Enhanced Digital Signature using RNS Digit Exponent Representation

Thomas Plantard, Jean-Marc Robert

Enhanced Digital Signature using RNS Digit Exponent Representation

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

[1] CCISR, SCIT, University of Wollongong, Australia
[2] Team DALI, Université de Perpignan Via Domitia, France
[3] LIRMM, UMR 5506, Université Montpellier and CNRS, France

Abstract. Digital Signature Algorithm (DSA) involves modular exponentiation, 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. However, due to the cost of the memory, to its small size and to the latency of access, previous research sought for minimization of the storage. In this paper, taking into account the modern processor features and the growing size of the cache memory, we improve the storage/efficiency trade-off, by using a RNS Digit exponent representation. We then propose algorithms for modular exponentiation. The storage is lower for equivalent complexities for modular exponentiation computation. The implementation performances show significant memory saving, up to 3 times for the largest NIST standardized key sizes compared to state of the art approaches.

Keywords: RNS, Digital Signature, Modular Exponentiation, Memory Storage, Efficient Software Implementation.

1 Introduction

In the DSS (Digital Signature Standard), DSA (Digital Signature Algorithm) is a popular authentication protocol. According to the NIST standard (see [3]), the public parameters are p, q and g. The parameter g is a generator of the multiplicative group $\mathbb{Z}/p\mathbb{Z}$ of order q, which is a prime of size corresponding to the required security level. Therefore, p is a prime chosen such that q divides $p-1$. The recommended security levels in the standard are 80-256 bits, corresponding to 160-512 bit sizes for the prime q. When a server needs to sign a batch of documents or authentications, the main operations are modular exponentiations $g^k \bmod p$ (one per signature), where k is a one time random parameter. Taking advantage of the fixed public parameter g is a natural way to speed-up the signature protocol, by storing well chosen powers of g. The main known methods of the state of the art are the one presented by Gordon in [6], which stores the $g^{R^i} \bmod p$ values, and also the *Fixed-base Comb*, which is presented by Lim and Lee in [10]. While improving the complexity, and therefore, lowering the computation time, these methods require some storage. The trade-off between

the efficiency and the storage amount is the comparison criteria between these different approaches.

All the protocols derived from the DSA can use these different approaches, since they all need an exponentiation of a known base by a random exponent. Blind signature and E-voting are examples of protocols using fixed-base modular exponentiation ([11]). Moreover, El-Gamal encryption and signature also use a public generator g to the power of a randomly chosen exponent (see [4]). However, the decryption uses the result of this operation, and the idea does not apply in this case.

On the arithmetic side, the Residue Number System (RNS), based on the Chinese Remainder Theorem, is a classical way to speed-up and/or parallelize arithmetic computations and was first presented by Svoboda in [14] and by Garner in [5]. One can find a complete classical presentation of the RNS in Knuth [9].

Contributions: In this paper, we propose to use the memorization of base powers with numeration scales in radix $R = m_0 \cdot m_1$ and the RNS representation of each digit using the base $\mathcal{B} = \{m_0, m_1\}$. We study the recoding algorithm and apply it to the exponent in modular exponentiation. We propose a modular exponentiation algorithm using this recoding of the exponent and memorization. We called this algorithm the m_0m_1 exponentiation method. We studied the corresponding complexities and storage amounts, and compared the results with the *Fixed-base Comb* and Radix-R methods. We showed that our m_0m_1 exponentiation method has better storage/complexity trade-off that the aforementioned methods, for the NIST recommended field sizes and a large range of storage amount. We then made software implementations of our algorithms and performed tests in order to validate the storage/timing trade-off. The speed-up comparison shows the benefits. This approach provides also a fair 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.

This paper is organized as follows: Section 2, we review the main classical fixed-base exponentiation algorithms, taking advantage of storage and give their complexities and storage requirements; Section 3, we present our approach for the m_0m_1 recoding; Section 4, we then show the application on modular exponentiation; Section 5 shows the implementation strategies and results in terms of performances we got in this work; finally Section 6, we give some concluding remarks and perspectives.

2 State of the Art Review

In this Section, we review the state of the art classical approaches for fixed-base modular exponentiation.

When a server needs to sign a document or a message, the computation consists of several operations, the main one being a modular exponentiation

$g^k \bmod p$, with k being a one-time random exponent. This computation can use the classical Square-and-Multiply algorithm (see Algorithm 1). In terms of complexity, given the exponent length t (that is, the size of the prime q), the number of modular squaring is $t-1$ and the number of modular multiplications to be computed is $t/2$ on average, half of this length, for a randomly chosen exponent. There is no storage in this case.

The method presented by Gordon in [6] first suggests to store the t successive squarings of g (that is the sequence of g^{2^i}). In terms of complexity, given the exponent length t (again, the size of the prime q), one has now no squarings and the number of modular multiplications to be computed is half of this length on average as in the previous case, for a randomly chosen exponent. The storage amount is t values in $\mathbb{Z}/p\mathbb{Z}$ as mentioned above. Gordon in [6] mentions the generalization of this idea into a radix R method, which consists of the memorization of the values $g^{i \cdot R^j}$, with $i \in [1, ..., R-1]$ and $0 \leq j < \ell$ where ℓ is the radix R length of the exponent, which we denote by $w = \log_2(R)$ ($\ell = \lceil t/w \rceil$). In this case, the complexity is $\ell - 1$ modular multiplications, for a storage amount of $\ell \cdot (R-1)$ values in $\mathbb{Z}/p\mathbb{Z}$. In the sequel, we will call this approach the Radix-R Exponentiation Method (see Algorithm 2).

