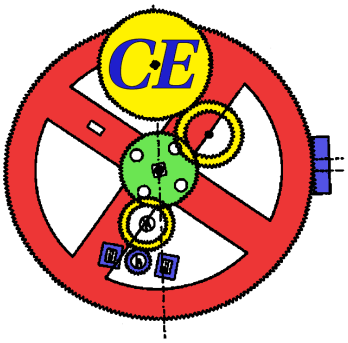


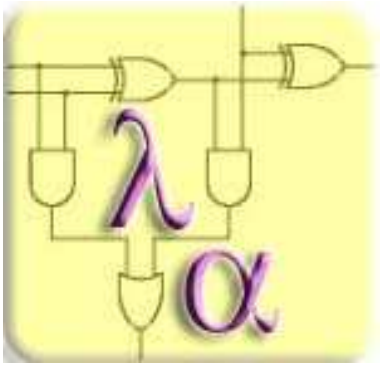
Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2010

MSc THESIS

RNS support for RSA cryptography

Nicoleta CUCU-LAURENCIU



CE-MS-2010-23

Abstract

The majority of currently established public-key cryptosystems, e.g., RSA, ECC, requires modular multiplication in finite fields as their core operation. As a result, the throughput rate of such cryptosystem depends upon the speed of modular multiplication and upon the number of performed modular multiplications. Montgomery algorithm is one method that allows efficient implementation of multiplication modulo large number, as required by the RSA cryptosystem. In the recent years, renewed interest has been payed to Residue Number Systems (RNS), due to their ability to enable parallel and fast modular arithmetic. Within RNS any integer is represented with a set of its residues with respect to a given base that comprises a set of relatively prime integers. In this way RNS distributes large dynamic range computations over small modular rings. Thus the computations are carried out independently in each of the small wordwidth RNS channels. Since the RNS is particularly suited for performing efficient long integer modular arithmetic, the Montgomery algorithm was adapted such that it can be utilized in conjunction with RNS. RNS Montgomery modular multiplication makes use of two representation bases B and B', and during the calculations requires two base extension operations. Such a base extension involve the calculation of the residue digits with respect to B'/B, when the digits relative to B/B' are known, and dominates the RNS Montgomery modular multiplication computational complexity. This thesis evaluates existing base extensions methods and proposes a new improved approach, which makes use of the linear Diophantine equations theory. Assuming that B and B' are k-moduli sets, to derive the value of k new residues, our method requires k^2 regular multiplications and k modular multiplications, while equivalent state-of-the-art methods require, depending on the extension sense, B to B' or B' to B, k^2+k and k^2+2k modular multiplications, respectively. When utilized in the RSA context, our method provides a speedup of $O(\mu)$ relative to the stateof-the art, where μ is the ratio between the computation time required by a modular multiplication and by a regular one. To better asses the practical implications of the proposed method we implemented RSA based on state of the art and on the Diophantine theory and compared their performance. For the evaluations we assumed various RSA keysizes, $e = 2^{16} + 1 = 65537$, different RNS moduli sets with varying cardinality (k=4, 5, and 6), bitlength and Hamming weight, and various messages to encrypt. Our experimental results indicate that for sets of 4, 5, and 6 moduli with bitlength of 512-bits, our method provides a speedup per Montgomery kernel of 1.93, 2.42, and 3.17, respectively.

TU Delft
Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

RNS support for RSA cryptography

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Nicoleta CUCU-LAURENCIU
born in Brasov, Romania

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

RNS support for RSA cryptography

by Nicoleta CUCU-LAURENCIU

Abstract

The majority of currently established public-key cryptosystems, e.g., RSA, ECC, requires modular multiplication in finite fields as their core operation. As a result, the throughput rate of such cryptosystem depends upon the speed of modular multiplication and upon the number of performed modular multiplications. Montgomery algorithm is one method that allows efficient implementation of multiplication modulo large number, as required by the RSA cryptosystem. In the recent years, renewed interest has been payed to Residue Number Systems (RNS), due to their ability to enable parallel and fast modular arithmetic. Within RNS any integer is represented with a set of its residues with respect to a given base that comprises a set of relatively prime integers. In this way RNS distributes large dynamic range computations over small modular rings. Thus the computations are carried out independently in each of the small wordwidth RNS channels. Since the RNS is particularly suited for performing efficient long integer modular arithmetic, the Montgomery algorithm was adapted such that it can be utilized in conjunction with RNS. RNS Montgomery modular multiplication makes use of two representation bases B and B', and during the calculations requires two base extension operations. Such a base extension involve the calculation of the residue digits with respect to B'/B, when the digits relative to B/B' are known, and dominates the RNS Montgomery modular multiplication computational complexity. This thesis evaluates existing base extensions methods and proposes a new improved approach, which makes use of the linear Diophantine equations theory. Assuming that B and B' are k-moduli sets, to derive the value of k new residues, our method requires k^2 regular multiplications and k modular multiplications, while equivalent state-of-the-art methods require, depending on the extension sense, B to B' or B' to B, $k^2 + k$ and $k^2 + 2k$ modular multiplications, respectively. When utilized in the RSA context, our method provides a speedup of $O(\mu)$ relative to the stateof-the art, where μ is the ratio between the computation time required by a modular multiplication and by a regular one. To better asses the practical implications of the proposed method we implemented RSA based on state of the art and on the Diophantine theory and compared their performance. For the evaluations we assumed various RSA keysizes, $e = 2^{16}+1 = 65537$, different RNS moduli sets with varying cardinality (k=4, 5, and 6), bitlength and Hamming weight, and various messages to encrypt. Our experimental results indicate that for sets of 4, 5, and 6 moduli with bitlength of 512-bits, our method provides a speedup per Montgomery kernel of 1.93, 2.42, and 3.17, respectively.

Laboratory : Computer Engineering
Codenumber : CE-MS-2010-23

Committee Members :

Advisor: Sorin Cotofana, CE, TU Delft

Chairperson:	Koen Bertels, CE, TU Delft
Member:	Koen Bertels, CE, TU Delft
Member:	Fernando Kuipers, NAS, TU Delft
Member:	Stephan Wong, CE, TU Delft

To my mentor

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my supervisor, prof. Sorin Cotofana, for his continuous guidance and support throughout the thesis. I was lucky and honoured to work under his coordination.

Nicoleta CUCU-LAURENCIU
Delft, The Netherlands
November 22, 2010

1 Introduction

1.1 Motivation

In this age of universal electronic connectivity, of electronic eavesdropping and fraud, it is of utmost importance to store information securely. This led to a heightened awareness to protect the data from disclosure, to guarantee the authenticity of data and messages, and to protect systems from network-based attacks. Cryptography plays a major role in mobile phone communications, e-commerce, pay-tv, sending private emails, transmitting financial information, security of ATM cards, computer passwords, electronic commerce digital signature and so on. Modular exponentiation $a^b \bmod m$, and implicitly modular multiplication $a \cdot c \bmod m$ are the operations intensively used, underlying many cryptographic schemes. In this thesis, we focus only on the Rivest Shamir and Adleman (RSA) [26] cryptosytem requirements: the execution of multiplications modulo a large number, chosen in the order of 512-2048 bits to safeguard the information, at high throughput rates.

One possible way to speed up multiplications modulo a large number is to rely on Residue Number Systems (RNS) to represent the operands. RNS has been an important field of research in computer arithmetic, due to its great potential for accelerating arithmetic computations, by breaking the arithmetic on large numbers to arithmetic on a set of smaller numbers. Thus, the carry-free and parallel nature of residue arithmetic makes RNS a powerful candidate for fast solutions to long integer arithmetic. However, applying the RNS to the long integer modular multiplication problem is not straightforward. The main difficulty is induced by the fact that the modulus used in RSA cryptosystem is a product of two prime numbers, which precludes coincidence with the dynamic range of a many moduli RNS base.

Montgomery algorithm [18] is an effective way to perform modular multiplication, while avoiding division by the large modulus. The majority of existing RSA algorithms use Montgomery method and high radix number systems. Montgomery algorithm has been also combined with RNS, which proves to be a promising alternative. The first attempts were made in 1995 by Posch et al. [25].

For reducing an integer $a \cdot b$ modulo m, the Montgomery method computes a multiple qm of m that need to be added to $a \cdot b$ such that the least significant part of $u = a \cdot b + qm$ is 0. This value can now be safely divided by $r = 2^n$, which reduces to a simple shift for binary numbers. The obtained result is equivalent to $a \cdot b \cdot r^{-1} \bmod m$. Since the division step is exact, in RNS the division is equivalent to multiplication by the divisor multiplicative inverse. However this multiplicative inverse does not exist in the first RNS base and thus a second RNS base has to be introduced. Therefore, at the begining one has to compute the product $a \cdot b$ in the first RNS base, then extend the result to a second RNS base and compute the value of u, and afterwards convert the result back to the

first RNS base. Most of the computational effort of the RNS Montgomery multiplication algorithm is required by the two necessary base extensions.

The aim of this thesis is to investigate alternative base extension methods that have lower computational demands such that the Montgomery modular multiplication can be executed faster and the overall RSA performance (throughput) is improved. One could reformulate this as follows: given the RNS representation of an integer X in a base B, it is required to compute the RNS representation of that integer with respect to another RNS base B' while minimizing the number of operations necessary to be performed.

1.2 Thesis Contribution

This section presents the main contributions of the thesis, which consist of:

1. We performed a thorough study of existing base extension methods. The methods are presented and explained and their performances are compared in the context of RNS Montgomery multiplication.

2. We proposed a novel base extension method based on the theory of linear Diophantine equations. To perform a base extension with respect to k new moduli, our method requires k^2 regular multiplications and k modular multiplications. The state-of-the-art methods [7] and [28] (Bajard and Shenoy) need $k^2 + k$ and $k^2 + 2k$ modular multiplications, respectively. Thus our method improves the overall performance by reducing the number of multiplications in general and by replacing most of the required modular multiplications with normal multiplications, which are faster than the modular ones. For the entire RNS Montgomery multiplication, our method requires $7k$ modular multiplications and $2k^2$ regular multiplications, while the state of the art method requires $2k^2 + 8k + 3$ modular multiplications.

3. We evaluated the practical implications of the prosed method by implementing the RSA encryption-decryption protocol in C++, for different RSA key sizes (512-bit, 1024-bit). Two versions of the Montgomey algorithm were considered: the former using the proposed Diophantine base extension method, and the latter using the state-of-the-art method (Bajard and Shenoy). The performance was assesed for various RNS base moduli (several sets with cardinality 4, 5, 6; RNS moduli with different bitlength and varying Hamming weight). The public exponent e was chosen 65537, which is usually a common choice for RSA implementations. Our experimental results indicate that for sets of 4, 5, and 6, 512-bit moduli, our method provides a speedup per Montgomery kernel of 1.93, 2.42, and 3.17, respectively.

1.3 Thesis Outline

This thesis is organized in 6 chapters and one appendix, as follows:

Chapter 2 introduces the fundamental mathematical results necessary for RNS. First, the basic principles of the RNS representation, with a numerical example, are explained.

Next, a basic algebra of residues is presented and the chapter concludes with the presentation of one of the most important results in the theory of RNS, the Chinese Remainder Theorem (CRT).

Chapter 3 starts by giving an overview of the RSA cryptosystem. Next, modular exponentiation and modular multiplication are briefly debated. Montgomery modular multiplication technique as a fast alternative to classical modular multiplication is introduced and its adaptation to the RNS system is discussed. The final of the chaper is dedicated to considerations regarding the selection of the moduli set, with its impact on the overall system performance.

Chapter 4 gives a survey of the previous existing base extension methods, and proposes a new approach for performing base extension. The methods are put in the context of the RNS Montgomery multiplication algorithm and their performance is analyzed.

Chapter 5, reports the simulation results. The simulation setup is presented, followed by a comparison analysis of the execution time of our work and of previous state-of-the-art implementation.

Finally, the thesis is concluded in Chapter 6.

The Appendix comprises the description of 3 auxiliary algorithms, referenced throughout the thesis.

2 RNS Preliminaries

This chapter introduces the basic notions and several important properties of Residue Number Systems (RNS) that are foundational for the succeeding chapters.

2.1 Introduction

The binary 2's complement is the conventional number system used for contemporary computer arithmetic. It is a weighted number system, with the major advantage that addition and subtraction can be easily performed ($-x = \bar{x} + 1$), allowing the use of the same circuitry for both operations. However, traditional binary 2's complement arithmetic suffer from long carry propagation delay, which imposes a limitation on the achievable performance. There exist two major approaches for dealing with the carry problem: (i) speedup the carry propagation via fast algorithms and (ii) use different number representations to limit the carry propagation to within a smaller number of bits or completely eliminate it. Specific to the first approach, a variety of adders such as carry-lookahead, conditional-sum, or multilevel carry select have been developed, exhibiting a time complexity of $O(log\, n)$ at best, for n-bit numbers. Alternatively, different representations were explored, such as redundant representations or RNS.

The ancient study of RNS, begins 1700 years ago with a verse, Suan-ching, by Sun Tzu: *"We have things of which we do not know the number, If we count them by threes, the remainder is 2, If we count them by fives, the remainder is 3, If we count them by sevens, the remainder is 2, How many things are there? The answer, 23."*. The theory of residue numbers was first adapted to computer arithmetic in 1967 by Szabo and Tanaka, in [30]. RNS is a nonweighted number system, in which each number is represented as a tuple of residues relative to a set of relatively prime moduli. RNS aims to increase the speed of computations using a divide-et-impera approach: it distributes large dynamic range modular computations over smaller but independent channels without inter-channel carry dependencies, over which computations can be performed in parallel. However, since it is a nonweighted number system, the magnitude related operations (i.e., division, sign detection, magnitude comparison, etc.) are more difficult to perform. In RNS, additions can be peformed in $O(log\, log\, n)$ with unrestricted moduli and in $O(log\, n)$ with restricted moduli. For addition and multiplication dominated applications and for arithmetic on long operands, as it is the case in RSA, RNS can be of interest.

2.2 RNS Representation

A residue number system is completely specified by stating its base, $B = (m_1, m_2, ..., m_k)$, which is a set of positive integers called moduli. In what follows, we shall assume that

we are dealing with RNS bases comprising moduli that are pairwise relatively prime to each other, i.e., $gcd(m_i, m_j) = 1, \forall i \neq j$.

For a given base B, the residue representation of an integer X is denoted by a k-tuple of integers

$$\langle X \rangle_B = (x_1, x_2, ..., x_k) \qquad x_i = |X|_{m_i},$$

where x_i represents the remainder of the division of X by the modulus m_i and is designated as "the residue of X modulo m_i".

Example 2.1

$$\langle 5 \rangle_B = (5, 5, 0, 2), \qquad \textit{where } B = (8, 7, 5, 3)$$

The dynamic range of the RNS system is represented by the product of all k relatively prime moduli. It reflects the number of different representable values in the defined RNS system.

$$M = m_1 \cdot m_2 \cdot ... \cdot m_k$$

We note that if the moduli are not pairwise relatively prime, then representations in such a system are not unique, such that two or more numbers may have the same representation.
The dynamic range M can be used to represent numbers in the $[0, M-1]$ set, or any other interval of M consecutive integers. For instance if a symmetric RNS is desired, the dynamic range of the system is chosen as $[-(M-1)/2, (M-1)/2]$ for M odd, and $[-M/2, (M/2)-1]$ for M even. Thus, residues of negative numbers are evaluated by complementing each of the digits x_i with respect to its correspondent modulus m_i (0 digits are left unchaged)

$$|-X|_{m_i} = |M - X|_{m_i}.$$

Example 2.2 *Given that* $\langle 21 \rangle_B = (5, 0, 1, 0)$ *with respect to the moduli* $(8, 7, 5, 3)$*, we obtain:*

$$\langle -21 \rangle_B = (8-5, 0, 5-1, 0) = (3, 0, 4, 0)$$

Unless otherwise stated, in this thesis we will assume nominal ranges of $[0, M-1]$.

