computing@tanet.edu.te.ua
www.tanet.edu.te.ua/computing

ISSN 1727-6209
International Scientific Journal of Computing

MODULAR EXPONENTIATION

Sorin Iftene [1)]

[1)] Faculty of Computer Science, "Al. I. Cuza" University, Iasi, Romania

e-mail: siftene@infoiasi.ro

Abstract: *Exponentiation is a fundamental operation in computational number theory. Primality testing and cryptography are important working fields in which the exponentiation is heavily used. In this paper we survey the most popular methods for modular exponentiation: basic techniques, fixed-exponent techniques, fixed-base techniques, and techniques based on modulus particularities. Some aspects related to parallelism are also discussed.*

Keywords: *modular exponentiation, basic techniques, fixed-exponent techniques, fixed-base techniques, techniques based on modulus particularities, binary fan-in technique.*

1. INTRODUCTION

The *exponentiation problem* in a monoid $(M, \cdot)$ is to compute a^n, for some $a \in M$ and integer $n > 0$. When M is a group, we may also raise the problem of computing a^n for negative integers n. The naive method for computing a^n requires n-1 multiplications, but as we shall see in this paper we can do much better. This is very important because exponentiation is heavily used in many working fields such as primality testing, cryptography, security protocols etc. In such fields, practical implementations depend crucially on the efficiency of exponentiation.

Although most of the methods we are going to present can be applied to any monoid, we shall deal only with *modular exponentiation*, that is exponentiation in $\mathbf{Z_m}$ where the multiplication is ordinary multiplication followed by reduction modulo m. Therefore, anything in this paper referrers to integers $a \in \mathbf{Z_m}$, $n \geq 1$, and $m \geq 2$.

We shall mainly discuss four types of exponentiation techniques:

- *general techniques*, where the base and the exponent are arbitrary;
- *fixed-exponent techniques*, where the exponent is fixed and arbitrary choices of the base are allowed;
- *fixed-base techniques*, where the base is fixed and arbitrary choices of the exponent are allowed;
- *techniques based on modulus particularities*, where special properties of the modulus are exploited.

In general, there are two ways to reduce the time required to do a modular exponentiation. One way is to reduce the number of modular multiplications used, and the other is to decrease the time required by a single modular multiplication. In this paper we shall focus on the first task.

Some of the presented algorithms lead to parallel implementations which will be discussed.

2. GENERAL TECHNIQUES

Most of the general exponentiation techniques, i.e., techniques which do not exploit any particularity of the exponent or of the base thus being generally applicable, can be viewed as particular cases or slight variants of the *sliding window method*. This method is based on arbitrary block decompositions of the binary representation of the exponent.

More exactly, we shall be sometimes interested in decomposing the binary representation of a positive integer n in blocks of bits. Moreover, it may be the case that we want that all the blocks have a given length and thus, an eventual padding on the left with 0's will be required. For example, we may need that the representation $(10100)_2$ be decomposed into 3 blocks each of length 2. In this case we shall write $[01,01,00]_2$. In general, we shall write $n = [w_{l-1}, \ldots, w_0]_2$ for an arbitrary decomposition of the binary representation of a positive integer n. The blocks w_i are usually called *windows*.

There are several variants of this method, depending on how the windows are built and

scanned. Usually, the windows can have prescribed values or can be built in an adaptive way. They can be scanned from left to right or vice-versa.

Any *left-to-right sliding window method* is based on a decomposition

$$a^n = (\cdots(a^{(w_{l-1})_2})^{2^{|w_{l-2}|}} \cdots a^{(w_1)_2})^{2^{|w_0|}} \cdot a^{(w_0)_2} \quad (1)$$

where $n = [w_{l-1},\ldots,w_0]_2$ is an arbitrary block decomposition of the binary representation of n. We can use a certain strategy of choosing the windows such that the resulted windows values are in a fixed set W. In this case, we obtain the following algorithm.