Algorithm 1 Left-to-Right Square-and-Multiply Modular Exponentiation

Require: $k = (k_{t-1}, \ldots, k_0)$, the DSA modulus p, g a generator of $\mathbb{Z}/p\mathbb{Z}$ of order q.
Ensure: $X = g^k \bmod p$
1: $X \leftarrow 1$
2: **for** i from $t-1$ downto 0 **do**
3: $\quad X \leftarrow X^2 \bmod p$
4: $\quad$ **if** $k_i = 1$ **then**
5: $\quad\quad X \leftarrow X \cdot g \bmod p$
6: $\quad$ **end if**
7: **end for**
8: **return** (X)

Algorithm 2 Radix-R Exponentiation Method

Require: $k = (k_{\ell-1}, \ldots, k_0)_R$, the DSA modulus p, g a generator of $\mathbb{Z}/p\mathbb{Z}$ of order q.
Ensure: $X = g^k \bmod p$
1: *Precomputation.* Store $G_{i,j} \leftarrow g^{i \cdot R^j}$, with $i \in [1, ..., R-1]$ and $0 \leq j < \ell$.
2: $X \leftarrow 1$
3: **for** i from $\ell - 1$ downto 0 **do**
4: $\quad X \leftarrow X \cdot G_{k_i,i} \bmod p$
5: **end for**
6: **return** (X)

Algorithm 3 *Fixed-base Comb* Exponentiation Method

Require: $k = (k_{t-1}, \ldots, k_1, k_0)_2$, the DSA modulus p, g a generator of $\mathbb{Z}/p\mathbb{Z}$ of order q, window width w, $d = \lceil t/w \rceil$.
Ensure: $X = g^k \bmod p$
1: *Precomputation.* Compute and store $g^{[a_{w-1},\ldots,a_0]} \bmod p, \quad \forall (a_{w-1}, \ldots, a_0) \in \mathbb{Z}_2^w$.
2: By padding k on the left by 0's if necessary, write $k = K^{w-1} \| \ldots \| K^1 \| K^0$, where each K^j is a bit string of length d. Let K_i^j denote the i^{th} bit of K^j.
3: $X \leftarrow 1$
4: **for** i from $d-1$ downto 0 **do**
5: $\quad X \leftarrow X^2 \bmod p$
6: $\quad X \leftarrow X \cdot g^{[K_i^{w-1},\ldots,K_i^1,K_i^0]} \bmod p$
7: **end for**
8: **return** (X)

Another classical method is the so called *Fixed-base Comb* method. In [8], Hankerson *et al.* present this method proposed by Lim and Lee in [10]. The window width w is the number of comb-teeth, and $d = \lceil t/w \rceil$ is the distance in bits between two teeth. This method is shown in Algorithm 3, in which we denote $[a_{w-1}, \ldots, a_1, a_0] = a_{w-1}2^{(w-1)d} + \ldots + a_2 2^{2d} + a_1 2^d + a_0$. The complexity of this approach is $d-1$ modular squarings and d mutiplications, for a storage amount of $2^w - 1$ values in $\mathbb{Z}/p\mathbb{Z}$. One drawback of this method is the lack of flexibility for the storage amount, which increases exponentially with respect to the window width w.

Table 1, we give the complexities and the storage amounts of all these approaches.

Table 1. Complexities and storage amounts of state of the art methods, average case, binary exponent length t. #MM denotes the number of modular multiplications, #MS the number of modular squarings.

	# MM	# MS	storage (# values $\in \mathbb{Z}/p\mathbb{Z}$)
Square-and-multiply, Algo. 1	$t/2$	$t-1$	-
Radix-R method, Algo. 2	$\lceil t/w \rceil$	-	$\lceil t/w \rceil \cdot (R-1)$
Fixed-base Comb, Algo 3	$d = \lceil t/w \rceil$	$d-1$	$2^w - 1$

3 m_0m_1 Recoding

In this section, we present our approach for the m_0m_1 recoding. Our goal is to use this representation in a modular exponentiation computation. The RNS digit representation with two moduli splits the digits in two parts. The first part will be used to select the precomputed values and the second part for final computation of the modular exponentiation, with the best possible trade-off.