2.3 The Algebra of Residues

The following relations form the basis for addition, subtraction or multiplication modulo m:

$$\begin{aligned} |a \pm b|_m &= ||a|_m \pm |b|_m|_m = ||a|_m \pm b|_m \\ |a \cdot b|_m &= ||a|_m \cdot |b|_m|_m = ||a|_m \cdot b|_m \end{aligned} \tag{2.1}$$

The additive inverse of a residue $|a|_m$, denoted by $|-a|_m$ is defined as:

$$|a + (-a)|_m = 0.$$

Every number has a unique additive inverse with respect to a modulus m and can be obtained through the following operation:

$$|-a|_m = |m - a|_m .$$

The additive inverse can be applied to individual residues or to the RNS number as an entire.

The multiplicative inverse of a mod m, denoted by $\left|a^{-1}\right|_m$, is a number $\in [0, m)$ such that the following relation holds true:

$$\left|a \cdot a^{-1}\right|_m = 1. \tag{2.2}$$

While every residue has an additive inverse, this is not the case for multiplication. Some residues may not have multiplicative inverses. However, if and only if $gcd(a, m) = 1$ and $|a|_m \neq 0$, then the multiplicative inverse exists and it is unique.

Example 2.3 *Given $a = 7$, we want to compute the multiplicative inverse of a with respect to the modulus* 11*.*

$$\left|7 \cdot a^{-1}\right|_{11} = 1 \qquad \Rightarrow \qquad \left|a^{-1}\right|_{11} = 8.$$

There exists no general rule for finding the multiplicative inverse of a certain number; a brute-force search is about the best possible approach. Nevertheless, if the modulus m is a prime number then Fermat's Theorem can be applied for finding the multiplicative inverse.

Theorem 2.1 (Fermat's Theorem) *Provided m is a prime modulus and $gcd(a, m) = 1$, we have*

$$|a^m|_m = |a|_m .$$

From this theorem, it may be deduced that

$$\left|a^{-1}\right|_m = \left|a^{m-2}\right|_m .$$

All the above relations were defined for individual moduli. In what follows addition, subtraction, and multiplication will be derived for RNS numbers. We denote by $\mathbb{Z}$ the ring of integers; $\mathbb{Z}_M$ is the residue ring $\mathbb{Z}/M\mathbb{Z}$ and its complete residue class is $\{0, 1, ..., M - 1\}$. Addition, subtraction, and multiplication are closed operations in RNS over $\mathbb{Z}/M\mathbb{Z}$. Let $\otimes$ represent the binary operations of $\{+, -, \times\}$. Therefore, we obtain that

$$X \otimes Y = (|x_1 \otimes y_1|_{m_1}, ..., |x_k \otimes y_k|_{m_k}) \tag{2.3}$$

The isomorphism between the ring $\mathbb{Z}/M\mathbb{Z}$ and the direct sum ring $\mathbb{Z}/m_k\mathbb{Z}$ implies that the computations can be carried out over small finite fields, instead of the original ring. There exists no intermodular carries, each residue digit of the result being a function of only the corresponding residue digits of the operands, which gives an inherent speed advantage to the RNS.

Example 2.4 *For the moduli $B = (3, 4, 5, 11)$ we want to add X=102 and Y=211.*

$$\begin{array}{l} \langle 102 \rangle_B = (0, 2, 2, 3) + \\ \langle 211 \rangle_B = (1, 3, 1, 2) \\ \hline |313|_{660} = (|1|_4, |5|_3, |3|_5, |5|_{11}) = (1, 1, 3, 5). \end{array}$$

The ability of performing fast, independent, and parallel computations within each small wordwidth RNS channel represents the main advantage of using RNS over other conventional weighted number system, e.g., binary. However, additional conversions of the inputs and the result, between standard binary format and RNS are required. Eventhough their performance have a great impact on hardware complexity, latency, and power consumption, for data-intensive calculations the savings achieved with the internal RNS computations can easily counterbalance the conversions overhead.

Unfortunately, what is gained in terms of speed and simplicity for addition, subtraction or multiplication, might be nullified by the complexity of division, and the difficulty of certain auxiliary operations, such as sign test, magnitude comparison, and overflow detection. Nevertheless, there are special cases of the division that make possible the use of simpler algorithms. One of these cases is when the dividend is known to be a multiple of the divisor. Then, the residue division and the normal division will correspond. Conventional division can be represented by the equation:

$$\frac{x}{y} = q \qquad \Leftrightarrow \qquad y \cdot q = x.$$

Expressed in RNS the previous relation becomes:

$$|y \cdot q|_m = |x|_m .$$

Provided y and the modulus m are relatively prime, if we multiply both sides by the multiplicative inverse of y with respect to m we obtain:

$$|q|_m = \left|x \cdot y^{-1}\right|_m .$$

Therefore, contrary to the corresponding situation in conventional arithmetic, in RNS multiplication by the multiplicative inverse is not always equivalent to division: this equivalence holds true only when the quotient is an integer value.

Example 2.5 *Assuming $m = 5$, we compute the following quotients:*

$$q = \frac{4}{2} \qquad \begin{aligned} |2q|_5 &= |4|_5 \\ |q|_5 &= \left|4 \cdot 2^{-1}\right|_5 \\ |q|_5 &= |4 \cdot 3|_5 \\ |q|_5 &= 2. \end{aligned}$$

Now we consider the case when the remainder of the division is not 0 *anymore:*

$$q = \frac{4}{3} \qquad \begin{aligned} |3q|_5 &= |4|_5 \\ |q|_5 &= \left|4 \cdot 3^{-1}\right|_5 \\ |q|_5 &= |4 \cdot 2|_5 \\ |q|_5 &= 3. \end{aligned}$$

One method extensively used in cryptography, that exploits the integer division case is the Montgomery multiplication technique and it is studied in detail in Chapter 3.

2.4 Chinese Remainder Theorem

One of the most important results in the theory of residue number systems is the Chinese Remainder Theorem, which allows the computation of the equivalent binary/decimal number of an RNS representation.

Theorem 2.2 (Chinese Remainder Theorem) *Under the assumption that $(m_1, ..., m_k)$ are pairwise relatively prime moduli, then there exists exactly one integer that satisfies the conditions:*

$$\begin{aligned} &c \leq X < c + M \\ &x_i = |X|_{m_i} \qquad\qquad i = 1, .., k, \end{aligned}$$

where $M = \prod_{i=1}^{k} m_i$ and c is an integer.

It assures us that for relatively prime moduli, each number in the dynamic range has a unique representation in the RNS, and that we can recover the number represented from such a representation.

An alternate way of stating the CRT, that allows us to recover the represented integer from its residues, is as follows:

Given the residues x_i and the pairwise prime moduli m_i, where $i = 1, ..., k$, the magnitude of the represented integer X can be obtained by using the subsequent relation:

$$|X|_M = \left| \sum_{i=1}^{k} M_i \cdot x_i \cdot \left| M_i^{-1} \right|_{m_i} \right|_M , \tag{2.4}$$

where $M = \prod_{i=1}^{k} m_i$, $M_i = \frac{M}{m_i}$ and $\left| M_i^{-1} \right|_{m_i}$ is the multiplicative inverse of M_i with respect to the modulus m_i as defined in Equation (2.2).

Since the sum may exceed M, a reduction modulo M is applied in order to yield the result within the dynamic range of the given residue number system. If it is known that X lies in the interval $[0, M-1]$, then the modular reduction on the left-hand side of the equation may be ommited.

In some cases, for instance when the storage or multiplier requirements need to be reduced, the CRT is reformulated as:

$$\begin{aligned} X &= \left| \sum_{i=1}^{k} M_i \cdot x_i \cdot \left| M_i^{-1} \right|_{m_i} \right|_M \\ &= \left| \sum_{i=1}^{k} M_i \cdot |x_i|_{m_i} \cdot \left| M_i^{-1} \right|_{m_i} \right|_M \\ &= \left| \sum_{i=1}^{k} M_i \cdot \left| x_i \cdot \left| M_i^{-1} \right|_{m_i} \right|_{m_i} \right|_M . \end{aligned} \tag{2.5}$$

In other cases, it is desirable to express the CRT equation without the modulo M operation. This is achieved by rewriting X according to the Fundamental Theorem of the Remainder. The quotient of the integer division is denoted by α_1 when the remainder theorem is applied to Equation (2.4), respectively by α_2 when applied to Equation (2.5).

$$\sum_{i=1}^{k} M_i \cdot x_i \cdot \left| M_i^{-1} \right|_{m_i} = X + \alpha_1 \cdot M \tag{2.6a}$$

$$\sum_{i=1}^{k} M_i \cdot \left| x_i \cdot \left| M_i^{-1} \right|_{m_i} \right|_{m_i} = X + \alpha_2 \cdot M \tag{2.6b}$$

Note that α_2 value is upper bounded by the number of moduli k, as indicated by Equation 2.7.

$$\sum_{i=1}^{k} M_i \cdot \left| x_i \cdot \left| M_i^{-1} \right|_{m_i} \right|_{m_i} < \sum_{i=1}^{k} \frac{M}{m_i} \cdot m_i = \sum_{i=1}^{k} M = k \cdot M \tag{2.7}$$

Thus $\alpha_2 \in [0, k-1]$. This observation proves to be crucial for some base extension methods presented in Chapter 4.

2.5 Conclusion

This chapter presented an introduction to RNS. It covered preliminary concepts, along with definitions and theorems that form the foundations of residue number systems. We have seen that RNS is best suited for applications that require several additions and multiplications, but relatively few conversions or magnitude related operations. Such applications are typical for public-key RSA cryptosystems for instance, in which modular exponentiation (essentially, sequences of modular multiplications) is the most significant operation, accounting for most of the time spent for encryption and decryption. In the next chapter we present RSA cryptosystems while giving special attention to fast modular multiplication/exponentiation methods for wide operands.

Modular Arithmetic in RSA Cryptography

3

Faster implementations of public-key cryptography, and in particular of RSA are of utmost importance nowadays. Performing fast modular multiplication for large integers is of special interest because it provides the basis for performing fast modular exponentiation, which is the key operation of the RSA cryptosystem. Currently, it seems that in a radix representation, all major performance improvements have been achieved. Nevertheless, the use of RNS proves to be a promising alternative for achieving a breakthrough. All these aspects are detailed throughout this chapter.

3.1 The RSA cryptosystem

This section presents an overview of the RSA cryptosystem, followed by a short proof of why the encryption-decryption mechanism works. The section concludes with considerations regarding the employed key-sizes and with an example of a small RSA cryptosystem.

3.1.1 Introduction

Generally speaking, cryptography falls into two main categories: secret and public key cryptography.

Secret-key cryptography is based on a prior exchange of a common secret key. Since a single key is used for both encryption and decryption, the major issue associated with symmetric-key systems is the key distribution problem, that is an efficient method has to be devised for the parties to agree upon and then exchange keys securely. In 1970, W. Diffie and M. E. Hellman proposed in [10] an efficient method of exchanging a shared secret key over an unsecured communications channel and thus setting up the basis of a new type of cryptography: the public-key cryptography. The assymetric-key cryptography uses a key (public) for encryption, which is made available to everyone at the sending end, and another one (secret) for decryption, that is known only by the recipient of the message.

In 1977, R. Rivest, A. Shamir, and L. Adleman introduced the RSA cryptosystem [26], which became the most widely used public-key cyptosystem in the world. Its security depends upon the intractability of the integer factorization problem and it can be used to provide both data encryption and digital signatures. In what follows the RSA encryption-decryption protocol is shortly reviewed.

3.1.2 RSA algorithm

Prior to the execution of the encryption-decryption protocol, outlined in Algorithm 2, each party that wants to communicate should generate first its own public/private key pair, as described in Algorithm 1.

Algorithm 1 Public key generation

Ouput: a public key (n, e) and a private key d.

1. Generate randomly two large primes p and q, which are kept secret.
2. Compute the modulus $n = p \cdot q$ and Euler's totient function $\Phi = (p-1)(q-1)$.
3. Select a random integer e, $1 < e < \Phi$, coprime with Φ.
4. Compute the multiplicative inverse of e with respect to modulus Φ ($d \cdot e \equiv 1\,(mod\,\Phi)$) (e.g., by using the Extended Euclidian algorithm, described in Appendix A.3).

Algorithm 2 RSA encyption-decryption protocol

Bob encrypts a message m and sends it to Alice; Alice decrypts the message

1. Encryption
   a. Bob should obtain the public key (n, e) of Alice.
   b. Bob represents the message m as an integer between 0 and $n-1$.
   c. Bob computes $c = m^e \bmod n$.
   d. Bob sends the cyphertext to Alice.
2. Decryption
   a. Alice should use its private key d to recover the message m form the cyphertext $m = c^d \bmod n$.

3.1.3 RSA proof

The RSA encryption system is based on Euler's theorem and its generalization, the Carmichael's theorem.

Theorem 3.1 (Euler's Theorem) *If n and a are two positive, relatively prime integers, then it holds*

$$\left|a^{\Phi(n)}\right|_n = 1,$$

where $\Phi(n)$ is the Euler's totient function (the number of integers less than n, and relatively prime with n).

Theorem 3.2 (Carmichael's theorem) *If n and a are two positive, relatively prime integers, then*

$$\left|a^{\lambda(n)}\right|_n = 1,$$

where $\lambda(n)$ is the Carmichael function (the least common multiple of the factors of $\Phi(n)$).

Theorem 3.3 *If n is a product of distinct primes, then, for all integers a*

$$\left|a^{\lambda(n)+1}\right|_n = |a|_n .$$

The corectness of the RSA scheme, i.e., the fact that the encryption and decryption are inverse operations, relies on the fact that

$$\left|m^{e\cdot d}\right|_n = m, \qquad \text{for } m \in [0, n-1].$$

There are two cases to consider.
Case 1. $gcd(m, n) = 1$
We have $|d \cdot e|_{\Phi(n)} = 1$, relation which rewritten for an integer $\alpha \geq 1$ becomes

$$d \cdot e = \alpha \cdot \Phi(n) + 1$$

$$\begin{aligned}
\left|c^d\right|_n &= \left|(m^e)^d\right|_n \\
&= \left|m^{1+\alpha\cdot\Phi(n)}\right|_n \\
&= \left|m \cdot (m^{\Phi(n)})^\alpha\right|_n \\
&= |m \cdot 1|_n \\
&= |m|_n .
\end{aligned}$$

Case 2. $gcd(m, n) > 1$
For n, a product of two odd distinct primes,

$$\lambda(n) = \frac{\Phi(n)}{gcd(p-1, q-1)}$$

will always be a divisor of $\Phi(n)$. Since $\lambda(n)|\Phi(n)$, the equality $|e \cdot d|_{\Phi(n)} = 1$ implies that $|e \cdot d|_{\lambda(n)} = 1$. Using a derivation similar to case 1, and based on Theorem 3.3, we obtain

$$\left|m^{d\cdot e}\right|_n = |m|_n ,$$

which concludes our explanation.

3.1.4 RSA key sizes

As far as the operands sizes are concerned, the following remarks can be made.

The stochastic primes p and q should be chosen such that they have approximately the same bit length to ensure that any attempts to factor the modulus are computationally infeasible. For instance, for 1024-bit modulus n, p and q should be chosen about 512-bits each.