LRSlidWind(*a*, *n*, *m*, *W*)

input: $n \geq 1$, $m \geq 2$, $0 < a < m$, $W \subseteq \mathbf{N}$;

output: $x = a^n \mod m$;

begin

1. **for** each $i \in W$ **do**
   compute $x_i = a^i \mod m$;
2. **let** $n = (n_{k-1},\ldots,n_0)_2$;
3. $x := 1$;
4. $i := k-1$;
5. **while** $i \geq 0$ **do**
   begin
6. **find** an appropriate bitstring $n_i \ldots n_j$ such that $(n_i,\ldots,n_j)_2 \in W$;
7. **for** $l := 1$ **to** $i-j+1$ **do**
   $x := x \cdot x \mod m$;
8. $x := x \cdot x_{(n_i,\ldots,n_j)_2} \mod m$;
9. $i := j-1$;
   end

end

The computation of $a^i \mod m$, for all $i \in W$, can be efficiently performed by using the technique of addition sequences. In general, a *addition sequence* for a sequence of positive integers $n_0,\ldots,n_{l-1}$ is a addition chain (See Section 3) for $max(n_0,\ldots,n_{l-1})$ which includes the sequence $n_0,\ldots,n_{l-1}$. We shall discuss now some important possible choices of W:

- $W = \{0,1,3,\ldots,2^w-1\}$, for some $w \geq 1$. The parameter w is referred to as the *window size*. This variant has been proposed by Thurber[3], by using two kinds of windows: *zero windows*, formed by a single 0, and *odd windows*, which begin and end with 1 and have length at most w;
- $W = \{0,1,3,\ldots,2^w+f\}$, for some $w \geq 1$ and odd f, $1 \leq f \leq 2^w-3$. This case has been considered by Möller[4] as an alternative to the sliding window method using only zero and odd windows for limited memory environments. Sometimes, the space available is more than sufficient for the mentioned sliding window method with the window size w but it is not enough for the case of window size $w+1$. In order to take advantage of a larger window, we may try to apply the above method with the window size $w+1$ as long as the resulted window values are in the pre-computed range. The method obtained in this way is referred to as the *left-to-right sliding window method with fractional windows*;
- $W = \{0,1,2,\ldots,2^w-1\}$, for some $w \geq 1$. This variant has been considered by Brauer[5] by using only windows of length w. The method obtained in this way is referred to as the *left-to-right 2^w-ary method,* or simply, as the *left-to-right window method* for exponentiation;
- $W = \{0,1\}$. In this case we use windows formed by a single bit and obtain the so-called *left-to-right binary method* for exponentiation.

Any *right-to left sliding window method* is based on the decomposition

$$a^n = (a^{2^{k_0}})^{(w_0)_2} \cdots (a^{2^{k_{l-1}}})^{(w_{l-1})_2} \quad (2)$$

where $n = [w_{l-1},\ldots,w_0]_2$ is an arbitrary block decomposition of the binary representation of n, $k_0=0$ and $k_i = \sum_{j=0}^{i-1} |w_j|$, for all $1 \leq i \leq l-1$.

The last product can be re-arranged by grouping all the terms with the same exponent:

$$\prod_{i=0}^{l-1} (a^{2^{k_i}})^{(w_i)_2} = \prod_{j \in W} \Big(\prod_{\{i \mid (w_i)_2 = j\}} a^{2^{k_i}} \Big)^j \quad (3)$$

where W is the set of the predicted windows values. Such grouping was first used by Yao[6]. All these lead to the following algorithm.

RLSlidWindExp(*a*, *n*, *m*, *W*)

input: $n \geq 1$, $m \geq 2$, $0 < a < m$, $W \subseteq \mathbf{N}$;

output: $x = a^n \mod m$;

begin

1. **let** $n = (n_{k-1}, \dots, n_0)_2$;
2. $y := a$;
3. **for** every $i \in W$ **do** $x_i := 1$;
4. $i := 0$;
5. **while** $i \leq k$-1 **do**

   begin

6. **find** an appropriate bitstring $n_j \dots n_i$ such that $(n_j, \dots, n_i)_2 \in W$;
7. $x_{(n_j,\dots,n_i)_2} := x_{(n_j,\dots,n_i)_2} \cdot y \bmod m$;
8. $y := y^{2^{j-i+1}} \bmod m$;
9. $i := j+1$;

   end

10. $x := \prod_{j \in W} x_j^j \bmod m$;

end

The set W and the windows can be chosen as in the left-to-right sliding method. Depending on the set W, some tricks can be used for efficiently computing $\prod_{j \in W} x_j^j \bmod m$ from line 10:

- $W = \{0,1,3,\dots,2^w\text{-}1\}$, for some $w \geq 1$. In this case the mentioned product can be expressed as in [7, answer to Exercise 4.6.3-9]:

$$(x_{2^w-1})^2 \cdots (x_{2^w-1} \cdots x_3)^2 \cdot (x_{2^w-1} \cdots x_1) \quad (4)$$

- $W = \{0,1,2,\dots,2^w\text{-}1\}$, for some $w \geq 1$. This variant has been considered by Yao[6] by using only windows of length w. The method obtained in this way is referred to as the *right-to-left* 2^w*-ary method,* or simply, as the *right-to-left window method* for exponentiation. In this case the mentioned product can be expressed as in [8, answer to Exercise 4.6.3-9]:

$$x_{2^w-1} \cdot (x_{2^w-1} \cdot x_{2^w-2}) \cdots (x_{2^w-1} \cdots x_1) \quad (5)$$

- $W = \{0,1\}$. In this case we use windows formed by a single bit and obtain the so-called *right-to-left binary method* for exponentiation.

The step 7 and the step 8 of the previous algorithm can be performed in parallel.

Moreover, the expression $\prod_{j \in W} x_j^j \bmod m$ in line 10 can be evaluated in parallel as follows. Suppose we have $|W|$ processors. Each processor can compute $x_j^j \bmod m$ and then, the resulted values can be combined by using the so-called *binary fan-in multiplication technique* (See, for example, [9]). Thus, $\prod_{j \in W} x_j^j \bmod m$ can be computed, using $\lceil |W|/2 \rceil$ processors, following the next steps:

- Partition W in $\{W_1, W_2\}$ so that $|W_1| \approx |W_2|$
- Compute in parallel recursively
  - $P_1 = \prod_{j \in W_1} x_j^j \bmod m$
  - $P_2 = \prod_{j \in W_2} x_j^j \bmod m$
- Return $P_1 \cdot P_2 \bmod m$

3. FIXED-EXPONENT TECHNIQUES

In general, the problem of finding the smallest number of modular multiplications required to compute $a^n \bmod m$ is very similar to the problem of finding one of the shortest addition chains for n.

An *addition chain* of length t for a positive integer n is a sequence of integers $e_0 < \cdots < e_t$ such that $e_0 = 1$, $e_t = n$, and for all $1 \leq i \leq t$, there are $0 \leq j, h \leq i$-1 such that $e_i = e_j + e_h$.

If $e_0 < \cdots < e_t$ is an addition chain for n, $a^n \bmod m$ can be computed by evaluating, step by step, $x_i = a^{e_i} \bmod m$, for all $2 \leq i \leq t$, where, every term x_i is obtained as $x_i = x_j \cdot x_h \bmod m$, for some $0 \leq j, h \leq i$-1, and $x_1 = a$.

As we can see, the required number of multiplications is exactly the length of the addition chain for n. Thus, the shorter the addition chain is, the shorter the execution time for modular exponentiation is. But one can remark that minimizing the exponentiation time is not exactly the same problem as minimizing the addition-chain length, for several reasons:

- the multiplication time is not generally constant. For example, the squaring can be performed faster, or, if an operand is in a given fixed set, pre-computation can be used;
- the time of finding an addition chain should be added to the time spent for multiplications, and must be minimized

accordingly.

Nöcker[10] introduced the *q-addition chains* in the context of exponentiation in finite fields F_{q^n}, where the computation of a q^{th} power is essentially for free in case of normal basis representations.

Let $q \geq 2$. A *q-addition chain* of length t for a positive integer n is a list of integers $e_0 < \cdots < e_t$ such that $e_0 = 1$, $e_t = n$, and for all $1 \leq i \leq t$, either there are $0 \leq j, h \leq i\text{-}1$ such that $e_i = e_j + e_h$, either there is $0 \leq j \leq i\text{-}1$ such that $e_i = q \cdot e_j$. Nöcker proposed a parallel technique for constructing q-addition chains. The algorithm is presented below.