3.1 Algorithm

We first remind the RNS representation with RNS base $\mathcal{B} = \{m_0, m_1\}$ of two moduli. 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 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:

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

We now present our recoding approach. Our idea here is to use an exponent k recoding in radix $R = m_0 \cdot m_1$. We represent every radix-R digits in RNS with RNS base $\mathcal{B} = \{m_0, m_1\}$. Let k_i be the digits of k in radix-R, and let us denote $(k_i^{(0)}, k_i^{(1)})$ their RNS representations in base $\mathcal{B}$. Thus, one has:

$$k = \textstyle\sum_{i=0}^{\ell-1} k_i R^i, \text{ with } \ell = \lceil t/\log_2(R) \rceil,$$
$$\text{and } \begin{cases} k_i^{(0)} = |k_i|_{m_0}, \\ k_i^{(1)} = |k_i|_{m_1}. \end{cases}$$

Let us denote (when $k_i^{(1)} \neq 0$)

$$\begin{aligned} m_0' &= m_1 \cdot |m_1^{-1}|_{m_0}, \\ m_1' &= m_0 \cdot |m_0^{-1}|_{m_1}, \\ k_i' &= |k_i^{(0)} \cdot (k_i^{(1)})^{-1}|_{m_0}. \end{aligned}$$

One keeps $\kappa_i \leftarrow (k_i', k_i^{(1)})$ as a representation of k_i and this leads to $k_i = \left| k_i^{(1)} |k_i' \cdot m_0' + m_1'|_R \right|_R$. We handle the modular reduction $\bmod R$ as follows:

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

Let us denote $C = \lfloor k_i^{(1)} \cdot (k_i' \cdot m_0' + m_1')/R \rfloor$. By noticing that $0 \leq C < m_1$, we now consider C as a carry that one can subtract to k_{i+1}. We then compute

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

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

When $k_i^{(1)} = 0$, we handle this by slightly rewriting κ_i as follows: $\kappa_i = (|k_i^{(0)} + 1|_{m_0} \cdot m_0' - m_0')$, thus keeping $\kappa_i \leftarrow (|k_i^{(0)} + 1|_{m_0}, 0)$ as a representation of k_i in this case. In addition, one notices that the carry C is not modified here (it is either 0 or 1 and has been previously settled).

The sequence of the $\kappa_i \leftarrow (k_i', k_i^{(1)})$ is the $m_0 m_1$ recoding of k we can use to compute a modular exponentiation.

One notices it might be necessary to process a last carry C, with a final correction. The recoding algorithm is shown in Algorithm 4.

Algorithm 4 m_0m_1 Recoding

Require: $\{m_0, m_1\}$ RNS base with $R = m_0 \cdot m_1$, $k = \sum_{i=0}^{\ell-1} k_i R^i$.
Ensure: $\{\kappa_i, 0 \leq i < \ell, (C)\}$, m_0m_1 recoding of scalar k.
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: **end if**
7: $k_i^{(0)} = |k_i|_{m_0}, k_i^{(1)} = |k_i|_{m_1}$.
8: **if** $k_i^{(1)} = 0$ **then**
9: $\kappa_i \leftarrow (|k_i^{(0)} + 1|_{m_0}, 0)$
10: **else**
11: $k'_i \leftarrow |k_i^{(0)} \cdot (k_i^{(1)})^{-1}|_{m_0}$
12: $C \leftarrow C + \lfloor k_i^{(1)} \cdot |k'_i \cdot m'_0 + m'_1|_R / R \rfloor$
13: $\kappa_i \leftarrow (k'_i, k_i^{(1)})$
14: **end if**
15: **end for**
16: **return** $\{\kappa_i, 0 \leq i < \ell, (-C)\}$

3.2 Example

We present here an example of m_0m_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 therefore

$$m'_0 = 8 \cdot |8^{-1}|_{11} = 56,$$
$$m'_1 = 11 \cdot |11^{-1}|_8 = 33.$$

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 4, which consists of a `for` loop, steps 2 to 15.

- 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 6 since $k_0 \geq 0$.
  - Step 7, 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 6 and 7, since $k_0^{(1)} = 0$, one sets

$$\kappa_0 \leftarrow (|k_0^{(0)} + 1|_{11}, 0) = (5, 0)$$

- In the second iteration ($i = 1$), one has $k_1 = 78$.

- One has $C \leftarrow 0$ and one skips the `if`-test steps 4 to 6 since $k_1 \geq 0$.
- Step 7, 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 11 and 12, since $k_1^{(1)} \neq 0$, one has

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

and finally

$$\kappa_1 \leftarrow (2, 6)$$

– 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 computation steps 11-12 gives $C \leftarrow 2$, and

$$\kappa_2 \leftarrow (8, 5).$$

Without providing all the details, one finally gives the values returned by the algorithm:

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

4 m_0m_1 Modular Exponentiation

4.1 Algorithm

We now present the use of our recoding in the modular exponentiation. One wants to compute

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

with

$$g^{\kappa_i \cdot R^i} \bmod p = g^{\kappa_1^{(1)} \cdot R^i \cdot |\kappa_i' \cdot m_0' + m_1'|_R} \bmod p, \text{ when } \kappa_1^{(1)} \neq 0$$

and

$$g^{\kappa_i \cdot R^i} \bmod p = g^{R^i \cdot |\kappa_i' \cdot m_0' + m_1'|_R} \cdot g^{-R^i \cdot |m_0' + m_1'|_R} \bmod p, \text{ when } \kappa_1^{(1)} = 0.$$