The exponent e is usually chosen small and preferably with a small Hamming weight (the number of 1's in its binary representation), in order to increase the efficiency of the exponentiation (for the square-and-add exponentiation algorithm introduced in the next section, a small Hamming weight reduces the number of modular multiplications). One exponent currently used in practice is $e = 2^{16} + 1 = 65537$. The exponentiation algorithm would require in this case 16 modular squarings and 2 modular multiplications (since the Hamming weight is 2).

For security reasons, the bit length of the modulus n is typically in the range 512-2048 bits or even more, and thus efficient long integer modular arithmetic is required for achieving high troughput rates at these bit precisions.

A small RSA cryptosystem example is illustrated subsequently.

Example 3.1 *We presume that Alice wants to make a public key, and that Bob wants to use that key to encrypt the plaintext* 9726 *and send it to Alice. Alice chooses the primes* $p = 101$ *and* $q = 113$ *and computes*

$$n = p \cdot q = 101 \cdot 113 = 11413$$
$$\Phi(n) = (p-1) \cdot (q-1) = 100 \cdot 112 = 11200$$

Next, the public exponent is selected such that it is a small integer coprime with $\Phi(n)$*. For instance,* $e = 3533$ *would satisfy this constraint.*
The secret decryption exponent d *is computed by*

$$\begin{aligned} d &= \left|e^{-1}\right|_{\Phi(n)} \\ &= \left|3533^{-1}\right|_{11200} \\ &= 6597. \end{aligned}$$

Alice publishes the pair $(11413, 3533)$*, respectively the modulus* n *and the public exponent* e*. Bob obtains the public key of Alice, computes the cyphertext*

$$c = m^e \ mod\ n = \left|9726^{3533}\right|_{11413} = 5761$$

and sends it over the channel. When Alice receives this cyphertext, she uses her secret decryption exponent to recover the message

$$m = c^d \ mod\ n = \left|5761^{6597}\right|_{11413} = 9726.$$

3.2 Modular Exponentiation

Modular exponentiation ($a^b \bmod m$) and its key constituent operation, modular multiplication ($a \cdot b \bmod m$), are the fundamental operations underlying cryptographic algorithms. Since modular multiplications account for most of the time spent for encryption and decryption, their optimization is crucial. This can be achieved either by reducing the number of modular multiplications or by reducing the latency of each modular multiplication.

Several algorithms have been reported in the literature for performing efficiently modular exponentiation. This section covers modular exponentiation using arithmetic in RNS.

3.2.1 Introduction

Assuming m and e have a bitlength of 1024 each, $c = m^e$ would require a total number of

$$log_2(m^e) = e \cdot log_2 m \approx 2^{1024} \cdot 1024 = 2^{1034}$$

bits in order to store its value. Therefore c cannot be obtained by performing first the modular exponentiation m^e and only after that the reduction modulo n. Thus these operations have to be interleaved at each step.

A straighthforwards way of performing exponentiation is

$$m \xrightarrow{SQ} m^2 \xrightarrow{MUL} m^3 \xrightarrow{MUL} m^4 \xrightarrow{MUL} \dots$$

However, this naive approach requires $e - 1$ modular multiplications, which would be infeasible for large exponents.

Taking into consideration that not all powers of m need to be computed in order to obtain m^e, a faster method would be:

$$m \xrightarrow{SQ} m^2 \xrightarrow{MUL} m^3 \xrightarrow{SQ} m^6 \xrightarrow{MUL} m^7 \dots$$

This method is called the square-and-multiply algorithm and is detailed in Subsection 3.2.2. The algorithm provides a systematic way for finding the exact sequence in which squarings and multiplications by m have to be performed in order to efficiently compute m^e.

3.2.2 Binary Exponentiation Method

The method is based on scanning bit-by-bit the exponent. At each step (i.e., for every scanned bit) a squaring is performed. If and only if the currently scanned exponent bit is 1, then a subsequent multiplication is performed. Depending on the direction of processing the exponent bits (i.e., from MSB to LSB, or vice-versa), there exist two versions of the algorithm: the left-to-right binary method which is described below and the right-to-left binary method that is similar but requires an extra variable to keep the powers of m.

Let $k = \lfloor log_2 e \rfloor + 1$ denote the bitlength of the exponent e whose binary expansion is

$$e = (e_{k-1}e_{k-2} \dots e_1 e_0) = \sum_{i=0}^{k-1} e_i \cdot 2^i.$$

The left-to-right binary exponentiation algorithm computes the exponentiation starting from the most significant bit position of the exponent E and proceeding to the right, as described in Algorithm 3.

Example 3.2 *Let $e = 26 = (11010)_2$ and $k = 5$. We want to compute m^e using the square-and-multiply method.*

Since $e_{k-1} = e_4 = 1$, in the initialization step 1. we take $c := m$. Then, the algorithm proceeds as follows:

		SQ	*MUL*
$e_3 = 1$	$i = 3$	$(m)^2 = m^2$	$m^2 \cdot m = m^3$
$e_2 = 0$	$i = 2$	$(m^3)^2 = m^6$	-
$e_1 = 1$	$i = 1$	$(m^6)^2 = m^{12}$	$m^{12} \cdot m = m^{13}$
$e_0 = 0$	$i = 0$	$(m)^{13} = m^{26}$	-

Algorithm 3 Left-to-right binary exponentiation algorithm

Input: m, e, n
Output: $c = |m^e|_n$

1. **if** $e_{k-1} = 1$ **then** $c := m$ else $c := 1$
2. **for** $i = k - 2$ **downto** 0
   2a. c:=$|c \cdot c|_n$
   2b. **if** $e_i = 1$ **then** $c := |c \cdot m|_n$
3. **return** c

Assuming $e_{k-1} = 1$, the algorithm requires $k - 1$ squarings (step 2a.) and $H(e) - 1$ multiplications (step 2b.), where $H(e)$ is the Hamming weight of the exponent (the number of ones in its binary representation). Since $0 \leq H(e) - 1 \leq k - 1$, we have a total maximum number of multiplications of $2 \cdot (k-1)$, a minimum of $k-1$, while in the average case ($H(e) = 0.5 \cdot k$, that is half of the bits of e are 1), $1.5 \cdot (k-1)$ multiplications are needed. For instance, for 1024-bit exponents, the square-and-add algorithm has a logarithmic computational complexity, requiring on average only $1.5 * 1023 = 1535$ multiplications, while the straightforward exponentiation needs a linear amount of 2^{1024} multiplications.
The binary method is used frequently in smart cards and embedded devices, due to its simplicity and low memory requirements.

This method can be generalized by scanning multiple bits of the exponent at a time. Generally, if $log_2 m$ bits are scanned, the method is called m-ary [14]. When compared to the binary method, it requires fewer iterations (clock cycles), but at the expense of higher memory resources. Usually, this method is used for software implementations on processors which have access to bigger memory resources.

3.3 Modular Multiplication

The modular multiplication operation may be decomposed in two parts: a normal multiplication followed by a reduction. In its simple form, modular reduction requires trial division for finding the multiple of the modulus that has to be subtracted from the result and thus it is inherently slow. For this reason, faster alternative algorithms are utilized (Fast-Fourier Transforms [27], Karatsuba-Ofman algorithm [15], [32], Barret reduction and Quisquater's modification [12], redundant-digit division [31], etc.). Since presenting

a survey of these algorithms is out of the scope of this thesis, we just focus on the one of interest in the context of this thesis, the Montgomery multiplication.

3.4 Montgomery Multiplication

One method that allows efficient implementation of modular multiplication is the Montgomery algorithm, introduced in 1985 by P. L. Montgomery [18]. The algorithm computes $c = |a \cdot b|_n$, without performing the division by n. Using a specific representation of the operands, that is the n-residue notation, the costly division by modulus n is replaced with division by a power of 2, which reduces to simple shifts for numbers represented in binary form.

In what follows we present the main idea behind the Montgomery reduction algorithm. The Mongomery reduction of $|a \cdot b|_n$ with respect to r is defined as

$$MontMult(a, b) = \left|a \cdot b \cdot r^{-1}\right|_n ,$$

where a, b, and n are k-bit numbers, and r^{-1} is the multiplicative inverse of r with respect to the modulus n, as defined in Equation (2.2). The radix r is usually chosen as two to the power of a multiple of the machine wordsize: $0 \leq r = 2^{k \cdot w} < n$. The algorithm requires that n and r are coprime ($gcd(r, n) = 1$). Since r is a power of 2, it implies that n should be taken odd. This requirement is easily satisfied in the case of RSA criptography, where the modulus n is a product of two primes.

Before the algorithm can be started, the input operands have to be converted to their n-residue representation. The n-residue of an integer $a < n$ with respect to r is defined as

$$\bar{a} = |a \cdot r|_n , \qquad \text{where } a < n. \tag{3.1}$$

The Montgomery multiplication algorithm computes the n-residue of the product of two integers whose n-residues are known as follows:

$$MontMult(\bar{a}, \bar{b}) = \bar{c} = \left|\bar{a} \cdot \bar{b} \cdot r^{-1}\right|_n = \left|a \cdot r \cdot b \cdot r \cdot r^{-1}\right|_n = \left|c \cdot r^{-1}\right|_n .$$

Thus, the n-residue representation is stable over Montgomery multplication. At the end, a final conversion step has to be performed to transform the result back from the n-residue representation to normal residue representation. An important observation is that these forward and backward conversions can be carried out using the same algorithm as follows:

$$\begin{aligned} \bar{a} &= MontMult(a, r^2) = \left|a \cdot r^2 \cdot r^{-1}\right|_n = |a \cdot r|_n \\ a &= MontMult(\bar{a}, 1) = \left|a \cdot r \cdot r^{-1}\right|_n = |a|_n . \end{aligned} \tag{3.2}$$

The method requires an additional integer n' such that

$$r \cdot r^{-1} - n \cdot n' = 1.$$

This relation is an immediate consequence of Bezout's identity, which states that the greatest common divisor of two numbers can be represented as a linear sum of the two

numbers. A possible solution for computing the values of r^{-1} and n' would be based on the utilization of the extended Euclidian Algorithm A.2. Also, applying a modulo r reduction to both sides of the previous equation, would result in $|-n \cdot n'|_r = 1$, which further implies that $n' = \left|-n^{-1}\right|_r$.

The Montgomery multiplication algorithm is outlined in Algorithm 4.

Algorithm 4 Montgomery multiplication

Input: an odd modulus n and a radix $r = 2^{\lceil log_2 n \rceil}$ such that $gcd(n, r) = 1$,
an auxiliary value $n' = \left|-n^{-1}\right|_r$,
2 n-residue integers $\bar{a}$ and $\bar{b}$ such that $\bar{a} \cdot \bar{b} < r \cdot n$.

Output: the product $\left|\bar{a} \cdot \bar{b} \cdot r^{-1}\right|_n$.

function MontMult($\bar{a}, \bar{b}$)

1. $t := \bar{a} \cdot \bar{b}$
2. $v := |t \cdot n'|_r$
3. $u := (t + v \cdot n)/r$
4. **if** $u \geq n$ **then** $u := u - n$
5. **return** u

In step 2. the integer n' is used in conjunction with the LSB half of t (since $|t \cdot n'|_r = ||t|_r \cdot n'|_r$), in order to compute the number of multiples of the modulus n that need to be added to t, such that the LSB half of u becomes zero. Now, since the lower half of u is 0, it can be safely shifted to right, which is equivalent to a division by r (step 3.). Note that an addition of an integer multiple of the modulus, such that u becomes a multiple of r, $|t + v \cdot n|_r = 0$, does not change the congruency modulo n: $|t + v \cdot n|_n = |t|_n$. An extra subtraction of n may be necessary at the end to guarantee that the result is fully reduced with respect to the modulus n, that is $0 \leq u < n$ (step 4.). This holds true, because we have a maximum bound of $2n$ for for the value of u in step 3. Under the assumption that $t < r \cdot n$

$$u = \frac{t + v \cdot n}{r} < \frac{r \cdot n + r \cdot n}{r} = 2n.$$

Example 3.3 *For $a = 21$ and $b = 7$, compute $|a \cdot b|_{13}$ using the Montgomery method.*

Since the modulus $n = 13$, we can take $r = 2^{\lceil log_2 13 \rceil} = 2^4 = 16$ and thus $n' = 11$. Next, we have to convert the input values a and b to their n-residue form:

$$\bar{a} = |a \cdot r|_n = |21 \cdot 16|_{13} = 11,$$
$$\bar{b} = |b \cdot r|_n = |7 \cdot 16|_{13} = 8$$

Step	*MontMult(11,8)*
1.	$t := \bar{a} \cdot \bar{b} = 88$
2.	$v := |t \cdot n'|_r = |88 \cdot 11|_{16} = 8$
3.	$u := (t + v \cdot n)/r = (88 + 8 \cdot 13)/16 = 192/16 = 12$
4.	$12 < n$

Therefore, we have obtained $\left|\bar{a} \cdot \bar{b} \cdot r^{-1}\right|_n = 12$. *A subsequent call to the Montgomery multiplication will yield the desired result as follows:*

Step	*MontMult(12,1)*
1.	$t := 12 * 1 = 12$
2.	$v := \|t \cdot n'\|_r = \|12 \cdot 11\|_{16} = 4$
3.	$u := (t + v \cdot n)/r = (12 + 4 \cdot 13)/16 = 64/16 = 4$
4.	$4 < n$

Thus, $|a \cdot b|_n = |21 * 7|_{13} = 4$ *as it should.*

3.4.1 Montgomery Exponentiation

The main disadvantage of Montgomery's method is the time-consuming pre and post processing for conversions between normal residue and n-residue representation. Therefore Montgomery algorithm is more suited when several modular multiplications with respect to the same modulus have to be performed, as required in the case of modular exponentiation. In this way the ratio between the overhead associated with the conversions and the actual modular arithmetic computation is reduced and the Montgomery approach can outperform classical exponentiation methods.

The following algorithm summarizes the computation of the modular exponentiation $x = |m^e|_n$ employing Montgomery multiplications. As exponentiation algorithm, we utilize the binary method described in Subsection 3.2.2.

Algorithm 5 Montgomery exponentiation algorithm

Input: $0 < m, e < n$

Output: $|m^e|_n$

function MontExp(m,e,n)

1. $\bar{m} := |m \cdot r|_n = MontMult(m, \left|r^2\right|_n)$
2. $\bar{x} := |1 \cdot R|_n = MontMult(1, \left|r^2\right|_n)$
3. **for** i=k-1 **downto** 0 **do**

3a. $\bar{x} := \text{MontMult}(\bar{x}, \bar{x})$

3b. **if** $e_i = 1$ **then** $\bar{x} := \text{MontMult}(\bar{m}, \bar{x})$

4. $x := \text{MontMult}(\bar{x}, 1)$
5. **return** x

Steps 1. and 2. convert the input operand m and the initilization value 1 to their n-residue representation. After this preprocessing has been completed, the binary exponentiation method is applied. The square and multiplication operations modulo N are performed using $O(k)$ calls to the Montgomery multiplication routine, where k is the bitlength of the exponent e. After the n-residue of $|m^e|_n$ is obtained, an additional step (step 4.) is required to convert this result back to an ordinary residue number representation. Except the preprocessing overhead (step 1.-2.), the inner-loop (step 3a.-3b.)

performs only multiplications modulo 2^k and divisions by 2^k, which makes this approach very fast.

Example 3.4 *It is required to compute* $x = \left|5^{10}\right|_{13}$ *using the Montgomery exponentiation method.*

The first step is to choose r*. Since* $n = 13$*, we can take* $r = 2^4 = 16$*. Next the values of* r^{-1} *and* n' *are derived. Thus we have* $r^{-1} = 9$ *and* $n' = 11$*.*

Step 1. $\bar{m} = |m \cdot r|_n = |5 \cdot 16|_{13} = 2$*.*

Step 2. $\bar{x} = |1 \cdot r|_n = |1 \cdot 16|_{13} = 3$*.*

Step 3. Next we perform the binary exponentiation method.