Parallel_*q*-addition_chain(*n*)

input: a positive integer n ;

output: L , a q -addition chain for n ;

begin

1. **represent** n as $n = (n_{k-1}, \ldots, n_0)_q$;
2. **if** $n = 1$ **then** $L = \{1\}$ **else**
3. **if** $q \mid n$ **then**

begin

4. **compute** L' , a q-addition chain for n/q by a recursive call on the same processor;
5. $L := L' \cup \{n\}$;
6. **end**

else

begin

7. **find** an appropriate i , $0 \leq i < k$;
8. **compute** the q-addition chains L_1 , L_2 for, respectively,

$$n_1 = \sum_{j=0}^{i} n_j q^j$$

$$n_2 = \sum_{j=i+1}^{k-1} n_j q^j$$

by recursive calls on two different processors;

9. $L :=$ **merge** (L_1, L_2) ;
10. $L := L' \cup \{n\}$;

end

end

For $q = 2$, the presented algorithm builds an addition chain for n. In this case, only two processors are required, one of them assigned to compute the powers of two and the other one assigned to combine these results.

In the case of a fixed exponent (that is, an exponent which is going to be used for many exponentiations), we may spend more time for finding a good (q-)addition chain.

4. FIXED-BASED TECHNIQUES

In several cryptographic systems, a fixed element a is repeatedly raised to many different powers. In such cases, pre-computing some of the powers may be an option to speed up the exponentiation.

Assume we pre-compute and store $a^{\alpha_i} \bmod m$ for some positive integers $\alpha_0, \ldots, \alpha_{k-1}$. We write $a_{\alpha_i} = a^{\alpha_i} \bmod m$. If we can decompose the exponent n as $n = \sum_{i=0}^{k-1} n_i \alpha_i$ then

$$a^n \bmod m = \prod_{i=0}^{k-1} a_{\alpha_i}^{n_i} \bmod m \qquad (6)$$

There are some techniques for choosing $\alpha_0, \ldots, \alpha_{k-1}$ and for computing products as above.

- **BGMW method**

In case $0 \leq n_i \leq h$ for some h and all $0 \leq i \leq k\text{-}1$, Brickell, Gordon, McCurly, and Wilson[11] re-iterated Yao's idea and expressed $\prod_{i=0}^{k-1} a_{\alpha_i}^{n_i} \bmod m$ as $\prod_{d=1}^{h} x_d^d \bmod m$ where $x_d = \prod_{i \mid n_i = d} a_{\alpha_i} \bmod m$. They also re-used a clever method presented in [8, answer to Exercise 4.6.3-9] for efficiently computing $\prod_{d=1}^{h} x_d^d$ as $x_d \cdot (x_d \cdot x_{d-1}) \cdots (x_d \cdots x_1)$.

The mentioned authors have also proposed two parallel versions. In the first of them, using h processors, the expressions x_d^d should be separately evaluated, and then the resulted values should be combined using the binary fan-in multiplication technique. One serious disadvantage of this variant is that each processor may require access to each of the pre-computed values a_{α_i}, so either a shared memory is used or all the pre-computed powers are stored at every processor.

The second version uses k processors, each of them computing $a_{\alpha_i}^{n_i}$ and then the resulted values are combined using the same mentioned technique.

- **De Rooij method**

De Rooij[12] found an efficient algorithm for computing the product $\prod_{i=0}^{k-1} a_{\alpha_i}^{n_i} \bmod m$ when n_i are relatively small, for all $0 \le i \le k$-1. The algorithm recursively uses the fact that

$$x^n \cdot y^m = (x \cdot y^{m \operatorname{div} n})^n \cdot y^{m \bmod n} \quad (7)$$