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

$$G_{i,j} = g^{R^i \cdot |j \cdot m_0' + m_1'|_R} \bmod p, \text{ with } 0 \leq i \leq \ell - 1, 0 \leq j < m_0$$

$$\text{and } G_{\ell,1} = g^{R^\ell \cdot |m'_0+m'_1|_R} \bmod p.$$

The field inversion is very costly over $\mathbb{Z}/p\mathbb{Z}$, therefore, one also stores the following inverses:

$$G_{i,-1} = g^{-|m'_0+m'_1|_R \cdot R^i} \bmod p \text{ avec } 0 \leq i \leq \ell$$

One uses one value K_j per possible values of $1 \leq \kappa_i^{(1)} < m_1$, that is m_1 points. Thus, one now has

$$K_j = \left(\prod_{\text{for all } \kappa_i^{(1)}=j} G_{i,\kappa_i^{(0)}} \right) \times \left(G_{\ell,sign(C)1}\right)_{|C|=j} \bmod p$$

and

$$K_0 = \prod_{\text{for all } \kappa_i^{(1)}=0} G_{i,\kappa_i^{(0)}} \times G_{i,-1} \bmod p.$$

This leads to

$$g^k \mod p = K_0 \times \prod_{j=1}^{m_1} K_j^j.$$

Every single individual modular exponentiation K_j^j is performed with a square-and-multiply approach, which is more efficient than performing $j-1$ modular multiplications, even for small m_1.

One may notice that the amount of storage is now $(m_0+1) \times \ell + 1$ points. This approach is depicted in Algorithm 5.

4.2 Example

We now go back to our previous example in Section 3.2 page 6. One considers again the same values and parameters:

- an exponent size t of 20 bits ($0 < k < 2^{20}$), and $\mathcal{B} = \{11, 8\}$ (i.e. $m_0 = 11, m_1 = 8$);
- radix $R = m_0 \cdot m_1 = 88$ ($\ell = 4$);
- one has $k = 936192_{10}$;
- and we use the $m_0 m_1$ recoding previously computed:

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

We present the computation of $g^k \bmod p$ using Algorithm 5. In terms of storage, one computes the values

$$G_{i,j} = g^{R^i \cdot |j \cdot m'_0+m'_1|_R} \mod p \text{ with } 0 \leq i \leq \ell - 1.$$

One has the following values of $|j \cdot m'_0 + m'_1|_R$ for $0 \leq j < 11$:

$$\{33, 1, 57, 25, 81, 49, 17, 73, 41, 9, 65\}$$

Algorithm 5 Fixed-base m_0m_1 method modular exponentiation

Require: $\{m_0, m_1\}$ RNS base with $R = m_0m_1, k = \sum_{i=0}^{\ell-1} k_i R^i$ and $\kappa = \{\kappa_i, 0 \leq i < \ell, (C)\}$ the m_0m_1 recoding of k, p, the DSA modulus, $g \in \mathbb{Z}/p\mathbb{Z}$, public generator of order q.
Ensure: $X = g^k \mod p$
1: *Precomputation.* Store $G_{i,j} \leftarrow g^{R^i \cdot |j \cdot m_0' + m_1'|_R}$ with $0 \leq i < \ell - 1, 0 \leq j < m_0, G_{\ell,1} \leftarrow g^{R^\ell \cdot |m_0' + m_1'|_R}, G_{i,-1} \leftarrow g^{-R^i \cdot |m_0' + m_1'|_R}, 0 \leq i \leq \ell$
2: $A \leftarrow 1, K_j \leftarrow 1$ for $0 \leq j < m_1$
3: **for** i from 0 to $\ell - 1$ **do**
4: **if** $\kappa_i^{(1)} = 0$ **then**
5: $K_0 \leftarrow K_0 \times G_{i,\kappa_i^{(0)}} \times G_{i,-1}$
6: **else**
7: $K_{\kappa_i^{(1)}} \leftarrow K_{\kappa_i^{(1)}} \times G_{i,\kappa_i^{(0)}}$
8: **end if**
9: **end for**
10: $K_{|C|} \leftarrow K_{|C|} \times G_{\ell, sign(C)1}$
11: $W \leftarrow$ size of m_1 in bits
12: **for** i from W downto 0 **do**
13: $A \leftarrow A^2$
14: **for** j from $m_1 - 1$ downto 1 **do**
15: **if** bit i of j is non zero **then**
16: $A \leftarrow A \times K_j$
17: **end if**
18: **end for**
19: **end for**
20: **return** $(A \times K_0)$

In our case, with the chosen parameters, this brings us to store the following values in $\mathbb{Z}/p\mathbb{Z}$:

$$G_i = \{g^{88^i \cdot 33}, g^{88^i}, g^{88^i \cdot 57}, g^{88^i \cdot 25}, g^{88^i \cdot 81}, g^{88^i \cdot 49}, g^{88^i \cdot 17}, g^{88^i \cdot 73}, g^{88^i \cdot 41}, g^{88^i \cdot 9}, g^{88^i \cdot 65}\}.$$

We now use κ in Algorithm 5.