E_i	*Step 4a.*	*Step 4b.*
1	*MontMult(3,3)=3*	*MontMult(2,3)=2*
0	*MontMult(2,2)=10*	-
1	*MontMult(10,10)=16*	*MontMult(2,16)=2*
0	*MontMult(2,2)=10*	-

Step 4. $x = MontMult(10, 1) = 12$*.*

Therefore, we obtain 12 as the result of $\left|5^{10}\right|_{13}$*.*

3.5 RNS Modular Multiplication

As described in Chapter 2, RNS exhibits several advantages over commonly employed fixed-radix, weighted number representations, that facilitates fast, parallel implementations of long integer arithmetic. This makes RNS a good candidate for supporting the long multiplications involved in Montogomery method. Recalling from Section 3.4, the Montgomery multiplication algorithm relies on the folowing two relations, repeated here for convenience:

$$\begin{cases} \left|t + \left|t \cdot n'\right|_r \cdot n\right|_r & = 0, \\ \left|t + \left|t \cdot n'\right|_r \cdot n\right|_n & = t. \end{cases}$$

The rationale behind choosing the value of r is to easily compute the operation modulo r in the first aforementioned equation. For weighted, binary number systems, this is achieved by letting r be an integer power of the radix 2, but for RNS representation, it is preferable to take r as the dynamic range of the RNS base. This way, all numbers less than r are already reduced modulo r. However, there is a drawback induced by this choice: all numbers greater than r need a larger RNS base to be represented. Therefore, costly base extension operations are required in order to compute these additional RNS residues.

The Montgomery algorithm was first adapted to RNS by Posch & Posch in [25]. In what follows, the RNS version of Montgomery multiplication Algorithm 4, is presented according to [7].

Algorithm 6 Montgomery Multiplication for RNS

Input: two RNS bases $B = (m_1, \ldots, m_k)$ and $B' = (m'_1, \ldots, m'_k)$ with $gcd(M, M') = 1$
a positive integer N represented in both bases such that $gcd(N, M) =$
$= 1$ and $0 < 4N < M < M'$
2 integers a and b, represented in both bases, with $a \cdot b < M \cdot N$

Output: $r' = \left|a \cdot b \cdot M^{-1}\right|_N$, represented in both bases

1. $t = a \cdot b$ in base $B \cup B'$
2. $q = t \cdot (-N^{-1})$ in base B
3. base extension: q from base $B \longrightarrow q'$ in base B'
4. $r' = (t + q' \cdot N)M^{-1}$ in base B'
5. base extension: r' from base $B' \longrightarrow r$ in base B

In order to derive the RNS Montgomery multiplication method, we consider two RNS bases $B = (m_1, \ldots, m_k)$ and $B' = (m'_1, \ldots, m'_k)$ of relatively prime moduli which implies that $gcd(M, M') = 1$, where M and M' are the dynamic ranges of the two bases. Eventhough it is not mandatory, it is assumed an equal number of elements for both bases, for regularity purposes of the correpondent hardware architecture or software implementation. Next we take $M = \prod_{i=1}^{k} m_i$ as the Montgomery constant r. Therefore the Montgomery method yields

$$\left|a \cdot b \cdot M^{-1}\right|_N,$$

where a, b, and the modulus N are positive integers represented in the predefined bases B and B', as

$$\begin{aligned} \langle a \rangle_B &= (a_1, \ldots, a_k) & \langle a \rangle_{B'} &= (a'_1, \ldots, a'_k) \\ \langle b \rangle_B &= (b_1, \ldots, b_k) & \langle b \rangle_{B'} &= (b'_1, \ldots, b'_k) \\ \langle N \rangle_B &= (N_1, \ldots, N_k) & \langle N \rangle_{B'} &= (N'_1, \ldots, N'_k) \end{aligned}$$

In step 1. we compute the product $a \cdot b$ in both RNS bases. This can be done in parallel for all moduli, in constant time, according to Equation (2.3) as follows:

$$\begin{aligned} t = |a \cdot b|_M &= (|a_1 \cdot b_1|_{m_1}, \ldots, |a_k \cdot b_k|_{m_k}), \\ t' = |a \cdot b|_{M'} &= (\left|a'_1 \cdot b'_1\right|_{m'_1}, \ldots, \left|a'_k \cdot b'_k\right|_{m'_k}). \end{aligned}$$

In step 2. we have to compute q such that $r = a \cdot b + q \cdot N$ is a multiple of M. The term $a \cdot b + q \cdot N$ represented in base B, composed solely of 0, since any multiple of M, represented modulo M equals 0. Thus, we have

$$\langle r \rangle_B = 0 \qquad \Leftrightarrow \qquad r_i = |a_i \cdot b_i + q_i \cdot N_i|_{m_i} = 0 \quad \text{for } i = 1, \ldots, k.$$

The value of q is then given by the solutions of the previous equations

$$q_i = \left|a_i \cdot b_i \cdot \left|-N_i^{-1}\right|_{m_i}\right|_{m_i} \qquad \text{for } i = 1, \ldots, k$$

Step 3. Since r is a multiple of M, as pointed out previously, it is composed only of 0 in base B, which further divided by M yields 0. Moreover, we have to perform the division by M, that reduces in RNS to a multiplication by the multiplicative inverse of M which unfortunately does not exist modulo M. Thus, we need a larger dynamic range to accommodate r. Also the new base moduli should be prime to M for the inverse $\left|M^{-1}\right|_{M'}$ to exist.
In step 3, a base extension is perfomed to obtain q', the RNS representation of q in base B'.

Step 4. Now we can evaluate $r' = \left|(t' + q' \cdot N') \cdot \left|M^{-1}\right|_{M'}\right|_{M'}$. The term $t' = |a \cdot b|_{M'}$ is already computed form step 1., the multplicative inverse $\left|M^{-1}\right|_{M'}$ is a precomputed constant, hence we obtain

$$r'_i = \left|(t_i + q'_i \cdot N'_i) \cdot \left|\left|M^{-1}\right|_{M'}\right|_{m'_i}\right|_{m'_i}.$$

Step 5. In order to be able to perform modular exponentiation by repeating the Montgomery multiplication, the range of the output should be made compatible with the range of the input. Hence we need to convert the result r' back to base B, which is achieved by applying again a base extension.

We note that if $a \cdot b < M \cdot N$ we have the output in the desired range that is $< 2N$, as suggested by the following derivation:

$$\frac{a \cdot b + q' \cdot N}{M} < \frac{M \cdot N + M \cdot N}{M} = 2N < M.$$

If only one multiplication is desired, then the condition $2N < M$ suffices for the algorithm to give the correct result. Otherwise, it is required that $4N < M < M'$ in order to reuse the ouput as input for a subsequent modular multiplication without any preceding reduction

$$(2N)^2 < M \cdot N \qquad \Rightarrow \qquad 4N < M.$$

There exist several RNS Montgomery multiplication methods in the literature. The main difference between them is the base extension algorithm, which induces different conditions for the range of the input and output values, or necessitates additional operations to counterbalance other introduced effects. In the following chapter, a survey of existing base extension methods is presented.

3.6 RNS Moduli Choice

The Montgomery method works for RNS basis with moduli of any form. However, if we choose the moduli of a particular form, the overall performance can be improved. The number of moduli and their form both affect the complexity of the algorithm to be performed and the efficiency of the representation. Thus, selecting the proper set of moduli is a major issue. An important remark is that usually, we tend to choose the moduli such that thay are comparable in magnitude to the largest one, due to the fact that the computation speed is dictated by the largest modulus. In the following

we present an overview of different possible moduli and highlight their advantages and disadvantages.

- **Mersenne numbers**

Among these special forms, moduli of the form $m = 2^k - 1$ are of special interest [14]. Based on the property

$$\left|2^k\right|_m = 1,$$

the modulo reduction operation $|a|_m$ is greatly simplified. It requires at most two k-bit operations, for $a < m^2$, obtained by writting $a = a_1 \cdot 2^k + a_0$ and then reducing both sides modulo m and observing that:

$$|a|_m = \begin{cases} a_1 + a_2 & \text{if } a_1 + a_2 < m, \\ a_1 + a_2 - m & \text{if } a_1 + a_2 \geq m. \end{cases}$$

When these moduli are prime numbers, they are called Mersenne numbers. Nevertheless, this approach is rather unpractical. Since there exists only one Mersenne number of k bits, choosing as the bases moduli only relatively prime moduli of this form would result in a base with moduli not comparable in magnitude and a dynamic range wider than required.

- **Pseudo-Mersenne numbers**

Crandall [9] enlarged the class of Mersenne numbers and proposed in 1992 the pseudo-Mersenne numers. The pseudo-Mersenne moduli have the form $m = 2^k - c$, where $k \in \mathbb{N}$ and $c < 2^{k/2}$. The operation $|a|_m$ costs in this case $2 \cdot H(c) + 2$ k-bit additions, where $a < 2^{2k}$ and $H(c)$ denotes the Hamming weight of c. Thus, it is preferable to choose c, such that $H(c)$ is minimized.

- **Generalized Mersenne numbers**

Solinas introduced in 1999, [29] another class of moduli, the Generalized Mersenne numbers. A Generalized Mersenne number has the following polynomial form

$$m = f(2^k), \quad \text{where } f(X) = X^n - C(X) \\ \text{with } C \text{ polynomial of degree } \leq n/2 \text{ and } \|C\|_\infty = 1.$$

In the above definition $\|C\|_\infty = 1$ means that the coefficients f_i of the polynomial f belong to the set $\{-1, 0, , 1\}$. Among these numbers, one particular form that allows smaller RNS base dynamic ranges when compared to Mersenne numbers, while still maintaing the efficiency of the modulo reduction, is $m = 2^{k_1} - 2^{k_2} - 1$. The operation $|a|_m$, where $a < m^2$, requires at most 6 k_1-bit additions for $0 < k_2 < \frac{k_1+1}{2}$. Furthermore, the number of additions can be reduced to 4, if k_2 is taken bigger than 1. However, there is a drawback when using these numbers: there exists only one number for a given bitlength of the modulus and a fixed $f(X)$. This implies that each generalized Mersenne number necessitates a dedicated implementation.

- **Extended Generalized-Mersenne numbers**

In 2003, Chung and Hassan in [8] proposed an extension class by combining the pseudo-Mersenne numbers and the Generalized Mersenne numbers. The Extended Generalized Mersenne numbers are defined as follows:

$$m = f(2^k - c), \quad \text{where } f(X) = X^n - C(X) \quad \text{with } \|C\|_\infty = 1 \text{ and degree}(C) \leq n/2.$$

Among these moduli form alternatives, the Generalized Mersenne numbers are more adapted for computations with large numbers, and thus they are better suited for improving the performance of the RNS Montgomery algorithm.

3.7 Conclusion

This chapter provides an overview of the RSA cryptosystem, emphasizing its computational requirement, the modular exponentiation that is broken into a sequence of modular multiplications with respect to a large modulus. The binary exponentiation method and the Montgomery multiplication are scrutinized in detail. By combining the efficient modulo reduction operations provided by the Montgomery method with RNS, a powerfull arithmetic tool is obtained, which improves the global performance. We have also given guidelines regarding the moduli form and their influence on the overall performance. Then, the bottleneck of the RNS Montgomery multiplication algorithm is identified, most of computational effort is due to the two required base extensions. Several methods have been proposed in the literature for performing the two base extensions and are presented in detail throughout the next chapter.

4 Base Extension

RNS base extension is the process of finding the residue digits with respect to a new set of moduli, given the residue digits relative to another set of moduli. In most cases, the original base could be a subset of the new RNS base, that is one ore more additional moduli are appended to the original set.

Formally stated, given the RNS representation of some integer X in base $B = (m_1, m_2, ..., m_k)$, $\langle X \rangle_B = (x_1, x_2, ..., x_k)$, compute the representation of X in another RNS base $B' = (m'_1, m'_2, ..., m'_k)$, $\langle X \rangle_{B'} = (x'_1, x'_2, ..., x'_k)$. $M = \prod_{i=1}^{k} m_i$ denotes the dynamic range of the original base B, while $M' = \prod_{i=1}^{k} m'_i$ represents the dynamic range of the second base B'. Eventhough it is not necessary to use the same number of moduli for both bases, it is preffered this way for regularity purposes. These notations are utilized throughout the remaining of the chapter.

Several techniques have been suggested in the literature for this purpose and are discussed in this chapter. Section 4.1 presents a short overview of the previous existing base extension methods. The section concludes with a comparative study of their performance, as seen form the prospective of the RNS Montgomery multiplication algorithm. In Section 4.2, a novel base extension method based on Diophantine equations is introduced.

4.1 Previous Work

Different methods have been proposed for performing RNS base conversion. However, there exist two main approaches: either using an auxiliary representation, e.g., Mixed Radix System or based on the CRT. All other methods are only variations of these two methods, which variations arise from choosing a particular set of moduli or from adapting certain properties in order to match a particular chosen approach.

4.1.1 Base extension based on the CRT

The CRT based methods make use of one of the two alternative forms of CRT, described by Equation (2.6a) and (2.6b), which are remainded here for convenience:

$$\sum_{i=1}^{k} M_i \cdot x_i \cdot \left| M_i^{-1} \right|_{m_i} = X + \alpha_1 \cdot M,$$

$$\sum_{i=1}^{k} M_i \cdot \left| x_i \cdot \left| M_i^{-1} \right|_{m_i} \right|_{m_i} = X + \alpha_2 \cdot M.$$

As it stands, provided the value of α is known, a new residue x_{k+1} with respect to the modulus m_{k+1} can be computed by

$$x_{k+1} = |X|_{m_{k+1}} = \left| \sum_{i=1}^{k} \left| x_i \left| M_i^{-1} \right|_{m_i} M_i \right|_{m_{k+1}} - |\alpha M|_{m_{k+1}} \right|_{m_{k+1}} \tag{4.2a}$$

$$x_{k+1} = |X|_{m_{k+1}} = \left| \sum_{i=1}^{k} \left| x_i \left| M_i^{-1} \right|_{m_i} \right|_{m_{k+1}} \cdot |M_i|_{m_{k+1}} - |\alpha M|_{m_{k+1}} \right|_{m_{k+1}} \tag{4.2b}$$

Hence, α has to be derived, whose computation is the main difference in the distinguished techniques. Its value can either be computed exactly or approximated (using fixed or floating point arithmetic).

- **Shenoy and Kumaresan** [28]

The method relies on the knowlegde of the residue x_r with respect to an additional redundant modulus $k \leq m_r < M$, chosen relatively prime to all other moduli from both bases ($gcd(M, m_r) = gcd(M', m_r) = 1$).

The exact value of α can be derived from Equation (4.2b) written with respect to the modulus m_r, as follows:

$$x_r = |X|_{m_r} = \left| \sum_{i=1}^{k} |M_i|_{m_r} \cdot \left| x_i \left| M_i^{-1} \right|_{m_i} \right|_{m_r} - |\alpha M|_{m_r} \right|_{m_r},$$

$$|\alpha M| = \left| \sum_{i=1}^{k} |M_i|_{m_r} \cdot \left| x_i \left| M_i^{-1} \right|_{m_i} \right|_{m_r} - x_r \right|_{m_r}.$$

Multiplying both sides of the above equation with the multiplicative inverse $\left|M^{-1}\right|_{m_r}$, yields

$$|\alpha|_{m_r} = \alpha = \left| \left|M^{-1}\right|_{m_r} \left(\sum_{i=1}^{k} |M_i|_{m_r} \cdot \left| x_i \left| M_i^{-1} \right|_{m_i} \right|_{m_i} - x_{m_r} \right) \right|_{m_r}. \tag{4.3}$$

Recalling from Equation (2.7) that for this form of the CRT (Equation (2.6b)), the value of α is upper bounded by the number of moduli k, and since the redundant modulus m_r was chosen such that $k \leq m_r \leq M$, it follows that $|\alpha|_{m_r} = \alpha$.