- **Lim-Lee Method**

Lim and Lee[13] divide the binary representation of the k-bit exponent n into h blocks w_i of size $\alpha = \lceil k/h \rceil$, for all $0 \le i \le h$-1, and then each w_i is subdivided into v blocks $w_{i,j}$, of size $\delta = \lceil \alpha / v \rceil$, for all $0 \le j \le v$-1. This can be easily done by first padding the binary representation of the exponent on the left with $(h \cdot v \cdot \delta - k)$ zeros. We obtain that

$$a^n \bmod m = \prod_{j=0}^{v-1} \left(\prod_{i=0}^{h-1} x_i^{(w_{i,j})_2}\right)^{2^{j\delta}} \bmod m \quad (8)$$

where $x_0 = a$, $x_i = x_{i-1}^{2^\alpha} \bmod m = a^{2^{i\alpha}} \bmod m$, for all $1 \le i \le h$-1. If we let $(e_{i,\alpha-1}, \ldots, e_{i,0})_2$ be the binary representation of $(w_i)_2$, for all $0 \le i \le h$-1, then $(w_{i,j})_2$ can be binary represented, for all $0 \le j \le v$-1, as $(e_{i,j\delta+\delta-1}, \ldots, e_{i,j\delta})_2$. From (8) we finally obtain that

$$a^n \bmod m = \prod_{j=0}^{v-1} \prod_{l=0}^{\delta-1} \left(\left(\prod_{i=0}^{h-1} x_i^{e_{i,j\delta+l}}\right)^{2^{j\delta}}\right)^{2^l} \bmod m \quad (9)$$

Assume now that the following values are pre-computed and stored:

$$X[0][i] = x_{h-1}^{e_{h-1}} x_{h-2}^{e_{h-2}} \cdots x_1^{e_1} x_0^{e_0} \bmod m$$

$$X[j][i] = (X[0][i])^{2^{j\delta}} \bmod m, \quad (10)$$

for $1 \le i \le 2^h$-1, $i = (e_{h-1}, \ldots, e_0)_2$, $0 \le j \le v$-1.

From (9) we finally obtain that

$$a^n \bmod m = \prod_{l=0}^{\delta-1} \left(\prod_{j=0}^{v-1} X[j][I_{j,l}]\right)^{2^l} \bmod m \quad (11)$$

where $I_{j,l} = (e_{h-1,j\delta+l}, \ldots, e_{0,j\delta+l})_2$.

We obtain the following algorithm.

LimLeeExp (*a*, *n*, *m*, *h*, *v*)

input: $n \ge 1$, m, h, $v \ge 2$, $0 < a < m$;

output: $x = a^n \bmod m$;

pre-computation: $X[j][i]$, for all $1 \le i \le 2^h$-1, $0 \le j \le v$-1 computed as in (10)

begin

1. **find** the binary representation of n;
2. **partition** it into $h \cdot v$ blocks of size δ;
3. **arrange** these $h \cdot v$ blocks in a $h \times v$ table as in Fig. 1;
4. $x := 1$;
5. **for** $l := \delta$ -1 **downto** 0 **do**

   begin

6. $x := x \cdot x \bmod m$;
7. **for** $j := v$-1 **downto** 0 **do**

   $x := x \cdot X[j][I_{j,l}] \bmod m$;

   end

end

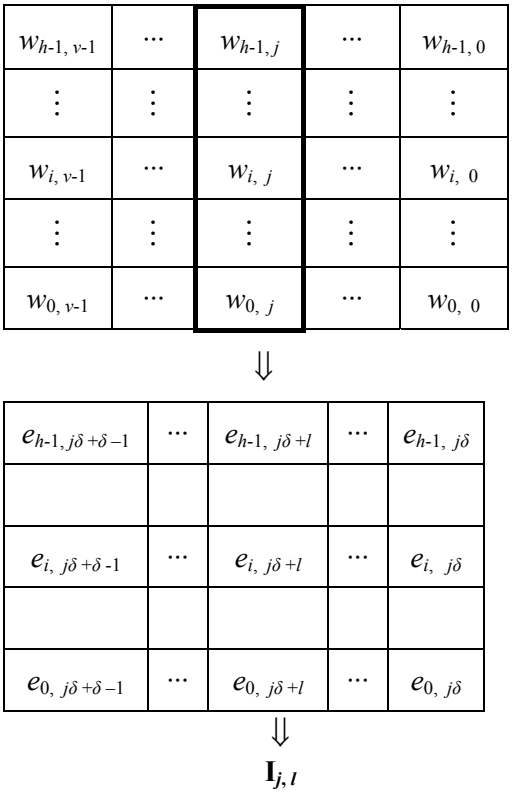

Fig. 1 The exponent processing in the Lim-Lee method

$I_{j,l}$ is the binary value of the l^{th} bit column of the j^{th} column of the mentioned table where the numbering of the columns is from right to left and it begins with 0.