- the first steps are a `for` loop (steps 3 to 9):
  - in the first iteration, one has $\kappa_0^{(1)} = 0$ (and $\kappa_0^{(0)} = 5$), and this gives
$$K_0 \leftarrow G_{0,\kappa_0^{(0)}} \times G_{0,-1} = g^{49} \times g^{-1} = g^{48}.$$
  - in the second iteration, one has $\kappa_1^{(1)} = 6$ (and $\kappa_1^{(0)} = 2$), and this gives
$$K_6 \leftarrow G_{1,\kappa_1^{(0)}} = g^{88 \cdot 57} = g^{5016}.$$
  - in the third iteration, one has $\kappa_2^{(1)} = 5$ (and $\kappa_2^{(0)} = 8$), and this gives
$$K_5 \leftarrow G_{1,\kappa_2^{(0)}} = g^{88^2 \cdot 41} = g^{317504}.$$

- in the fourth and last iteration, one has $\kappa_2^{(1)} = 7$ (and $\kappa_2^{(0)} = 3$), and this gives
$$K_7 \leftarrow G_{1,\kappa_2^{(0)}} = g^{88^3 \cdot 25} = g^{17036800}.$$

– the last carry $C = -2$ is now processed (step 10):
$$K_2 \leftarrow G_{4,-1} = g^{88^4 \cdot (-1)} = g^{-59969536}.$$

– the reconstruction in the second `for` loop (steps 12 to 16) provides the final result by computing
$$\begin{aligned} g^k \bmod p &= K_0 \times \textstyle\prod_{j=1}^{m_1} K_j^j \bmod p \\ &= g^{48+2\cdot(-59969536)+5\cdot 317504+6\cdot 5016+7\cdot 17036800} \bmod p \\ &= g^{936192} \bmod p, \end{aligned}$$

which is the desired result.

4.3 Complexity

The complexity of Algorithm 5 is evaluated step by step in Table 2 for the average case. The number of field multiplications (MM) is evaluated as follows:

– the MMs in step 5 are performed only in case of $K_0 \neq 1$, instead, it is only an instantiation of K_0;
– the MMs in step 7 are performed only in case of $K_{\kappa_i^{(1)}} \neq 1$, instead, it is only an instantiation of $K_{\kappa_i^{(1)}}$;
– the same applies for step 10;

This saves on average m_1 MMs, and this is taken into account in the Total line in Table 2 (it explains the $-m_1$ term). The number of operations in the final reconstruction is evaluated as follows:

– the modular squaring in step 13 is performed only in case of $A \neq 1$;
– the MMs in step 15 and 18 are performed only in case of $K_j \neq 1$;

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 (`foreach` loop step 14). The value of $\mathcal{H}$ is as follows for the different values of m_1 :

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

We now discuss the complexity comparison of the considered methods (*Fixed-base Comb* Algorithm 3, Radix-R Algorithm 2 and m_0m_1 Algorithm 3). Since the parameters are very different between these three methods, a formal comparison is difficult. Therefore, we present a comparison based on numerical application, for NIST recommended sizes. In the sequel of this section, we then provide complexity evaluations in terms of field multiplications MM, under the assumption

Table 2. m_0m_1 modular exponentiation complexity and storage, average case.

Complexity		
Step	Operation	Complexity
$\ell/m_1\times$ step 5	$K_0 \leftarrow K_0 \times G_{i,\vert\kappa_i^{(0)}+1\vert_{m_0}} \times G_{i,-1}$	2 MM
$\ell\frac{m_1-1}{m_1}\times$ step 7	$K_{\kappa_i^{(1)}} \leftarrow K_{\kappa_i^{(1)}} \times G_{i,\kappa_i^{(0)}}$	1 MM
$1\times$ step 10	$K_{\vert C\vert} \leftarrow K_{\vert C\vert} \times G_{\ell,1}^{sign(C)}$	1 MM
$(W-1)\times$ step 13	$A \leftarrow A^2$	1 MS
$(\mathcal{H}-1)\times$ steps 15	$A \leftarrow A \times K_j$	1 MM
$1\times$ step 18	$(A \times K_0)$	1 MM
TOTAL	$(\ell\frac{m_1+1}{m_1} - m_1 + \mathcal{H})$ MM $+(W-1)$ MS	
TOTAL STORAGE	$(m_0+1)\times\ell + m_1 + 2$ elements of $\mathbb{Z}/p\mathbb{Z}$	

of squaring MS = 0.86 MM, which is the average value of our implementations for the NIST DSA recommended field sizes.

Figure 1 gives the general behavior of the three algorithms in terms of storage with respect to the complexity. One can see that the *Fixed-base Comb* method is the best for small storage amount. Our m_0m_1 approach is better for larger amount of storage, however, the Radix-R method is the best when the storage is increasing. In the figure, the field size is the largest of the ones recommended in the NIST standards (see [13]). Thus, the storage amount for such size is very big. Nevertheless, the behavior is roughly the same for smaller sizes, although the benefit of our approach is lower. The NIST provides recommended key sizes and corresponding field size (respectively the size of the primes q and p, see NIST SP800-57 [13]). This standardized sizes are as follows:

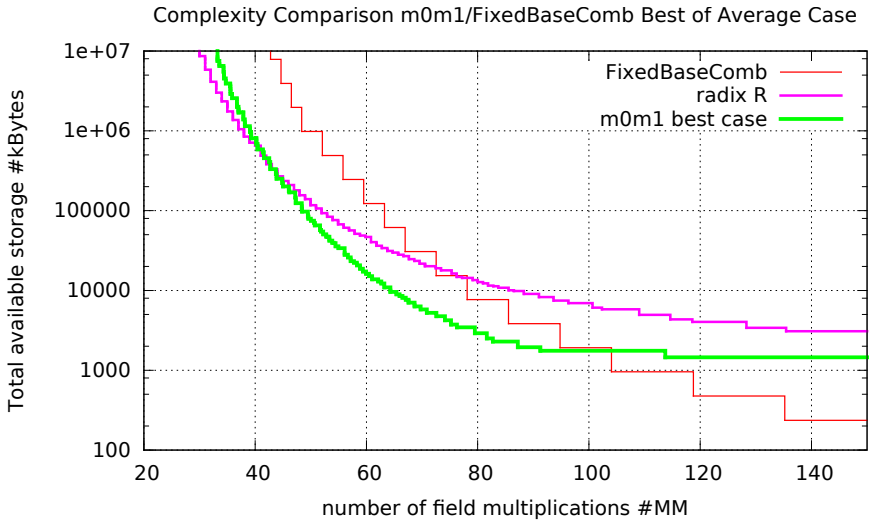
key size (bits)	160	224	256	384	512
field size (bits)	1024	2048	3072	7680	15360

For these sizes, Table 3 shows the complexity comparison between the *Fixed-base Comb* Algorithm 3, the Radix-R Method Algorithm 2 and our m_0m_1-Recoding approach Algorithm 5. For an equivalent number of MMs, we provide the minimum amount of storage.

We now provide a few comments about this table.

- For all sizes, we do not provide the results for small amount of storage (values of $w < 8$). For such storage, the *Fixed-base Comb* method is the best. One may notice that the Radix-R approach needs the greatest storage at this complexity level.
- For intermediate values of complexity, our proposed m_0m_1 approach shows the best storage/complexity trade-off. However, the benefits are greater for the larger key sizes.
  - $t = 224$, the best gain of our m_0m_1 approach is for $\#MM \approx 24$, with a storage 5 to 8 times smaller than the storage required for the *Fixed-base*

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



Comb method, respectively for $\#MM = 30$ and $\#MM = 24$, and 35% less than the one of the Radix-R method. However, below $\#MM \approx 24$, the Radix-R approach is better.

- $t = 256$, the best gain of our m_0m_1 approach is for $\#MM \approx 32$, with a storage about 6 times smaller than the storage required for the *Fixed-base Comb* method, and 44% less than the one of the Radix-R method. Again, with decreasing values of $\#MM$ (below 26), the Radix-R approach is better.
- $t = 384$, the best gain of our m_0m_1 approach is for $\#MM \approx 35$, with a storage about 15 times smaller than the storage required for the *Fixed-base Comb* method, and 19% less than the one of the Radix-R method. Again, with decreasing values of $\#MM$ (below 33), the Radix-R approach is better.
- $t = 512$, the best gain of our m_0m_1 approach is for $\#MM \approx 41$, with a storage about 22 times smaller than the storage required for the *Fixed-base Comb* method, and 27% less than the one of the Radix-R method. Again, with decreasing values of $\#MM$ (below 38), the Radix-R approach is better.

However, one may notice that the bigger memory storage sizes exceed the common values of Random Access Memory, and also the maximum allowed for the `malloc` function of the standard C library for memory allocation. Neverthe-

Table 3. Storage amount comparison, *Fixed-base Comb* method and m_0m_1 modular exponentiation fixed-base, average case, NIST recommended exponent sizes.

Key size $t = 224$ bits

$\#MM$	*Fixed-base C.*	Radix-R	m_0m_1
45	127.5 kB	345 kB	**108** kB
	$w = 9$	$R = 31$	$m_0 = 11; m_1 = 9$
37	511.5 kB	594 kB	**242** kB
	$w = 11$	$R = 61$	31; 7
30	4095.5 kB	1386 kB	**770** kB
	$w = 14$	$R = 179$	127; 7
24	32767.5 kB	4230 kB	**4173** kB
	$w = 17$	$R = 677$	877; 7
19	524287.5 kB	**27084** kB	50409 kB
	$w = 21$	$R = 5417$	13441; 5

Key size $t = 256$ bits

$\#MM$	*Fixed-base C.*	Radix-R	m_0m_1
46	383 kB	845 kB	**241** kB
	$w = 10$	$R = 47$	$m_0 = 17; m_1 = 11$
39	1535 kB	1454 kB	**579** kB
	$w = 12$	$R = 97$	47; 7
32	12287 kB	3179 kB	**2070** kB
	$w = 15$	$R = 257$	211; 6
26	98303 kB	**9486** kB	9642 kB
	$w = 18$	$R = 937$	1223; 6
20	1572863 kB	**66676** kB	225482 kB
	$w = 22$	$R = 8467$	37579; 5