- **Posch and Posch** [24], [25]

Posch & Posch presented in [24] a base extension method that approximates α using either floating point or fixed point arithmetic.

As opossed to Shenoy and Kumaresan technique, this method uses the alternative form of the CRT described in Equation (2.6a). Dividing both sides of this equation by

the dynamic range M, results in

$$\alpha + \frac{X}{M} = \sum_{i=1}^{k} x_i \cdot w_i, \qquad \text{where } w_i = \frac{\left|M_i^{-1}\right|_{m_i}}{m_i} < 1. \tag{4.4}$$

Then, they made the following oberservation: α is an integer, which implies that the rational number $\frac{X}{M} < 1$ should cancel out the fractional part of the right hand sum. Therefore, the desired value of α is obtained by rounding the sum towards minus infinity

$$\alpha = \left\lfloor \sum_{i=1}^{k} x_i \cdot w_i \right\rfloor. \tag{4.5}$$

Basically, their method computes a weighted sum of all x_i, with the weights being rational constants, < 1 and depending on the set of moduli m_i.
The effect of limited precision representation of the w_i weights, is reflected in the error δ_i

$$w_i = w_i^* + \delta_i.$$

Using these truncated values w_i^*, instead of the accurate ones (w_i), introduces a total error ϵ that in rare occurences could lead to an α greater by one than the estimated value

$$\alpha = \left\lfloor \sum_{i=1}^{k} x_i \cdot w_i^* + \epsilon \right\rfloor = \lfloor \alpha^* + \epsilon \rfloor, \qquad \text{where } \epsilon = \sum_{i=1}^{k} x_i \cdot \delta_i.$$

As a result the base extension method returns either the correct value x_{k+1} or $x_{k+1} - M$. The latter case requires additional steps to be dealt with.

In order to keep the error within certain bounds ($<< 1$), it is required that the fixed sized mantissa of the arithmetic unit in use should have at least $\lceil log_2 k + m + \xi \rceil$ bits, where m accounts for the maximum bit lentgh of the base moduli m_i and ξ is a parameter > 2 used to discrimate whether $\alpha = \lfloor \alpha^* \rfloor$ or $\alpha = \lfloor \alpha^* + 1 \rfloor$.

This method was further enhanced in [25], by tuning up the RNS Montgomery multiplication algorithm in order to cope with the offset of M, such that the exception handling module for detecting and correcting the wrong approximation of α is eliminated.

- **Kawamura et al.** [13]

Kawamura et al. propose a similar technique that approximates α by fixed point computations. They consider Equation (2.6b), rearrange it and divide it by M

$$\alpha + \frac{X}{M} = \sum_{i=1}^{k} \frac{w_i}{m_i}, \qquad \text{where } w_i = \left| x_i \cdot \left|M_i^{-1}\right|_{m_i} \right|_{m_i}. \tag{4.6}$$

Therefore,

$$\alpha = \left\lfloor \sum_{i=1}^{k} \frac{w_i}{m_i} \right\rfloor \tag{4.7}$$

When compared to Posch's solution, this representation has the advantage that α is upper bounded by k, which leads to a simpler derivation of its value.

They approximate α by using as numerator the most significant q bits of w_i, denoted by $trunc_q(w_i)$ and as denominator 2^m, followed by an addition of a suitable chosen offset δ to compensate the induced approximation errors as follows:

$$\alpha = \left\lfloor \sum_{i=1}^{k} \frac{trunc_q(w_i)}{2^m} + \delta \right\rfloor = \left\lfloor \sum_{i=1}^{k} \frac{\lfloor w_i/2^{m-q} \rfloor}{2^q} + \delta \right\rfloor, \tag{4.8}$$

with m the maximum bit-length of the moduli m_i.

If the offset δ is chosen 0 the algorithm outputs either the correct α value, or the correct value plus 1, as obtained also from the Posch & Posch technique.
However, provided δ is chosen such that

$$\frac{1}{2^q} < \frac{\xi}{k}\left(1 - \frac{\epsilon k}{\xi}\right) \qquad \Leftrightarrow \qquad 2^{-q} < \frac{\xi}{k} - \epsilon, \tag{4.9}$$

where $2^{-(w-q)} << 1$ and $\epsilon = max_k\left(\frac{2^w - m_i}{2^w}\right) << 1$, then the base extension algorithm is error-free and the approximate value equals α.

- **Bajard et al.** [7]

Bajard et al. follow a rather different approach. They do not compute α at all, instead they allow an offset of $\alpha M \leq (k-1)M$ to occur in Equation (4.2a). They compute

$$|X + \alpha \cdot M|_{m_{k+1}} = \left| \sum_{i=1}^{k} \left| x_i \left| M_i^{-1} \right|_{m_i} M_i \right|_{m_{k+1}} \right|_{m_{k+1}}.$$

Therefore, instead of extending X, the authors extend $X + \alpha \cdot M$. This method is not a generally available base extension method; it was developed as an optimization in the context of RNS Montgomery multiplication algorithm. Its corectness relies on the fact that the offset is eliminated via a second base extension.

Performance evaluation

Subsequently, the aforementioned methods are put in the context of the RNS Montgomery multiplication method, outlined in Algorithm 6, and their performance is analyzed. Before further proceeding with the methods, a remark is in order: the underlying implementation of the RNS arithmetic for all methods, assumes $2k$ computational elements (functional units) working in parallel. The $2k$ value is due to the assumption that the cardinality of both RNS bases is equal to k. Each of these elements is dedicated to a specific modulus m_i and can compute independently

$$|x_i \cdot y_i + z_i|_{m_i}.$$

We consider that this operation takes one cycle to complete. In consequence, performance is assesed either in terms of the number of elementary arithmetic operations (modular

multiplication and addition, as described in the equation above) or in terms of the asymptotic time complexity.

The RNS Montgomery algorithm requires two base extensions: step 3. and step 5. from Algorithm 6. However, as explained below, not all presented CRT-based base extension methods can be used for both directions of conversion required by the Montgomery technique.

Shenoy et al. technique requires as a prerequisite that the residue modulo the redundant modulus m_r is known beforehand. We now consider the first base extension case. We recall that in step 2., the value of q is computed in base B. The major drawback of emplyoing Shenoy method for the first base extension is the fact that whenever the value of q is greater than M, an operation modulo M is required prior to the reduction by m_r. Since there is no efficient way to carry the redundant residue through the computation of $|q|_M$, the method is not suited for the first base extension. However, for the second base extension, the method can be efficiently used.

The method of Posch et al. is available for both base extensions. However, the Montgomery algorithm has to be modified according to the needs of the base extension method, that is additional steps are required to detect and fix the wrong approximations of the result and thus we will be exclud it from our considerations.

The method of Kawamura et al. can also be used for both base extensions. In order to use the Montgomery algorithm repeatedly to construct a modular exponentiation, the base extension method requires that

$$\begin{cases} 2N/(1-\xi) & \leq M \\ 4N/(1-\Delta) & \leq M' \end{cases} \qquad \text{where } \Delta = k(\epsilon+\delta),$$

and $\epsilon = max(2^m - m_i)/2^m$ and $\delta = max(w_i - trunc(w_i))/m_i$ accounts for the denominator, and respectively the numerator approximations from Equation (4.8). It can be easily proved that with these conditions satisfied and keeping in mind that $r' < (1+\Delta)M$, all intermediate values are less than MM' and the largest output value is less than $2N$:

$$\begin{aligned} r' = \frac{ab+q'N}{M} &< \frac{4N^2 + (1+\Delta)MN}{M} \\ &\leq \frac{(1-\Delta)MN + (1+\Delta)MN}{M} \\ &= 2N \\ &\leq (1-\xi)MM' \\ &\leq MM'. \end{aligned}$$

Finally, we consider the method of Bajard et al. and assume that we use it for the first base extension. After performing step 3., we obtain the extended value $q' = q + \alpha M$. At step 4., $r' = (ab+q'N)M^{-1}$ has to be evaluated, which now becomes

$$r' = (ab+q'N)M^{-1} = (ab+qN)M^{-1} + \alpha N = r + \alpha N.$$

Since no α is computed, the algorithm has the advantage of being fast but at the expense of additional measures of precaution. The conditions $ab < MN$, $\alpha < k$, and $q < M$ result in

$$r' = \frac{ab + qN}{M} + \alpha N \quad < \quad \frac{MN + MN}{M} + kM = (k+2)N \quad < \quad M'.$$

Furthermore, in order to apply repeatedely the Mongomery multiplication, without reducing the intermediate results, it is required that

$$ab < MN \quad \Rightarrow \quad ((2+k)N)^2 < MN \quad \Rightarrow \quad M > (2+k)^2 N.$$

We note that if r' is reduced modulo N, it gives the correct result: $\left|ab \cdot M^{-1}\right|_N$. In conclusion, a maximum offset of $(n-1)M$ can be carried through the computations. This unwanted offset is removed by the second base extension method.
If we apply Bajard method also for the second base extension, then we would end up with a value r^* in base B that is not equal to r' in base B'. As a consequence, r in base B is not derived correctly, making it an invalid input for a subsequent execution of the Montgomery algorithm.
Thus the Bajard method is suitable only for the first base extension.

Table 4.1 summarizes the available base extension methods performance, the required conditions for the methods to be used within the Montgomery algorithm and their compatibility with the first and second base extension steps of the Montgomery algorithm.

Examining Table 4.1, we note that all methods exhibit the same asymptotic complexity both in terms of number of iterations performed (clock cycles) and in terms of required modular multiplications. However, in the context of RNS Montgomery multiplication, the state-of-the-art base extension methods are Shenoy et al. method and Bajard et al. technique. The former is suited only for the first base extension. The results are not perfectly reduced and the method relies on the fact that another base extension will follow to yield the correct result. The latter has the disadvantage that requires the knowledge of an extra residue beforehand and thus cannot be applied to the second base extension. Shenoy et al. method is preffered to Kawamura et al. technique mainly due to its simpler implementation (i.e., no pseudo floating point unit is required; only identical integer channels are needed). Furthermore, the conditions are easy to satisfy (e.g., no restriction on the choice of the moduli is imposed as in the case of Kawamura et al.).

4.1.2 Base extension via MRS

The main ideea behind performing base extension via Mixed-Radix Number System (MRS) is to derive the MRS representation of X and afterwards to compute the residues modulo the target base B'. Before explaining the conversion procedure, it is necessary to explain the system itself.

Definition 4.1 Mixed-Radix Number System (MRS) *Given a set of radices m_i and a set of digits a_i, where $0 \leq a_i < m_i$, a number X that belongs to the set $[0, \prod_{i=1}^{k} m_i)$*

	Kawamura et al.	Bajard et al.	Shenoy et al.
range of input values a, b	$a, b < 2N$	$a \cdot b < MN$	$a \cdot b < MN$
range of output value r'	$r' < 2N$	$r' < (k+2)N$	$r' < 2N$
other conditions	$2N/(1-\xi) \leq M$ $4N/(1-\Delta) \leq M'$ $\Delta \leq \xi$	$0 < (k+2)^2 N < M$	$\lvert r' \rvert_{m_r}$ known $4N < M$
time complexity	$O(log_2 k)$	$O(log_2 k)$	$O(log_2 k)$
no. of modular multiplications	$k^2 + 2k$	$k^2 + k$	$k^2 + 2k$
first base extension (step 3.)	•	•	-
second base extension (step 5.)	•	-	•

Table 4.1: **CRT-based base extension algorithms.**

can be uniquely represented in the mixed-radix form as the k-tuple $(a_1, ..., a_k)$ *such that:*

$$X = a_1 + a_2 \cdot m_1 + a_3 \cdot m_1 m_2 ... + a_k \cdot m_1 ... m_{k-1}. \tag{4.10}$$

This is a weighted, positional number system, in which each of the weights determines a different radix, whence the term "mixed-radix". An example of a mixed-radix number system is the familiar binary number system, in which $m_i = 2$ for all i, and the weights are consequently powers of two. In an RNS with a given set of moduli m_i, the MRS defined by the set of radices that is equal to the set of RNS moduli is called the associated system. **1. RNS to MRS conversion**

The Mixed Radix Conversion problem can be formulated as follows:

Given a RNS base B defined by the set of moduli $(m_1, ..., m_k)$*, compute the associated mixed radix digits* a_i *of a known residue number* $\langle X \rangle_B = (x_1, x_2, ..., x_k)$.

- **Szabo and Tanaka, [30]**

The classical MRS conversion was proposed by Szabo and Tanaka in 1967 and is briefly described next.
Performing a modular reduction w.r.t. m_1 on both sides of Equation (4.10) yields

$$|X|_{m_1} = a_1.$$

Hence a_1 is equal to the first residue digit x_1.
To find the second MRS digit, a_2, we subtract a_1 from both sides of Equation (4.10)

$$X - a_1 = a_2 \cdot m_1 + a_3 \cdot m_1 m_2 ... + a_k \cdot m_1 ... m_{k-1},$$

perform a reduction modulo m_2 that gives

$$|X - a_1|_{m_2} = |a_2 m_1|_{m_2},$$

and finally multiply both sides with the multiplicative inverse of m_1 with respect to the modulus m_2

$$\begin{aligned} \left|(X - a_1) \cdot \left|m_1^{-1}\right|_{m_2}\right|_{m_2} &= |a_2|_{m_2} \\ |x_2 - a_1|_{m_2} &= a_2. \end{aligned}$$

The third MRS digit can be derived in a similar manner:

$$\begin{aligned} |X - a_1|_{m_3} &= |a_3 m_2 m_1 + a_2 m_1|_{m_3} \\ \left|(X - a_1) \cdot \left|m_1^{-1}\right|_{m_3}\right|_{m_3} &= |a_3 m_2 + a_2|_{m_3} \\ \left|(X - a_1) \cdot \left|m_1^{-1}\right|_{m_3} - a_2\right|_{m_3} &= |a_3 m_2|_{m_3} \\ \left|((X - a_1) \cdot \left|m_1^{-1}\right|_{m_3} - a_2) \cdot \left|m_2^{-1}\right|_{m_3}\right|_{m_3} &= |a_3|_{m_3} \\ &= a_3. \end{aligned}$$

Following the same procedure, that is succesively subtracting and dividing in residue notation, all a_i values can be obtained:

$$\begin{aligned}
a_1 &= x_1 \\
a_2 &= \left|(x_2 - a_1) \cdot \left|m_1^{-1}\right|_{m_2}\right|_{m_2} \\
a_3 &= \left|((x_3 - a_1) \cdot \left|m_1^{-1}\right|_{m_3} - a_2) \cdot \left|m_2^{-1}\right|_{m_3}\right|_{m_3} \\
&\ldots \\
a_k &= \left|((...(x_k - a_1) \cdot \left|m_1^{-1}\right|_{m_k} - a_2) \cdot \left|m_2^{-1}\right|_{m_k} - ...a_{k-1}) \cdot \left|m_{k-1}^{-1}\right|_{m_k}\right|_{m_k}
\end{aligned} \tag{4.11}$$

The main drawback of this approach is that the mixed radix conversion as described by Equation (4.11) is inherently sequential. That is for computing a_i it is necessary to determine first all a_j, with $j = 1, ..., i-1$. Furthermore, for each a_i there are nested multiplications that must be carried out in sequence. The method requires a large number of arithmetic operations $k(k-1)/2$ modular additions and multplications, resulting in an asympototic complexity in the order of $O(k^2)$.

