complexity		
Step	Operation	Cost
$\ell \times$ Step 3	$Y_{k_i^{(0)}} + s_i \cdot T[i][k_i^{(1)}]$	$1 MixedAdd$
$1 \times$ Step 5	$Y_{|k'_\ell|} + sign(k'_\ell) \cdot T[\ell][1]$	$1 MixedAdd$
$(W-1) \times$ Step 7	$X \leftarrow 2 \times X$	$1 Doubling$
$(\mathcal{H}-1) \times$ Step 10	$X \leftarrow X + Y_j$	$1 Addition$
TOTAL	$(\ell+1) \times MixedAdd$ $+(W-1) \times Doubling + (\mathcal{H}-1) \times Addition$	
TOTAL STORAGE	$\ell \times (\lceil R/c \rceil + 1) + 1$ points on $E(\mathbb{F}_p)$	