Key size $t = 384$ bits

$\#MM$	*Fixed-base C.*	Radix-R	m_0m_1
63	1918 kB	4081 kB	**969** kB
	$w = 11$	$R = 67$	$m_0 = 19; m_1 = 11$
50	15358 kB	10087 kB	**3742** kB
	$w = 14$	$R = 191$	101; 11
41	122878 kB	26655 kB	**17284** kB
	$w = 17$	$R = 677$	541; 6
35	983038 kB	80357 kB	**64768** kB
	$w = 20$	$R = 2381$	2381; 6
30	7864318 kB	**246070** kB	315053 kB
	$w = 23$	$R = 8467$	13441; 5
26	62914558 kB	**951217** kB	3256278 kB
	$w = 26$	$R = 37579$	165397; 5
24	503316478 kB	**1750756** kB	- kB
	$w = 29$	$R = 74699$	–

Key size $t = 512$ bits

$\#MM$	*Fixed-base C.*	Radix-R	m_0m_1
86	3836 kB	9841 kB	**1940** kB
	$w = 11$	$R = 59$	$m_0 = 13; m_1 = 11$
73	15356 kB	17855 kB	**4747** kB
	$w = 13$	$R = 127$	41; 10
60	122876 kB	46775 kB	**16224** kB
	$w = 16$	$R = 409$	179; 11
52	491516 kB	93110 kB	**54680** kB
	$w = 18$	$R = 937$	677; 7
48	983036 kB	156091 kB	**106185** kB
	$w = 19$	$R = 1699$	1489; 10
41	7864316 kB	489112 kB	**355573** kB
	$w = 22$	$R = 6211$	5417; 7
35	62914556 kB	**2048419** kB	2113890 kB
	$w = 25$	$R = 30347$	37579; 7

less, the storage savings proposed by our method and the Radix-R ones allow to keep the level under the limit for lower complexities.

As a conclusion, our m_0m_1 approach shows lower storage amount for intermediate values of storage, whatever the standardized key size.

5 Implementation results

5.1 Implementation strategies

We review hereafter the main implementation strategies and test process. This applies for the three considered exponentiation algorithms. The algorithms were coded in C, compiled with `gcc 4.8.3` and run on the same platform.

Multiprecision 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 sub quadratic 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 has been presented by Mont-

gomery in [12]. More specifically, we used the block Montgomery algorithm suggested by Bosselaers *et al.* in [2]. In this algorithm, the multi-precision operations combine full size operand with one word operand and are also available in the GMP library [1]. Although the complexity is the same, the implementation is more computer friendly.

m_0m_1 Recoding: 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. Steps 9 and 21 in Algorithm 4, an inversion modulo m_0 is required ($|(k_i^{(1)})^{-1}|_{m_0}$). 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 the key sizes given page 11). The timings given in the next Section include this recoding.

Test processing : the tests involve a few hundred dataset, which consists of random exponent inputs and an exponentiation base with the precomputed values stored. 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 dataset.

5.2 Tests results and comparison

The three 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 recoding in the radix-R and m_0, m_1 cases. The implementation results confirm the complexity evaluation, for key sizes of 224, 256, 384, and 512 bits. However, the better results are for 384 and 512 bits.

In Table 4, 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 m_0, m_1 approach 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 Radix-R method, for the same computation timing, about $12.5 \times 10^6 \#CC$.

6 Conclusion and future work

In this paper, we have presented a new method for fixed-base exponentiation using a radix-R conversion with RNS representation of every radix-R digits, using an RNS base with two moduli $\mathcal{B} = \{m_0, m_1\}$. We have designed a recoding algorithm, which computes our m_0m_1 representation of the exponent, and

Table 4. Synthesis of implementation results, clock cycles and storage (kB). Test performed on an Intel XEON E5-2650 (Ivy bridge), gcc 4.8.3, CENTOS 7.0.1406.