- **Gbolagade and Cotofana, [11]**

They propose a parallel approach that require a linear amount of arithmetic operations. The method enables parallelization by maximizing the utilization of the modulo adders and multipliers at each iteration. Due to the very nature of the evaluated expression, not all modulo units can be utilized in parallel at a particular iteration i. Based on this observation, they use the remaining modulo m_i units to calculate intermediate results that are required in further iterations. At each level, one MR digit is computed by

$$a_k = \left|\left|m_{k-1}^{-1}\right|_{m_{j+(k-1)}} \cdot \left|\left(y_{j+(k-1)}^{k-2} - y_{k-1}^{k-2}\right)\right|_{m_{j+(k-1)}}\right|_{m_{j+(k-1)}},$$

where y_a^b is an auxiliary variable, with the exponent b denoting different levels where MRDs are computed, and a incrementing as progress is made from one level to another. For instance, at level 1,

$$y_{j+1}^1 = \left|(x_{j+1} - x_1) \cdot \left|m_1^{-1}\right|_{mj+1}\right|_{m_{j+1}}$$

are computed in parallel by the different modulo m_i ways for $j = 1, ..., k-1$. For $j = 1$, the value of a_2 is derived. The remaining y values will be used in the next level.
Level 2 computes

$$y_{j+2}^2 = \left|\left|m_2^{-1}\right|_{m_{j+2}} (y_{j+2}^1 - y_2^1)\right|_{m_{j+2}}$$

for $j = 1, ..., k-2$, where $j = 1$ gives a_3. The process is continued untill all MRDs are obtained.

2. MRS to RNS conversion

The conversion form MRS to a target RNS can be formulated as follows:

Given the mixed-radix representation $(a_1, ..., a_k)$ *and the set of radices* $(m_1, ..., m_k)$, *find the residues of the MRS encoded number* X *modulo an RNS base* B', *defined by the set of moduli* $m'_1, ..., m'_k$.

- **Szabo and Tanaka, [30]**

The classical way to convert X from the MRS representation to the target RNS base B' is also due to Szabo and Tanaka.
The new residues can be obtained if Equation (4.10) is reduced by each target modulus m'_i

$$|X|_{m'_i} = |a_1 + a_2 \cdot m_1 + a_3 \cdot m_1 m_2 ... + a_k \cdot m_1 ... m_{k-1}|_{m'_i}, \tag{4.12}$$

which gives us

$$\begin{aligned} |X|_{m'_i} &= \left| |a_1|_{m'_i} + ... + |a_k|_{m'_i} \cdot |m_1 ... m_{k-1}|_{m'_i} \right|_{m'_i} \\ &= \left| \sum_{s=1}^{k} \left(|a_s|_{m'_i} \cdot \left| \prod_{t=1}^{s-1} m_t \right|_{m'_i} \right) \right|_{m'_i}, \end{aligned} \tag{4.13}$$

where $\left| \prod_{t=1}^{s-1} m_t \right|_{m'_i}$ are constants and so can be pre-computed and stored beforehand. This method is highly parallelizable, but at the expense of k^2 look-up tables, for base B' with moduli set cardinality of k.

- **Bajard et al., [6], [5]**

Recalling Equation (4.12) and rewriting it in a serial, reccursive manner yields

$$\begin{aligned} |X|_{m'_i} &= |a_1 + a_2 \cdot m_1 + a_3 \cdot m_1 m_2 ... + a_k \cdot m_1 ... m_{k-1}|_{m'_i} \\ &= |a_1 + m_1 \cdot (a_2 + ... m_{k-3} \cdot (a_{k-2} + m_{k-2} \cdot (a_{k-1} + m_{k-1} a_k)) ...)|_{m'_i}. \end{aligned}$$

The method requires $|m_j|_{m'_i}$ pre-computed constants. Each of these occurences are substituted by the difference $m_j - m'_i$, which is a small number (less than the maximal difference between two moduli of base B and B'). Thus under the assumption that all the moduli have roughly the same size, $|X|_{m'_i}$ can be obatined using these differences that are not completely reduced, but smaller than a predefined bound. In this way the need for the k^2 look-up tables is alleviated, but the sum in Equation (4.13) can no longer be computed in parallel.

Further improvements are made by encoding each modulus m_i and m'_i as the difference c_i, respectively c'_i to the first value that cannot be stored in the register anymore (e.g., for a wordlength of w, $m_i = 2^w - c_i$, with $c_i < w$). The authors use this differential encoding to compute efficiently the required modular multiplications.
The same approach can also be used for computing the multiplicative inverses $\left|m_i^{-1}\right|_{m_j}$ in the RNS to MRS conversion. We note that the addition of any multiple of m_j does not have any effect on the final residue with respect to the modulus m_j:

$$\left|m_i^{-1}\right|_{m_j} = \left|(2^w - c_i)^{-1}\right|_{m_j} = \left|(2^w - c_i - (2^w - c_j))^{-1}\right|_{m_j} = \left|(c_j - c_i)^{-1}\right|_{m_j},$$

where $c_j - c_i$ is a small positive number, assuming that the moduli are sorted ascending after size. Using this differential approach, the storage requirements for each modulus are signifincantly reduced, and thus the complexity of the conversion is also reduced. The authors propose an efficient way of performing modular multiplications using this differential encoding approach.

Performance evaluation

The base extension methods via MRS are available for both step 3. and step 5. of the Montgomery Algorithm 6. They are error-free, and thus do not require any additional conditions besides the ones mentioned in Section 3.5:

$$\begin{cases} 4N < M' \\ 2N < M. \end{cases}$$

However, since generally these methods are slower than the CRT-based methods, we decided not to study their performance in details.

4.2 Diophantine Base Extension

This section begins with a short introduction into the mathematics of linear Diophantine equations [19]. Then a detailed description of the proposed base extension algorithm is presented, followed by a numerical example to illustrate the process. Finally, the method performance is evaluated and compared with the state-of-the-art base extension algorithms.

4.2.1 Diophantine Equations Basics

Definition 4.2 (Diophantine Equation) *Let $x_1, x_2, \ldots, x_n$ be n variables and let $f(x_1, x_2, \ldots, x_n)$ be a polynomial in these variables with integer coefficients. The indeterminate equation $f(x_1, x_2, \ldots, x_n) = c$, with the added proviso that the solutions are rational integers (i.e., in the ring $\mathbb{Z}$) or rational numbers (i.e., in the field $\mathbb{Q}$), is called a diophantine equation.*

Definition 4.3 (Linear Diophantine Equation) *A linear diophantine equation is an equation of the form*

$$a_1x_1 + a_2x_2 + \ldots + a_nx_n = c,$$

where $a_1, a_2, \ldots, a_n$ and c are integers, and where integer solutions are sougth for the unknowns $x_1, x_2, \ldots x_k$.

In what follows, we restrict ourselves to the case of linear diophantine equations in two variables.

Theorem 4.1 (Integer solutions existence) *Let a, b, c be integers with $ab \neq 0$. The linear Diophantine equation $ax + by = c$ has integer solutions in x and y if and only if $gcd(a, b)|c$.*

Theorem 4.2 (General form of solutions) *Let a, b, c be integers with $ab \neq 0$. If a primitive solution (x_0, y_0) of the linear Diophantine equation $ax + by = c$ is known, then the general form of solutions is given by*

$$\begin{cases} x = x_0 + t \cdot (b/d) \\ y = y_0 - t \cdot (a/d), \end{cases}$$

where $d = gcd(a, b)$ and t is a parameter $\in \mathbb{Z}$.

This is evident, since

$$a(x - x_0) + b(y - y_0) = 0.$$

To be noted that the equation in two unknown variables $ax + by = c$, as an equation over the reals has a continuously infinite number of solutions, while as a Diophantine equation, it has only a countably infinite number of solutions.

A particular solution (x_0, y_0) can be found by expressing $d = gcd(a, b)$ as a linear combination of a and b: $d = ua + vb$ and then multiplying it by c/d

$$\begin{cases} x_0 &= uc/d \\ y_0 &= vc/d, \end{cases}$$

while u and v can be found for instance using the extended Euclidian algorithm, decribed in Appendix A.2.

4.2.2 Proposed Diophantine Base Extension Method

In the following theorem we introduce a novel base extension method, which makes use of linear Diophantine equations theory in order to reduce the amount of calculations required by the base extension process.

Theorem 4.3 (Diophantine Base Extension) *Given the residues x_i of an integer $X < M = \prod_{i=1}^{k} m_i$ and the relatively prime moduli m_i, where $i = 1, ..., k$, the residue x_{k+1} w.r.t. to an additional modulus $m_{k+1} < M$ such that $gcd(m_{k+1}, M) = 1$, can be computed using the subsequent relations*

$$\begin{aligned} v &= \left\lfloor \sum_{i=1}^{k} \left(\frac{m_{k+1}}{m_i} \cdot x_i \cdot \left| (M_i \cdot m_{k+1})^{-1} \right|_{m_i} \right) \right\rfloor, \\ x_{k+1} &= \left| \pm c \cdot v \right|_{m_{k+1}}, \end{aligned} \tag{4.14}$$

where c is a positive or negative precomputed value.

The proof is as follows.
We denote with B' the extended base, composed out of the original base moduli m_i, with $i = 1, ..., k$, to which the m_{k+1} modulus is appended.

First, we represent X in this extended base B', using the CRT reconstruction formula in Equation (2.5)

$$X = \left| \sum_{i=1}^{k+1} M'_i \cdot \left| x_i \left| M_i'^{-1} \right|_{m_i} \right|_{m_i} \right|_{M'},$$

where M' is the dynamic range of base B'.
Since $M' = M * m_{k+1}$ and base B is a subset of base B', we can split the sum from the previous equation in two parts: the former term that contains all the base B residues and the latter term that includes only the unknown residuu x_{k+1}:

$$X = \left| \sum_{i=1}^{k} M_i \cdot m_{k+1} \cdot x_i \cdot \left| (M_i \cdot m_{k+1})^{-1} \right|_{m_i} + M \cdot x_{k+1} \cdot \left| M^{-1} \right|_{m_{k+1}} \right|_{M'}.$$

Next, we rewrite the previous equation according to the Fundamental Theorem of the Remainder, and we obtain for some integer α as follows:

$$\sum_{i=1}^{k} M_i \cdot m_{k+1} \cdot x_i \cdot \left| (M_i \cdot m_{k+1})^{-1} \right|_{m_i} + M \cdot x_{k+1} \cdot \left| M^{-1} \right|_{m_{k+1}} = X + \alpha \cdot M \cdot m_{k+1}.$$

Rearranging the above and dividing both sides by M, yields

$$x_{k+1} \cdot \left| M^{-1} \right|_{m_{k+1}} = \alpha \cdot m_{k+1} + \frac{X}{M} - \sum_{i=1}^{k} \left(\frac{m_{k+1}}{m_i} \cdot x_i \cdot \left| (M_i \cdot m_{k+1})^{-1} \right|_{m_i} \right).$$

We observe that the left side of this equation is an integer. It follows that the right side has to be also an integer, which implies that the fractional part of $\sum_{i=1}^{k} \left(\frac{m_{k+1}}{m_i} \cdot x_i \cdot \left| (M_i \cdot m_{k+1})^{-1} \right|_{m_i} \right)$ must cancel out $\frac{X}{M}$.
The following result is thus obtained

$$x_{k+1} \cdot \left| M^{-1} \right|_{m_{k+1}} - \alpha \cdot m_{k+1} = - \left\lfloor \sum_{i=1}^{k} \left(\frac{m_{k+1}}{m_i} \cdot x_i \cdot \left| (M_i \cdot m_{k+1})^{-1} \right|_{m_i} \right) \right\rfloor \tag{4.15}$$

We note that the aforementioned equation is a linear Diophantine equation with two unknowns.
Recalling from the previous subsection, the equation has the form

$$a_1 \cdot x_1 + a_2 \cdot x_2 = a \qquad \text{where } gcd(a_1, a_2) = 1,$$

where a_1, a_2 and a are given integers and x_1, x_2 are the unknowns. The solution is given by the Euclidian algorithm for finding the greatest common divisor of a_1 and a_2, which is further detailed.
We may write:

$$a_2 = q_1 \cdot a_1 + r_1 \qquad \text{where } 0 < r_1 < a_1 \text{ and } gcd(a_1, r_1) = 1.$$

Then

$$x_1 = -q_1 \cdot x_2 + \frac{-r_1 \cdot x_2 + a}{a-1} = -q_1 \cdot x_2 + x_3$$

Therefore

$$r_1 \cdot x_2 + a_1 \cdot x_3 = a$$

We write again

$$a_1 = q_2 \cdot r_1 + r_2 \qquad \text{where } 0 \leq r_2 < r_1 \text{ and } gcd(r_2, r_1) = 1$$

Then

$$x_2 = -q_2 \cdot x_3 + \frac{-r_2 \cdot x_3 + a}{r_1} = -q_2 \cdot x_3 + x_4$$

and so

$$r_2 \cdot x_3 + r_1 \cdot x_4 = a$$

Continuing this process, we have a decreasing set of positive integers $r_1, r_2, ...$ untill we come to a stage where

$$r_{n-2} \cdot x_{n-1} + r_{n-3} \cdot x_n = a, \qquad r_{n-2} = 1.$$

Then $x_{n-1}, x_{n-2}, ..., x_2, x_1$ are given succesively in terms of a parameter x_n.
Thus instead of computing first a particular solution p, q, and then derive all integer solutions

$$x_1 = p + t \cdot a_2 \qquad x_2 = q - t \cdot a_1 \qquad \text{for } t \in \mathbb{Z},$$

as explained in Subsection 4.2.1, we have obtained another parametrization of the integer solutions x_1, x_2.

In Equation (4.15), we denote

$$x_1 = x_{k+1} \qquad a_1 = \left|M^{-1}\right|_{m_{k+1}} \qquad x_2 = B \qquad a_2 = -m_{k+1}$$

$$a = -\left\lfloor \sum_{i=1}^{k} \left(\frac{m_{k+1}}{m_i} \cdot x_i \cdot \left|(M_i \cdot m_{k+1})^{-1}\right|_{m_i} \right) \right\rfloor .$$

We note that M and m_{k+1} are coprime from hyphotesis, which implies that the multiplicative inverse $\left|M^{-1}\right|_{m_{k+1}}$ is also coprime with m_{k+1}. According to Theorem 4.1, this means that the equation will always have integer solutions. Solving the linear Diophantine equation as described above, yields the solution x_1, expressed in terms of x_n and a.
We observe that if $a_1 = 1$, there is no need to solve the Diophantine equation, since it reflects already the solution in terms of $x_n = x_2$ and a.
What we search for, is a particular solution p of x_1 such that $0 \leq p < m_{k+1}$. From the general form of the solution described by Theorem 4.2, we note that when increasing t with 1, the solution x_1 also increases with a_2. Otherwise stated, there is a unique solution p in the interval $[0, m_{k+1})$, given by

$$p = |x_1|_{m_{k+1}} = |\pm m_{k+1} \cdot x_n + c \cdot a|_{m_{k+1}} = |+c \cdot a|_{m_{k+1}} ,$$

where $c \in \mathbb{Z}$. In conclusion, if we take the absolute value of a and denote it with v, we obtain:

$$x_{k+1} = \begin{cases} |+c \cdot v|_{m_{k+1}} & \text{for } c < 0 \\ |-c \cdot v|_{m_{k+1}} & \text{for } c > 0 \end{cases}$$

This concludes the proof.