The proposed method can be easily parallelized,

using v processors. Because

$$a^n \bmod m = \prod_{j=0}^{v-1}\prod_{l=0}^{\delta-1} X[j][I_{j,l}]^{2^l} \bmod m \quad (12)$$

the result of the modular exponentiation can be obtained by computing each $\prod_{l=0}^{\delta-1} X[j][I_{j,l}]^{2^l} \bmod m$ on a different processor and combining then the resulted values using the binary fan-in multiplication technique. Each processor has to store in its local memory 2^h-1 pre-computed values.

As Bernstein pointed in [14], Lim-Lee technique has already been discovered by Pippenger.

5. TECHNIQUES BASED ON MODULUS PARTICULARITIES

Suppose we have to compute $a^n \bmod m$ and we know a factorization of m, $m = m_1 \cdots m_l$, where $m_1, \ldots, m_l$ are distinct primes of about the same size. Such cases appear in the basic *RSA* decryption[15]

for $l = 2$, or in *multi-prime RSA*[16] for $l = 3$.

Because

$$a^n \bmod m_i = (a \bmod m_i)^{n \bmod (m_i - 1)} \bmod m_i \quad (13)$$

for all $1 \le i \le l$, $a^n \bmod m$ can be computed using the Chinese Remainder Theorem as the unique solution modulo $m_1 \cdots m_l$ of the system:

$$\begin{cases} x \equiv x_1 \bmod m_1 \\ \quad \vdots \\ x \equiv x_l \bmod m_l \end{cases} \quad (14)$$

where $x_i = (a \bmod m_i)^{n \bmod (m_i - 1)} \bmod m_i$, $1 \le i \le l$.

An efficient algorithm for the Chinese Remainder Theorem was proposed by Garner[17]. The first who used the Chinese Remainder Theorem for modular exponentiation were Quisquater and Couvreur[18].

We shall present next an exponentiation algorithm that can be used for the RSA-decryption, and, thus, we shall also consider that the exponent is fixed.

Mod2PrimeExp(*a*, *n*, *m*)

input: $n \ge 1$, $m \ge 2$, $0 < a < m$, $m = m_1 \cdots m_l$, where $m_1, \ldots, m_l$ are distinct primes;

output: $x = a^n \bmod m$;

pre-computation: $n_1 = n \bmod (m_1 - 1)$,

$n_2 = n \bmod (m_2 - 1)$,

$m_1^{-1} \bmod m_2$;

begin

1. $x_1 = (a \bmod m_1)^{n_1} \bmod m_1$;
2. $x_2 = (a \bmod m_2)^{n_2} \bmod m_2$;
3. $x := x_1 + m_1((x_2 - x_1)(m_1^{-1} \bmod m_2) \bmod m_2)$;

end.

The exponentiations in lines 1 and 2 can be performed using, for example, the sliding window method. Moreover, these exponentiations can be performed in parallel, using 2 processors.

6. CONCLUSIONS

In this paper we have presented the most popular methods for modular exponentiation. First, we present some techniques for general modular exponentiation. The sliding window method using only zero and odd windows is, for the optimal window size, the best choice in case of variable base and exponent. In the next section we consider the case of the fixed exponent. Finding a good (q-)addition chain may represent a serious option in this case. Moreover, particular (q-)addition chains can be combined with faster modular multiplication methods. Next we consider the case of fixed base. The BGMW method, De Rooij method and Lim-Lee method give very good results with the cost of pre-computing and storing some powers. Some particular forms of modulus are exploited in the next section. The key point is to perform more exponentiations with smaller exponents and combine the results using the Chinese Remainder Theorem.