Modular Exponentiation			
State of the Art methods			
Fixed-base Comb	radix R	m_0, m_1 rec.	ratio
$\#CC$ Storage	$\#CC$ Storage	$\#CC$ Storage	m_0, m_1 /Best S.o.A.
key size 224 bits, field size 2048 bits (level of security: 112 bits)			
221108	227838	**219864**	×0.994
1023.5 kB ($w = 12$)	829 kB ($R = 91$)	**580 kB** ($m_0 = 89, m_1 = 6$)	×0.700
210074	206888	**207072**	×0.985
2047.5 kB ($w = 13$)	1324 kB ($R = 163$)	**766 kB** ($m_0 = 127, m_1 = 7$)	×0.579
149690	147877	**146156**	×0.988
65535 kB ($w = 18$)	**7289kB** ($R = 1223$)	21599 kB ($m_0 = 5417, m_1 = 6$)	×2.96
key size 256 bits, field size 3072 bits (level of security: 128 bits)			
524539	502981	**501466**	×0.997
1535 kB ($w = 12$)	1411 kB ($R = 91$)	**897 kB** ($m_0 = 79, m_1 = 6$)	×0.636
449397	445871	**446444**	×1.001
6143 kB ($w = 14$)	2251 kB ($R = 163$)	**2056 kB** ($m_0 = 211, m_1 = 6$)	×0.913
356892	354640	354071	×0.998
98303 kB ($w = 18$)	**6414 kB** ($R = 571$)	12843 kB ($m_0 = 1721, m_1 = 7$)	×2.002
key size 384 bits, field size 7680 bits (level of security: 192 bits)			
4442590	4492191	**4409584**	×0.993
1918 kB ($w = 11$)	3430 kB ($R = 53$)	**1134 kB** ($m_0 = 23, m_1 = 10$)	×0.467
3554339	**3524896**	3551437	×1.008
15358 kB ($w = 14$)	8290 kB ($R = 163$)	**4164 kB** ($m_0 = 113, m_1 = 10$)	×0.502
2736341	**2543480**	2743399	×1.079
245758 kB ($w = 18$)	45221 kB ($R = 1223$)	**29961 kB** ($m_0 = 1031, m_1 = 7$)	×0.662
key size 512 bits, field size 15360 bits (level of security: 256 bits)			
18632429	19260731	**18550238**	×0.996
15536 kB ($w = 13$)	13765 kB ($R = 91$)	**4745 kB** ($m_0 = 41, m_1 = 10$)	×0.345
14848261	15401002	**14813453**	×0.998
122876 kB ($w = 16$)	34418 kB ($R = 163$)	**22109 kB** ($m_0 = 257, m_1 = 11$)	×0.642
12477816	**12193232**	12499600	×1.025
983036 kB ($w = 19$)	119061 kB ($R = 1223$)	**102820 kB** ($m_0 = 1381, m_1 = 7$)	×0.863

we have used it in a modular exponentiation algorithm which provides memory storage savings or improve the performance in terms of clock cycles per modular exponentiation, while offering a total flexibility in terms of storage amount. We have provided a complexity evaluation, which shows that our approach improves significantly the complexity/storage trade-off. We have then implemented this approach in order to check the performance benefits. We have compared our approach with two other classical approaches, the *Fixed-base Comb* and the Radix-R, and have confirmed the complexity results, showing the better storage/performance trade-off of our approach.

Two issues remain opened:

– Side-channel analysis is also a major threat, even in case of software implementation. For example, Gueron in [7] mentions the cache attack. In the present paper, we did not take this threat into account in the memorization process. However, by using a storage pattern spreading the data in memory, we could ensure the resistance against cache-attacks in the same way as the

one used by Gueron without penalty. This needs to be implemented in all algorithms for fair comparison.
– Our approach can be applied to Elliptic Curve Cryptography, particularly to the ECDSA signature protocol. In this case, one needs to compute Elliptic Curve Scalar Multiplication. However, the relatively cheap doubling of point operation in comparison with point addition for the NIST recommended curves makes the benefits of our approach not as good as the one in the modular exponentiation case. Therefore, this approach needs to be implemented in relevant curves. For example, the twisted Edwards curve is an example of curve with relatively equivalent doubling and addition in terms of complexity.

References

1. The GNU Multiple Precision Arithmetic Library (GMP). http://gmplib.org/.
2. Antoon Bosselaers, René Govaerts, and Joos Vandewalle. Comparison of three modular reduction functions. In *Advances in Cryptology - CRYPTO '93, Proceedings*, pages 175–186, 1993.
3. Acting Secretary Cameron Kerry and USST/Director Patrick Gallagher. Digital Signature Standard (DSS). In *Federal Information Processing Standards Publications*, volume FIPS PUB 186-4. NIST, 2013.
4. Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Advances in Cryptology, Proceedings of CRYPTO '84*, pages 10–18, 1984.
5. Harvey L. Garner. The residue number system. In *Proceedings of the Western Joint Computer Conference*, pages 146–153, 1959.
6. Daniel M. Gordon. A survey of fast exponentiation methods. *J. Algorithms*, 27(1):129–146, 1998.
7. Shay Gueron. Efficient software implementations of modular exponentiation. *J. Cryptographic Engineering*, 2(1):31–43, 2012.
8. Darrel Hankerson, Julio López Hernandez, and Alfred Menezes. Software Implementation of Elliptic Curve Cryptography over Binary Fields. In *Cryptographic Hardware and Embedded Systems - CHES 2000*, volume 1965 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2000.
9. Donald E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms, 3rd Edition*. Addison-Wesley, 1998.
10. Chae Hoon Lim and Pil Joong Lee. More flexible exponentiation with precomputation. In *Advances in Cryptology - Crypto '94*, volume 839 of *Lecture Notes in Computer Science*, pages 95–107. Springer, 1994.
11. Lourdes López-García, Luis J. Dominguez Perez, and Francisco Rodríguez-Henríquez. A pairing-based blind signature e-voting scheme. *Comput. J.*, 57(10):1460–1471, 2014.
12. Peter Montgomery. Modular Multiplication Without Trial Division. *Mathematics of Computation*, 44(170):519–521, 1985.
13. U.S.D.C. Rebecca Blank and USST/Director Patrick Gallagher. Recommendation for Key Management. In *Computer Security*, volume Part 1, Rev 3. of *NIST Special Publication 800-57*, pages 62–64. NIST, 2012.
14. Antonin Svoboda. The numerical system of residual classes in mathematical machines. In *IFIP Congress*, pages 419–421, 1959.