4.2.3 Numerical Example

Let $(6, 12, 17, 20)$ be the residues of $X < M$, w.r.t. to the moduli $(7, 13, 19, 29)$. The problem is to compute the residue x_{k+1} relative to the modulus $m_{k+1} = 8$.

First we derive the value of a:

$$\begin{aligned} a &= -\left\lfloor \sum_{i=1}^{k} \left(\frac{m_{k+1}}{m_i} \cdot x_i \cdot \left| (M_i \cdot m_{k+1})^{-1} \right|_{m_i} \right) \right\rfloor \\ &= -\lfloor 6 * 4.5714 + 12 * 1.2308 + 17 * 5.4737 + 20 * 7.7241 \rfloor = -289. \end{aligned}$$

The solution of the Diophantic equation $5x_1 - 8x_2 = a$ is precomputed as follows:

$$\begin{array}{lll} 5x_1 - 8x_2 = a & q_1 = -1 & r_1 = -3 \\ -3x_2 + 5x_3 = a & q_2 = -1 & r_2 = 2 \\ 2x_3 - 3x_4 = a & q_3 = -1 & r_3 = -1 \\ -x_4 + 2x_5 = a & & \end{array}$$

, followed by back-substitution to obtain the general solution

$$\begin{aligned} x_4 &= 2x_5 - a \\ x_3 &= 3x - 5 - a \\ x_2 &= 5x_5 - 2a \\ x_1 &= 8x_5 - 3a \end{aligned}$$

Finally, we compute the residue modulo m_{k+1} by

$$x_{k+1} = |-3a|_{m_{k+1}} = 3,$$

as it should.

4.2.4 Performance Evaluation

The best state-of-the-art RNS Montgomery Multiplication algorithm [3], uses Bajard et al. base extension method [7] for the first base extension and Shenoy et al. base extension technique [28] for the second base extension method.

- **Performance evaluation in terms of arithmetic operations**

Performing base extension relative to one modulus, results in the following number of operations:

Method	No. of modular multiplications	No. of regular multiplications
Bajard et al.	$k+1$	-
Shenoy et al.	$k+2$	-
Our method	1	k

One can observe that we have obtained the same asymptotic complexity as Bajard and Shenoy methods in terms of elementary operations performed. However, Bajard method is suitable only for the first base extension of the Montgomery algorithm (it returns the expected correct result plus an offset that is eliminated by the second base extension), while our method is exact, allowing to be used for both base extensions. Moreover most of the required multiplications are regular ones which may result in a substantial speedup in practice.

Since

$$\frac{m_{k+1}}{m_i} \cdot \left|(M_i \cdot m_{k+1})^{-1}\right|_{m_i}$$

are constants, they can be precomputed and stored offline. In what follows, we denote these constants by ξ_i. Our method involves fixed point computations. However this requirement can be alleviated and replaced with integer computations. This can be achieved if we multiply each constant ξ_i with 2^p, perform integer multiplications with each x_i, compute the sum and then divide the final result by 2^p as follows:

$$v = (\sum_{i=1}^{k} x_i \cdot \xi_i \, 2^p)/2^p.$$

p should be chosen sufficiently large such that we obtain the correct truncated result v. For example, for 64-bit moduli, p can be chosen 40.

When executing the RNS modular multiplication algorithm, the two base exensions performed for k moduli, require in total $2k^2 + 3k$ modular multiplications, when using Bajard-Shenoy combination and $2k^2$ regular multiplications and $2k$ modular multiplications when our method is employed. We note that for the state-of-the-art RNS Montgomery Multiplication with Bajard-Shenoy combination, the reported improvement relative to prior work [13] was of k modular multiplications. A modular multiplication involves a regular multiplication and a reduction, and thus it is more time consuming than a regular multiplication. Since the latency is dependent on the implementation, there is no effective way of normalizing the two operations in order to accurately compare the methods theoretically. Furthermore, k is actually a small number, which implies that the asymptotic notation might not be that relevant in practical situations. In this case the constants count and practical evaluations may reflect better the performance improvement.
If we consider that the latency of a normal multiplication represents a fraction μ from the latency of a modular multiplication, i.e., $\tau_m = \mu \cdot \tau$, an RNS Montgomery multiplication requires $2k^2 + 2\mu k$ multiplications when employing our method for both base extensions and $2\mu^2 k^2 + 3\mu k$ for the state-of-the-art Bajard-Shenoy combination. Generally speaking we can assume that, μ is a constant between 1 and 2 depending on the practical

	Our algorithm	Bajard + Shenoy	Kawamura et al.
Step 1, 2, 4 of Algorithm 6	$5k$	$5k+3$	$5k$
First base extension	k^2+k	k^2+k	k^2+2k
Second base extension	k^2+k	k^2+2k	k^2+2k
Total	$2k^2+7k$	$2k^2+8k+3$	$2k^2+9k$

Table 4.2: Number of arithmetic operations (multiplications) of three RNS Montgomery multiplication algorithms.

implementation details and k, the cardinality of the two RNS bases, is a small number, e.g., 6. In Figure 4.1, we present the theoretical speedup per Montgomery multiplication computed for different values of α (from 1.5 to 2 in increments of 0.05) and $k = 6$.

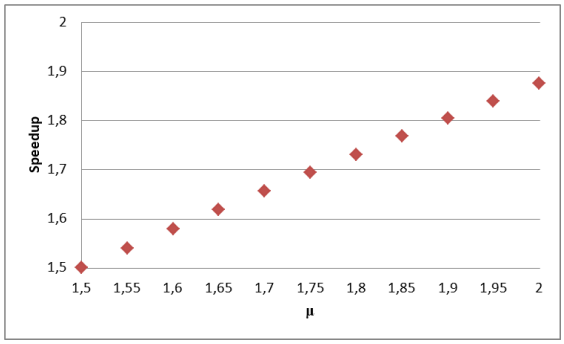


Figure 4.1: Theoretical speedup per Montgomery multiplication for k = 6 moduli and variable μ.

- **Performance evaluation in terms of time complexity**

Similar to the previous section, we assume k^2 processing elements working in parallel and that one modular multiplication and addition take 1 cycle. When performing the base extension relative to one modulus, we obtain the following figures of merit:

Method	No. of cycles consumed
Bajard et al.	$\lceil log_2 k + 1 \rceil$
Shenoy et al.	$\lceil log_2 (k+1) + 1 \rceil$
Our method	$\lceil log_2 k + 1 \rceil$

Thus, all methods have the same asymptotic complexity in the order of $O(log_2(k))$ for one modulus, and $O(k \cdot log_2(k))$ for k moduli.

4.3 Conclusion

This chapter begins with a thorough study of existing base extension methods. Then, the methods are put in the context of RNS Montgomery multiplication algorithm and

their performance is analyzed comparatively in terms of performed modular multiplications. Afterwards, the proposed Diophantine base extension method is presented in detail, and its performance is analyzed theoretically. Our method reduces the number of modular multiplications involved: for the derivation of k new residues, it requires k^2 regular multiplications and k modular multiplications, while the state-of-art methods necessitates $k^2 + k$ and $k^2 + 2k$ modular multiplications, respectively. We have obtained the same asymptotic complexity as the state-of-the-art base extension methods both in terms of arithmetic operations and in terms of time complexity. However, in the context of RNS Montgomery algorithm k is usually chosen a small number. Thus the asymptotic complexity does not reflect the actual speedup due to the fact that constants in this particular case, have a big impact on performance. Moreover, since our dominant operations are regular multiplications, which are generally less time consuming than modular ones, the effect on the overall performance is significant.
The proposed method can be used for both base extensions required by the Montgomery algorithm, as opposed to Bajard method that can be used only for the first base extension and to Shenoy method that can be employed solely for the second base extension. Furthermore, our method does not require any additional information that must be known beforehand (e.g., the residue with respect to a redundant modulus) as in the case of Shenoy base extension. The practical performance of the Diophantine base extension method is analyzed throughout the next chapter.

5 Performance Evaluation

This chapter analyzes the pratical implications of the proposed Diophantine base extension method on the overall RSA performance. The simulation environment is presented in Section 5.1, while the performance is evaluated in Section 5.2 both for the proposed method and for the state-of-the-art methods.

5.1 Simulation Setup

This subsection describes the simulation environment and the employed system components.

In order to ensure the required level of cryptographic strength, mathematical functions used in public key cryptography schemes require performing operations on large integers in the range of 512 to 2048 bits. Software implementation of such arithmetic operations is difficult since currently available processors have the word-size limited to up to 64 bits. Multiple algorithms have been developed to perform these multi-precision arithmetic operations efficiently, and several libraries implementing such algorithms exist both commercially and in public domain.
The GNU Multiple Precision library (GMP) [1] is a portable C library for arbitrary precision arithmetic on integers, floating point and rational numbers.
The selection of the GMP library was mainly driven by the fact that it offers a well-tested set of cryptographic operations and all the necessary primitives for implementing new algorithms. The library supports different processor families and provides optimizations in the form of assembly code for specific families. GMP represents large numbers internally in a signed-magnitude format. The magnitude of the number is represented as an array of unsigned integers called limbs, while a signed integer keeps track of the sign of the number and of the number of the limbs. A last variable in the structure is required for recording the space allocated for the array. GMP provides two APIs for development: a high level API which manages the memory allocated for the operands and the result and a low level API for time-critical user.

The RSA cryptosystem was coded in C++, compiled with g++ version 4.4.5 and ran on an Intel i5 540 machine at 2.54 GHz, under Ubuntu Linux, with a 2.6.35 kernel in single user mode. We implemented the RNS Montgomery multiplication Algorithm 6 and the RSA encryption-decryption protocol, considering 2 cases: in the former one, we used the proposed Diophantine base extension method for both base extensions, while in the latter one we used the state-of-the-art combination, that is Bajard et al. [7] method for the first base extension and Shenoy et al. method for the second base extension [28]. These two cases are denoted in what follows by Implementation 1 and Implementation 2, respectively.

As exponentiation technique, we use the left-to-right binary method described in

	Base B	Base B'
4 moduli	$2^{512}-2^{10}-1$ $2^{512}-2^{19}-1$ $2^{512}-1$ $2^{512}-2^{28}-1$	$2^{512}-2^{22}-1$ $2^{512}-2^{23}-1$ $2^{512}-2^{22}-1$ 2^{512}
5 moduli	$2^{512}-2^{10}-1$ $2^{512}-2^{19}-1$ $2^{512}-2^{28}-1$ $2^{512}-2^{20}-1$ 2^{512}	$2^{512}-2^{8}+1$ $2^{512}-2^{17}-1$ $2^{512}-2^{16}-1$ $2^{512}-2^{17}+1$ $2^{512}-2^{22}-1$
6 moduli	$2^{512}-2^{10}-1$ $2^{512}-2^{16}-1$ $2^{512}-2^{19}-1$ $2^{512}-2^{28}-1$ $2^{512}-2^{20}-1$ 2^{512}	$2^{512}-2^{26}-1$ $2^{512}-2^{18}+1$ $2^{512}-2^{25}-1$ $2^{512}-2^{17}-1$ $2^{512}-1$ $2^{512}-2^{5}-1$

Table 5.1: Example of 3 RNS moduli sets utilized for simulation

Algorithm 5. For all implementation scenarios, we employ the same public exponent $e = 2^{16}+1 = 65537$. This value a common choice for RSA cryptosystems, the encryption for this case requiring 16 modular squarings and one modular multiplication. Since the number of RNS moduli and their form, both have a big impact the algorithm performance and the efficiency of the representation, we used moduli sets of varying cardinality (k = 4, 5, 6) and form, i.e., generalized Mersenne numbers with different Hamming weights and bit lengths. An example of employed RNS moduli set is presented in Table 5.1. Throughtout the experiments, we denote by *mu* the ratio between the latency of a modular multiplication and the latency of a regular one.

5.2 Performance Analysis

The profiling analysis of our RSA implementation, revealed that modular exponentiation operation accounted for 99.5% of the overall execution time.

To achieve accurate, high-resolution timings in the order of micro-seconds, we used the Performance API (PAPI), developed by Browne et al. [2]. PAPI is an API for accesing hardware performance counters on different platforms. In order to achieve a better abstraction of various tasks, is built upon two different layers: the Portable Layer, that consists of the API for tool and application developers and the Machine Specific Layer, used to accces performance counters on a particular platform. The Portable Layer is composed of a low-level API, allowing acces to all core PAPI functions and direct interaction with the counter interface and a simple high-level interface for performing simple time measurements. Before the measurements were performed, the turbo boost, hyper-threading and speedstep technologies were disabled.

In order to evaluate the performance of our method, we considered different RSA key sizes, e.g., 512, 1024 bits, and several messages to encrypt. For every case, we measure the execution times of the RNS Montgomery multiplication kernel for both implementations. Each time we compute the speedup as the ratio between the execution time of Implementation 2 and the execution time of Implementation 1.

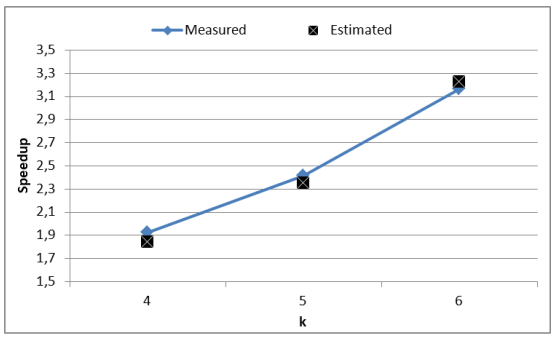


Figure 5.1: Measured vs. estimated (using μ estimated from the two Montgomery kernels) speedup for 512-bit RSA key n, and for 512-bit RNS moduli.

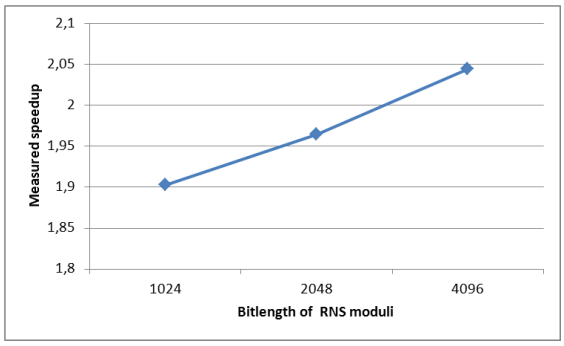


Figure 5.2: Measured speedup for 1024-bit RSA key N, and for k = 4.

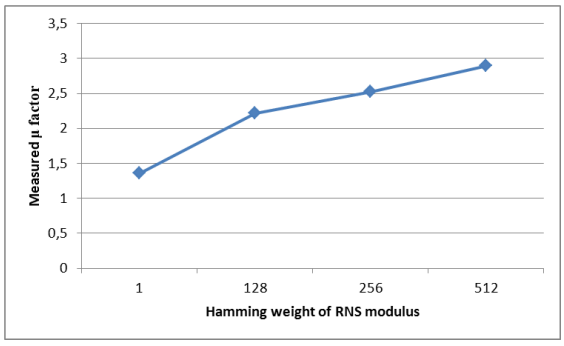


Figure 5.3: Measured speedup for 512-bit multiplication operands.