Many of the presented algorithms lead to parallel implementations, the performance being greatly improved even in the case of a small number of available processors.

For a particular implementation, it must first choose the method which best fits with the purpose of the application and the available computational power and storage, and then experiment with the parameters to optimize performance.

Acknowledgements This work is part of the MpNT Project on designing a multi-precision number theory package. I am grateful to all members of this project for useful discussions and suggestions.

7. REFERENCES

[1] S. Iftene. Modular exponentiation. Pre-Proceedings of the NATO Advanced Research Workshop "Concurent Information Proccesing and Computing", July 5-10, 2003, Sinaia, Romania, pp. 125-144

[2] F.L. Tiplea, S. Iftene, C. Hritcu, I. Goriac, R.M. Gordan and E. Erbiceanu. MpNT: A Multi-

Precision Number Theory Package. Number-Theoretic Algorithms (I). Technical Report TR03-02 (2003), Faculty of Computer Science "Al.I.Cuza" University of Iasi. http://www.infoiasi.ro/~tr/tr.pl.cgi

[3] E.G. Thurber. On addition chains l(mn) ≤ l(n)-b and lower bounds for c(r), Duke Mathematical Journal 40 (1973), pp. 907-913

[4] B. Möller. Improved Techniques for Fast Exponentiation. Proceedings of the Workshop "Information Security and Cryptology ICISC 2002", pp. 298-312

[5] A. Brauer. On Addition Chains, Bulletin of the American Mathematical Society 45 (1939), pp. 736-739

[6] A. C. Yao. On the evaluation of powers, SIAM Journal on Computing 5 (1976), pp. 100-103

[7] D.E. Knuth. The Art of Computer Programming. Seminumerical Algorithms. Addison-Wesley, third edition, 1997

[8] D.E. Knuth. The Art of Computer Programming. Seminumerical Algorithms. Addison-Wesley, second edition, 1981

[9] D.R. Stinson. Some observations on parallel algorithms for fast exponentiation in GF(2^n). SIAM Journal on Computing 19 (4) (1990), pp. 711-717

[10] M. Nöcker. Some remarks on parallel exponentiation. Extended abstract. Proceedings of the Workshop "International Symposium on Symbolic and Algebraic Computation ISSAC 2000", pp. 250-257

[11] E. F. Brickell, D. M. Gordon, K. S. McCurley and D. B. Wilson. Fast Exponentiation with Precomputation: Algorithms and Lower Bounds,preprint,1995. http://research.microsoft.com/~dbwilson/bgmw/

[12] P. De Rooij. Efficient exponentiation using precomputation and vector addition chains. Proceedings of the Workshop "EUROCRYPT 1994", pp. 389-399

[13] C. H. Lim and P.J. Lee. More flexible exponentiation with precomputation. Proceedings of the Workshop"CRYPTO 1994", pp. 95-107

[14] D. J. Bernstein. Pippenger's exponentiation algorithm, preprint, 1991. http://cr.yp.to/papers.html

[15] R. L. Rivest, A. Shamir and L. M. Adelman. A Method for Obtaining Digital Signatures and Public-key Cryptosystems. Communications of the ACM 2 (21) (1978), pp. 120-126

[16] T. Collins, D. Hopkins, S. Langford and M. Sabin. Public Key Cryptographic Apparatus and Method. United States Patent 5, 848, 159 (1997)

[17] H. Garner. The residue number system. IRE Transactions on Electronic Computers EC-8 (1959), pp. 140-147

[18] J.-J. Quisquater and C. Couvreur. Fast decipherment algorithm for the RSA public-key cryptosystem. IEE Electronics Letters 8 (21) (1982), pp. 905-907

Sorin Iftene *has received his M.Sc. and B.Sc. degrees at Faculty of Computer Science, "Al. I. Cuza" University, Iasi, Romania, and is currently an assistant professor at the same faculty. He is interested in algebraic foundations of computer science, cryptography and computer security.*

He is presently engaged in the MpNT project at the Faculty of Computer Science, Iasi. MpNT is a free C++ library that offers high performance computations on numbers with infinite precision. The package will soon be released, and another phase of the project will commence – the development of a library offering cryptography primitives.