The main results of our experiments are graphically presented in Figure 5.1, 5.2, and 5.3. We observe that the speedup exhibits a linear increase tendency proportional with the RNS sets cardinality k, see Figure 5.1. Moreover, as indicated by Figure 5.2, the speedup increases for large bit length moduli. This is quite expected, since our method is dominated by normal multiplications, thus it exhibits a very small dependency on the form of RNS moduli, while the dominant operations in Bajard and Shenoy methods are modular multiplications. To get a better inside in the implications of the moduli form on the base extension complexity we also studied the way the Hammming weight of the RNS moduli influence the μ value. We assumed 512-bit moduli with various Hamming weights and determined the ratio between the coresponding modular multiplication and a regular one. The results are depicted in Figure 5.3 and one can observe that the larger the Hamming weight the larger μ, thus by implication the speedup provided by our method. Furthermore, we note that our method is general and can be efficiently applied to moduli of any form while for Bajard and Shenoy methods, choosing moduli that are not optimized for modulo operation cost reduction result in added complexity.

5.3 Conclusion

In order to evaluate the performance of our method, we implemented and simulated the RSA encryption-decryption protocol, using both the state-of-the-art methods and our proposed Diophantine base extension method. Several measurements were performed under different setup conditions. We considered several cases: (i) we varied the RNS moduli set cardinality; (ii) we varied the bitlength of the RNS moduli set; (iii) we varied the Hamming weight of an RNS modulus. We considered different RSA key-sizes, e.g., 512-bits, 1024 bits and messages to encrypt of varying length. For each case, we measured the execution time required per Montgomery kernel for both implementations and computed the speedup. Simulation results reveal that for sets of 4, 5 and 6 moduli of 512-bits, our method provides a speedup per Montgomery kernel of 1.93, 2.42, and 3.17, respectively.

6 Conclusions

This chapter summarizes in Section 6.1 the issues that were addressed by this thesis, along with the proposed solution. Section 6.2 presents the contributions of the performed research, while Section 6.3 concludes this thesis with some suggestions for future works.

6.1 Summary

Chapter 2 introduced the basic mathematical concepts underlying RNS. First, we mentioned the major difference between residue arithmetic and standard 2's complement arithmetic, which is the lack of carry-propagation (i.e., the independence of digits). A basic algebra of residues was developed and we have noted that addition and multiplication are easy to perform in RNS, as opposed to operations that require the determination of magnitudes (e.g., division, overflow, sign determination). Thus, the best applications for RNS are the ones involving several additions and multiplications modulo large numbers, such as the case of RSA cryptography. The chapter concluded with one of the most important results in RNS, the Chinese Remainder Theorem that allows the reconstruction of the binary/decimal number from its RNS representation.

Chapter 3 presented an overview of the RSA cryptosystem, emphasizing its computational requirements: modular multiplications with respect to a large modulus as core operations of exponentiation. Next, the left-to-right binary method, which provides a systematic way for finding the exact sequence in which squarings and multiplications have to be applied in order to compute efficiently the exponentiation, was detailed. Afterwards, the Montgomery algorithm, for performing effectively modular multiplications while avoiding division by the large modulus, was scrutinized in detail. In order to improve the global performance, RNS was combined with the Montgomery method and the bottleneck of this algorithm was identified as being the two required base extensions.

The main purpose of Chapter 4 was to survey the existing base extension methods and to propose a novel base extension technique that improves the overall performance of the RNS Montgomery algorithm. There are basically two main approaches for performing base extension: using the CRT or via an intermediate representation such as MRS. The methods were put in the context of the RNS Montgomery multiplication algorithm and their performances were compared. Then, the basic mathematical background of linear Diophantine equations was covered, followed by a detailed description of the proposed Diophantine base extension method. The method was compared with the state-of-the-art methods employed for the RNS Montgomery algorithm: Bajard et al. for the first base extension and Shenoy et al. for the second base extension and its performance was analyzed in terms of time complexity and arithmetic operations.

Chapter 5 evaluated practically the proposed method performance. The RSA encryption-decryption scheme was implemented in C++ both for our method and for the

state-of-the-art methods. Several measurements were performed under different setup conditions: for various RNS moduli sets, for different length of the plaintext to be encrypted, and for various RSA key sizes.

6.2 Main Results

In this thesis, a novel RNS base extension method is proposed to speedup the Montgomery modular multiplication. By making use of linear Diophantine equations theory, our method significantly diminishes the latency of the RNS base extension/conversion by reducing the number of required multiplications and by replacing most of the modular multiplications with regular ones. Given the representation of an integer X in an RNS base B comprising k relatively prime moduli $\{m_1, m_2, \ldots, m_k\}$, our method has an $O(log_2 k)$ asymptotic time complexity and requires k regular multiplications and one modular multiplication for the derivation of a residue digit with respect to a new relatively prime modulus m_{k+1}.

When applied in public key cryptography, in the context of RNS based Montgomery modular multiplication that requires two base extensions, each of them with respect to k new relatively prime moduli (forward base extension - from base B to base B' and backward base extension - from B' to B), our method provides a speedup of $O(\mu)$ relative to the state-of-the art, where μ is the ratio between the computation time required by a modular multiplication and by a regular one. The result is perfectly reduced and does not require any additional information that must be known beforehand, as opposed to similar state-of-the-art base extension techniques.

The method's practical performance was assessed by implementing the RSA encryption-decryption protocol in C++. Two implementations were made: the first one using our method and the second one using the state-of-the-art base extension methods. Several measurements were performed under different setup conditions: various RNS sets were considered (for k=4, 5, and 6), with different Hamming weights, the length of the RSA modulus was varied and the plaintext size was increased gradually. Experimental results obtained for software implementations of two RNS based RSA encryption-decryption kernels based on state of the art and on our approach, for 512-bit moduli indicate that for sets of 4, 5, and 6 moduli, our method provides a speedup per Montgomery kernel of 1.93, 2.42, and 3.17, respectively.

6.3 Proposed Future Research

The future research avenues can be summarized as follows:

- Based on our experience this far we see the task of reducing the number of multiplications for the base extension in the following perspective: (i) for a new modulus the new residue must be a function of all k residues in the initial base (and of course, of the new modulus); (ii) in the best case, a linear relation might exist and then at least k multiplications are required; (iii) for k new moduli at least k^2 mutiplications and thus further reduction of the number of multiplications is unlikely. Special cases may exist for which one encounters a situation similar to fast

transforms. Then more realistic approach would be to find such bases for which the multiplications could be shorter due to special form of the moduli (polynoms of powers of 2).

- The background of the proposed work lies in the CRT, which enables breaking down the multiplication in a field of a very large number in mutiplications into smaller fields. The main issue in this context is that division and other operations are not simpler but more cumbersome. Then if one could find a way to construct another representation of numbers in finite fields with a better behavior when writing them as a product of fields, more efficient algorithms might be developed.

Appendices

Auxiliary Algorithms

A

A.1 Computing the greatest common divisor using the Euclidian algorithm

Given two non-negative integers a and b, their greatest common divisor can be coomputed as follows:

Algorithm 7 Euclidian algorithm for computing the greatest common divisor of two numbers

Input: $a, b \in \mathbb{Z}$ such that $a \geq b > 0$

Output: $gcd(a, b)$

function Euclid(a, b)

1. **while** $b \neq 0$ **do**

1a. $r := |a|_b$

1b. $a := b$, $b := r$

2. **return** a

Example A.1 *Apply the Euclidian algorithm to find the greatest common divisor of* $a = 77$ *and* $b = 42$.

The following steps of division are performed:

$$77 = 42 \cdot 1 + 3542 \qquad = 35 \cdot 1 + 735 = 7 \cdot 5 + 0$$

So, we have obtained $gcd(77, 42) = 7$.

A.2 Expressing gcd(a,b) as a linear combination of a and b

Algorithm 8 Extended Euclidian algorithm for finding the multiplicative inverse

Input: $a, b \in \mathbb{Z}$ such that $a \geq b > 0$

Output: $d = gcd(a, b)$ and integers x and y such that $ax + by = d$

function ExtEuclid(a, b)

1. **if** $b = 0$ **then return** $(a, 1, 0)$
2. $x_1 := 1$, $x_2 := 0$, $y_1 = 0$, $y_2 = 1$, $r_1 = a$, $r_2 = b$, $i := 2$
3. **do**

3a. $i := i + 1$

3b. $q_i = \lfloor r_{i-2}/r_{i-1} \rfloor$

3c. $r_i = r_{i-2} \bmod r_{i-1}$

3d. $x_i = x_{i-2} - q_i \cdot x_{i-1}$

3e. $y_i = y_{i-2} - q_i \cdot y_{i-1}$

3f. **while** $r_i \neq 0$

4. **return** $(r_{i-1}, x_{i-1}, y_{i-1})$

Example A.2 *Let $a = 137$ and $b = 60$. It is required to compute $d = gcd(a, b)$,x and y such that $ax + by = d$.*

i	r_i	q_i	x_i	y_i
1	137	-	1	0
2	60	-	0	1
3	17	2	1	-2
4	9	3	-3	7
5	8	1	4	-9
6	1	1	-7	16
7	0			

Thus $gcd(137, 60) = 8$, $x = -7$ and $y = 16$.

A.3 Computing the multiplicative inverse using the Extended Euclidian algorithm

Given two positive coprime integers a and m, it is often required to find $\left|a^{-1}\right|_m$. The algorithm below, using the reccursive relation for finding the greatest common divisor between a and m: $gcd(a, m) = gcd(a, m \bmod a)$, computes x such that the following Diophantine equation holds $a \cdot x + m \cdot y = 1$.

Algorithm 9 Extended Euclidian algorithm for finding the multiplicative inverse

Input: $a \in \mathbb{Z}_m$ such that $gcd(a, m) = 1$

Output: $\left|a^{-1}\right|_m$ provided that it exists

function ExtEuclidInv(a, m)

1. $e := m$, $f := a$, $x := 0$, $v := 1$;
2. **while** $f > 0$ **do**

2a. $q := \lfloor e/f \rfloor$

2b. $r := e - q \cdot f$

2c. $e := f$, $f := r$

2d. $r := x - q \cdot v$

2e. $x := v$, $v := r$

3. **if** $x < 0$ **then** $x := m + x$
4. **return** x

Example A.3 *Let $a = 7$ and $m = 60$. It is required to compute $\left|a^{-1}\right|_m$.*

e	f	x	v	q	r
60	7	0	1	0	0
7	4	1	−8	8	−8
4	3	−8	9	1	9
3	1	9	−17	1	−17
1	0	−17	−52	3	−52

Hence, we have

$$\left|a^{-1}\right|_m = \left|7^{-1}\right|_{60} = |-17|_{60} = |60 - 17|_{60} = 43$$

.

Bibliography

[1] http://gmplib.org/.

[2] http://icl.cs.utk.edu/papi/index.html.

[3] J.-C. Bajard and L. Imbert, *Brief Contributions: A Full RNS Implementation of RSA*, IEEE Transactions on Computers, Vol. 53, No. 6, June 2004, pp. 769–774.

[4] J. C. Bajard, M. Kaihara, and T. Plantard, *Selected RNS Bases for Modular Multiplication*, 18th IEEE International Symposium on Computer Arithmetic, 2009, pp. 25–32.

[5] J.-C. Bajard, N. Meloni, and T. Plantard, *Efficient RNS Bases for Cryptography*, IMACS'05: World Congress: Scientific ComputationApplied Mathematics and Simulation, Paris, France, July 2005.

[6] J.-C. Bajard and T. Plantard, *RNS Bases and Conversions*, Advanced Signal Processing Algorithms, Architectures, and Implementations XIV, 2004, pp. 60–69.

[7] J.C. Bajard, L.S. Didier, and P. Kornerup, *Modular Multiplications and Base Extension in Residue Number Systems*, Proceedings ARITH15, the 15th IEEE Symposium on Computer Arithmetic, June 2001, pp. 59–65.

[8] J. Chung and A. Hassan, *More Generalized Mersenne Numbers*, Selected Areas in Cryptography SAC 2003, volume 3006 of LNCS, August 2003.

[9] R. Crandall, 1992, Method and Apparatus for Public Key Exchange in a Cryptographic System. U.S. Patent number 5159632.

[10] W. Diffie and M. E. Hellman, *New Directions in Cryptography*, IEEE Transactions on Information Theory, IT-22(6), November 1976, pp. 644–654.

[11] A. Gbolagade and S. D. Cotofana, *An O(n) Residue Number System to Mixed Radix Conversion*, 2009 IEEE International Symposium on Circuits and Systems, Taipei, Taiwan, China, May 2009, pp. 521–524.

[12] L. Hars, *Long Modular Multiplication for Cryptographic Applications*, Cryptographic Hardware and Embedded Systems CHES 2004 (LNCS 3156), 2004, pp. 45–61.

[13] S. Kawamura, M. Koike, F. Sano, and A. Shimbo, *Cox-Rower Architecture for Fast Parallel Montgomery Multiplication*, Proc. EUROCRYPT 2000, LNCS 1807, Springer Verlag, 2000, pp. 523–538.

[14] D. E. Knuth, *The art of computer programming: Seminumerical algorithms - volume 2*, Addison-Wesley, 1981.

[15] C. K. Koc, *High-speed RSA Implementation*, RSA Laboratories, Redwood City, CA, USA, November 1994.

[16] A. Menezes, P. van Ooschot, and S. Vanstone, *Handbook of applied cryptography*, CRC Press, 1996.

[17] A. Mohan, *Residue number systems: Algorithms and architectures*, Kluwer Academic Publishers, 2002.

[18] P. L. Montgomery, *Modular Multiplication Without Trial Division*, April 1985.

[19] L. J. Mordell, *Diophantine equations*, Academic Press Inc, 1969.

[20] H. Nozaki, M. Motoyama, A. Shimbo, and S. Kawamura, *Implementation of RSA Algorithm Based on RNS Montgomery Multiplication*, Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems, LNCS 2162, Springer Verlag, 2001, pp. 364–376.

[21] A. Omondi and B. Premkumar, *Residue Number systems: Theory and Implementation*, Imperial College Press, 2007.

[22] B. Parhami, *Computer arithmetic - algorithms and hardware designs*, Oxford University Press, New York, 2000.

[23] K. C. Posch and R. Posch, *Residue Number Systems: a Key to Parallelism in Public Key Cryptography*, Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing, 1992, pp. 432–435.

[24] _____, *Base Extension Using a Convolution Sum in Residue Number System*, Computing, vol. 50, June 1993, pp. 93–104.

[25] _____, *Modulo Reduction in Residue Number System*, IEEE Transactions on Parallel and Distributed Systems, vol. 6, May 1995, pp. 449–454.

[26] A. Shamir R. L. Rivest and L. Adleman, *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*, Communications of the ACM (1978), pp. 120 – 126.

[27] G. Saldamli and C. K. Koc, *Spectral Modular Exponentiation*, 18th IEEE Symposium on Computer Arithmetic, (ARITH '07), 2007, pp. 123–132.

[28] A.P. Shenoy and R. Kumaresan, *Fast Base Extension Using a Redundant Modulus in RNS*, IEEE Transactions on Computers, vol. 38, no. 2, February 1989, pp. 292–297.

[29] J. Solinas, 1999, Generalized Mersenne numbers. Research Report CORR-99-39, Center for Applied Cryptographic Research, University of Waterloo, Waterloo, ON, Canada.

[30] N. Szabo and R. Tanaka, *Residue Arithmetic and its Applications to Computer Technology*, McGraw-Hill, New York, 1967.

[31] P. T. P. Tang, *Modular Multiplication using Redundant Digit Division*, 18th IEEE Symposium on Computer Arithmetic (ARITH '07), 2007, pp. 217–224.

[32] A. Weimerscirch and C. Paar, *Generalizations of the Karatsuba Algorithm for Efficient Implementations*, http://eprint.iacr.org/, 2006.

[33] H. M. Yassine and W. R. Moore, *Improved Mixed-Radix Conversion for Residue Number System Architectures*, IEE Proceedings-G Circuits, Devices and Systems, vol. 138, no. 1, February 1991, pp. 120–124.